UNIVERSITY OF CALIFORNIA,
IRVINE

New Applications of the Nearest-Neighbor Chain Algorithm

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Nil Mamano Grande

Dissertation Committee:
Professor Michael T. Goodrich, Chair
Professor David Eppstein
Professor Sandy Irani

2019

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I would like to thank:

My advisors, David Eppstein and Michael Goodrich, for supporting me during the PhD program, for guiding this dissertation, and for their confidence in letting me pursue my own research interests (what I call "grad student tenure").

My coauthors on the papers on which this dissertation is based, with whom I've only had excellent experiences: Gill Barequet, Alon Efrat, Daniel Frishberg, Stephen Kobourov, Doruk Korkmaz, Pedro Matias, and Valentin Polishchuk.

Roger Rangel and the Balsells fellowship program, for the opportunity and financial support to come to UCI.

The faculty and colleagues at the Center for Algorithms and Theory of Computation at UCI. Sandy Irani, Kourosh Saberi, and Vijay Vazirani for serving in the defense/advancement committees.

All the people who mentored me along the way. Especially Carles Creus, Guillem Godoy, Wayne Hayes, Marc Noy, and Vera Sacristan.

Als meus pares, Anna i Carles—el suport a distància se sent més del que reconec. A la Glòria i al Ramon, per representar a la família "localment".

This work was partially supported by DARPA under agreement no. AFRL FA8750-15-2-0092 and NSF grants 1228639, 1526631, 1217322, CCF-1616248, CCF-1618301, 1815073, CCF-1740858, CCF-1712119, DMS-1839274, and DMS-1839307. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

CURRICULUM VITAE

Nil Mamano Grande

EDUCATION

Doctor of Philosophy in Computer Science **2019**
University of California, Irvine *Irvine, California, US*

Master in Computer Science **2019**
University of California, Irvine *Irvine, California, US*

Bachelor in Computer Engineering **2015**
Polytechnic University of Catalonia *Barcelona, Catalonia, Spain*

RESEARCH EXPERIENCE

Graduate Research Assistant **2019–2015**
University of California, Irvine *Irvine, California, US*

Undergraduate Research Assistant **2015**
University of California, Irvine *Irvine, California, US*

Undergraduate Research Assistant **2015–2014**
Polytechnic University of Catalonia *Barcelona, Catalonia, Spain*

TEACHING EXPERIENCE

Teaching Assistant **2018–2017**
University of California, Irvine *Irvine, CA*

REFEREED JOURNAL PUBLICATIONS

SANA NetGO: a combinatorial approach to using Gene Ontology (GO) terms to score network alignments — **Dec 2017**
W. Hayes and N. Mamano

Oxford Journals Bioinformatics

SANA: Simulated Annealing far outperforms many other search algorithms for biological network alignment — **Feb 2017**
N. Mamano and W. Hayes

Oxford Journals Bioinformatics

REFEREED CONFERENCE PUBLICATIONS

New Applications of Nearest-Neighbor Chains: Euclidean TSP and Motorcycle Graphs — **Dec 2019**
N. Mamano, A. Efrat, D. Eppstein, D. Frishberg, M.T. Goodrich, S. Kobourov, P. Matias, and V. Polishchuk

30th International Symposium on Algorithms and Computation

Stable-Matching Voronoi Diagrams: Combinatorial Complexity and Algorithms — **Jul 2018**
G. Barequet, D. Eppstein, M.T. Goodrich, and N. Mamano

45th International Colloquium on Automata, Languages, and Programming

Reactive Proximity Data Structures for Graphs — **Apr 2018**
D. Eppstein, M.T. Goodrich, and N. Mamano

13th Latin American Theoretical Informatics Symposium

Defining Equitable Geographic Districts in Road Networks via Stable Matching — **Nov 2017**
D. Eppstein, M.T. Goodrich, D. Korkmaz, and N. Mamano

25th ACM International Conference on Advances in Geographic Information Systems

Algorithms for Stable Matching and Clustering in a Grid — **Jun 2017**
D. Eppstein, M.T. Goodrich, and N. Mamano

18th International Workshop on Combinatorial Image Analysis

Models and algorithms for Graph Watermarking — **Sep 2016**
D. Eppstein, M.T. Goodrich, J. Lam, N. Mamano, M. Mitzenmacher, and M. Torres

19th Information Security Conference

ABSTRACT OF THE DISSERTATION

New Applications of the Nearest-Neighbor Chain Algorithm

By

Nil Mamano Grande

Doctor of Philosophy in Computer Science

University of California, Irvine, 2019

Professor Michael T. Goodrich, Chair

The nearest-neighbor chain algorithm was proposed in the eighties as a way to speed up certain hierarchical clustering algorithms. In the first part of the dissertation, we show that its application is not limited to clustering. We apply it to a variety of geometric and combinatorial problems. In each case, we show that the nearest-neighbor chain algorithm finds the same solution as a preexistent greedy algorithm, but often with an improved runtime. We obtain speedups over greedy algorithms for Euclidean TSP, Steiner TSP in planar graphs, straight skeletons, a geometric coverage problem, and three stable matching models.

In the second part, we study the stable-matching Voronoi diagram, a type of plane partition which combines properties of stable matchings and Voronoi diagrams. We propose political redistricting as an application. We also show that it is impossible to compute this diagram in an algebraic model of computation, and give three algorithmic approaches to overcome this obstacle. One of them is based on the nearest-neighbor chain algorithm, linking the two parts together.

Chapter 1

Introduction

Algorithm design is a core discipline in computer science that deals with finding the most efficient way to solve computational problems. There are several standard design paradigms, such as divide-and-conquer, dynamic programming, and greedy algorithms. In this work, we consider a spin on the greedy algorithm paradigm. We propose a fairly general technique for converting greedy algorithms into a new type of algorithms which we call *local greedy algorithms*. We show that, under some conditions, the local greedy algorithm finds the same solution as the greedy algorithm, but with the additional advantage that it can be implemented more efficiently. Chapter 2 focuses on proving the equivalence between greedy and local greedy algorithms, which we call *global–local equivalence*. Since it does not hold for every greedy algorithm, we characterize the necessary conditions.

To implement local greedy algorithms efficiently, we adapt an existing algorithm known as the *nearest-neighbor chain algorithm*. This algorithm was proposed in the eighties as a way to speed up certain agglomerative hierarchical clustering algorithms [24, 121]. We show that its application is not limited to clustering. Instead, its prior use in clustering can be seen as a special case of local greedy. In Chapter 3, we use it to speed up greedy algorithms for a variety of geometric and

combinatorial problems. For each problem, we show that the nearest-neighbor chain algorithm finds the same solution as a preexisting greedy algorithm, but often with an improved runtime.

We also consider local greedy algorithms for the stable matching problem, a problem that originated in market design [97]. It is well known that the stable matching problem cannot be solved by a greedy algorithm. Nonetheless, in Chapter 4 we identify a special case, which we call *symmetric stable matching*, and show that it can be solved by a greedy algorithm. We then show global–local equivalence for this greedy algorithm, and propose nearest-neighbor chain algorithms for three special cases.

Chapter 5 describes a new data structure that we use in some of our nearest-neighbor chain algorithms. This data structure adapts the notion of dynamic nearest neighbors from the geometric domain to a graph-based setting. It has other interesting applications, such as matching nearby drivers and riders in a private-driver service.

In Chapter 6, we study *stable-matching Voronoi diagrams*, a type of plane partition introduced by Hoffman et al. [113] which combines properties of stable matchings and Voronoi diagrams. We analyze their combinatorial complexity and propose algorithms to compute them. Since it is impossible to compute these diagrams exactly in an algebraic model of computation, we give three algorithmic approaches to overcome this obstacle. One of them is based on symmetric stable matching and the nearest-neighbor chain algorithm.

Finally, Chapter 7 deals with one of the applications of stable-matching Voronoi diagrams: political redistricting. This is the problem of partitioning a territory into districts with balanced population and compact shapes. We propose a solution inspired by the concept of stable matchings and test it empirically with geographic data.

1.1 Greedy algorithms

The greedy strategy is one of the most ubiquitous algorithm design paradigms (e.g., see [61, 104]). Greedy algorithms can be used for a broad class of optimization problems, but with some elements in common. The input consists of a set of mathematical objects of some kind. They can be points in a metric space, nodes in a graph, strings, and so on. The inputs elements can be arranged into solutions. The structure of a solution may vary depending on the problem. Perhaps we need to select a subset of the input elements, order them, or group them in some fashion. Each problem is characterized by two things. First, a criteria that determines which solutions are valid. For instance, in a setting where solutions are subsets of the input elements, not every possible subset is necessarily a valid solution. Second, an objective function among valid solutions. The goal is to find a valid solution optimizing the objective function.

Let us look at a classic example, the maximum-weight matching problem [19, 72, 146, 162]:

Problem 1 (Maximum-weight matching)**.** Given an undirected graph with positively-weighted edges, find a maximum-weight matching. A *matching* is a subset of edges such that each vertex is adjacent to at most one edge in the subset. The weight of a matching is the sum of the weights of its edges.

Greedy algorithms are often employed for this kind of combinatorial problems because they have an exponential number of potential solutions. For instance, the number of different subsets of a set of n elements is 2^n, while the number of orderings is $n!$. Thus, a naive exhaustive search is prohibitively expensive.

The greedy algorithm approach is to construct a solution one component at a time. At each step, we consider a set of legal choices that allow us to make progress towards a solution. Among those, we choose the one that seems best according to a heuristic *evaluation function*. This function, which the algorithm designer must devise, should evaluate the utility of each choice, i.e., how "desirable"

the choice appears in terms of reaching a good solution. Using the right evaluation function is key for the success of a greedy algorithm. The name *greedy* comes from the fact that once a choice is made, that choice is permanent; in algorithmic terms, there is no backtracking. Given its nature, a greedy algorithm should only consider "legal" choices in the sense that they should always lead to a final solution that is valid. However, a choice that seem best at an early stage (according to the evaluation function) may turn out to be suboptimal. This is why, in general, greedy algorithms are not guaranteed to find the optimal solution. Even when a greedy algorithm is not optimal, it is sometimes possible to prove an approximation ratio, i.e., prove that the quality of the greedy solution is within a certain factor of the optimal one. For background on approximation algorithms, see [182].

In the maximum-weight matching example (Problem 1), a greedy algorithm constructs a matching one edge at a time. At each step, it chooses the edge with the maximum weight among the valid edges. An edge is valid if neither of its endpoints is adjacent to an edge that has already been selected. While this greedy algorithm is not optimal, the weight of the resulting matching is at least half the weight of the optimal one [19]. This greedy algorithm is of practical interest because of its simplicity, as optimal algorithms for the maximum-weight matching problem can be intricate [146].

1.2 Local greedy

So far, we have been vague about the problems where we can use greedy algorithms. However, it is clear from the informal description that, in order to design a greedy algorithm, only two elements are necessary: first, a way to construct a valid solution by making a sequence of choices. Second, an evaluation function to rank the choices at each step.

We propose a variant of the greedy strategy that can also be applied to problems with these two elements. We call the resulting algorithms *local greedy algorithms*. In fact, the set of choices and the evaluation function of the local greedy algorithms will often be borrowed from preexisting greedy algorithms.

As in a greedy algorithm, a local greedy algorithm constructs a solution by making an irrevocable choice at each step. However, we relax the condition that we must pick the best choice at each step. In order to define the new criteria for picking a choice, we need one more ingredient. We need a notion of *interaction* between the available choices at each step. Two choices can interact in two ways. First, in terms of validity: often, making a choice means that another choice stops being compatible with a valid solution. Choices can also interact in terms of utility: making a choice might make another one less or more desirable according to the evaluation function. Not all pairs of choices necessarily interact.

For instance, in the greedy algorithm for maximum-weight matching, when we pick an edge, all the edges that share an endpoint with it become invalid. Generally speaking, the interactions depend on the problem and the evaluation function. The interaction between the choices defines an *interaction graph*: a graph with the set of valid choices as nodes and where edges represent interaction. This graph is formalized in Chapter 2.

Given the interaction graph, we can define the local greedy algorithm. For simplicity, assume that there are no ties in the evaluations of the choices. We call the choice with the highest evaluation *globally dominant*. We call a choice *locally dominant* if it has a higher evaluation than its neighbors in the interaction graph. Whereas the standard greedy algorithm makes the globally-dominant choice at each step, local greedy makes *any* locally-dominant choice. Note that there is a unique globally-dominant choice, while there can be multiple locally-dominant ones. The globally-dominant choice is also locally dominant, but the converse is not necessarily true. Henceforth, we refer to the standard greedy algorithm *global greedy* (GG) to distinguish it from local greedy (LG).

Even if GG and LG operate over the same set of choices and with the same evaluation function, they have two remarkable differences: first, local greedy is non-deterministic. If there are multiple locally-dominant choices, any of them can be chosen. Thus, one can entertain different strategies for finding locally-dominant choices. In fact, GG is one such strategy, since GG is a special case of LG. The second difference, naturally, is its locality: to implement GG, one needs to know *every* choice in order to determine the globally-dominant one. In contrast, LG can make a choice while being only aware of its neighbors.

Definition 1.1 (Local greedy algorithm for maximum-weight matching)**.** Given an undirected graph with uniquely- and positively-weighted edges, the local greedy algorithm constructs a matching by repeatedly choosing a locally-dominant edge. An edge $\{u, v\}$ is locally dominant if none of its neighboring edges (edges incident to u or v) are in the matching and $\{u, v\}$ is heavier than all of them. The constructed matching is returned when no more edges can be added.

For some problems, the interaction graph may just be a complete graph. In such cases, LG becomes GG, so the distinction is inconsequential. For instance, consider a problem where we have to choose a subset of elements under a cardinality constraint that says that the solution cannot exceed a certain size. In this case, every element interacts with every other element because they all contribute to the size of the solution. We focus on problems where the interaction graph is not a complete graph.

1.3 Global–local equivalence

At first glance, local greedy seems like a downgrade from global greedy. When both operate over the same set of choices and with the same evaluation function, at best, the choices of local greedy are as good as those of global greedy, but they can also be worse (according to the evaluation function). Our line of work begins with a perhaps unexpected observation: for many greedy algorithms,

local greedy produces the same solution as global greedy. This is known in the context of specific problems such as maximum-weight matching (see Figure 1.1), but, to our knowledge, has not been studied as a general phenomenon (see Chapter 2 for further discussion on related work). We coin the term *global–local equivalence* (GLE) for greedy algorithms with this property. A consequence of global–local equivalence is that *every run of local greedy produces the same solution*, the global greedy solution, regardless of the order in which the choices are made.

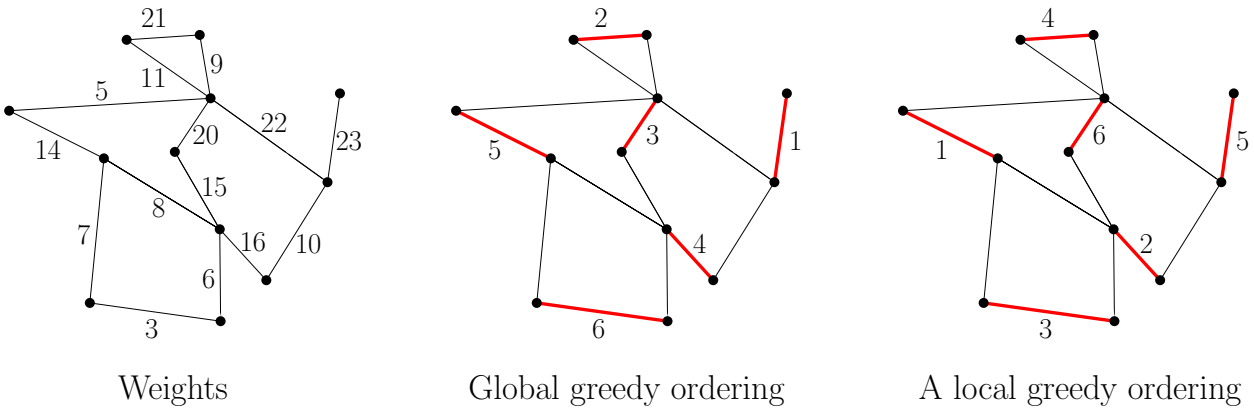


Figure 1.1: The matching and selection order found by global greedy and a possible run of local greedy for the maximum-weight matching problem for the graph on the left.

Global–local equivalence is a valuable property because any guarantee about the GG solution carries over to the solution found by LG. In particular, if GLE holds, LG achieves the same approximation ratio as GG. Our objective is to answer two main questions: which greedy algorithms have global–local equivalence? for these greedy algorithms, how can we exploit the locality of local greedy to improve the runtime?

As mentioned, there may be different strategies for implementing local greedy, as it does not specify how to choose between locally-dominant choices. This is analogous to how there are several ways to implement the Ford–Fulkerson algorithm [91] for the maximum flow problem, such as the Edmonds–Karp algorithm [75]. The Ford–Fulkerson algorithm says to choose an augmenting path, but does not specify how. Nonetheless, regardless of the order in which augmenting paths are chosen, the final computed flow is always the same. Likewise, if GLE holds, the final solution computed by any implementation of local greedy will be the same.

Our main approach for implementing local greedy is inspired by an algorithm from agglomerative hierarchical clustering: the nearest-neighbor chain algorithm. Next, we review it in its original context.

1.4 Background

1.4.1 Hierarchical clustering

Hierarchical clustering deals with the organization of data into hierarchies. Given a set of points, the *agglomerative hierarchical clustering problem* is defined procedurally as follows: each point starts as a base cluster (a cluster is a set of points). Then, a hierarchy of clusters is constructed by repeatedly merging the two closest clusters into a single one until there is only one cluster left. This creates a hierarchy where any two clusters are either nested or disjoint. A key component of hierarchical clustering is the function used to measure distances between clusters. Popular metrics include minimum distance (also known as single linkage), maximum distance (or complete linkage), and centroid distance [151, 152].

The agglomerative hierarchical clustering problem relates to our framework as follows. The process of repeatedly merging the closest pair of clusters corresponds to a global greedy algorithm; at each step, we merge a pair of clusters, so our pool of choices is the set of pairs of clusters. The globally-dominant choice is the closest pair. We can also define local greedy. We say two choices (i.e., two pairs of clusters) interact if they have a cluster in common. They interact because if clusters A and B are merged, the pair $\{B, C\}$ becomes invalid, as the cluster B becomes part of another cluster and cannot be merged individually with C. Conversely, if two pairs do not share a cluster, they do not interact: whether A and B are merged does not affect whether C and D can be merged nor the distance between C and D. A pair of clusters $\{A, B\}$ is locally dominant if their distance is minimum among any pair containing A or B. Equivalently, two clusters A and B are

locally dominant if they are *mutual nearest neighbors* (MNN), that is, they are the nearest neighbor of each other.

With this definition, local greedy is the algorithm that repeatedly merges any pair of MNN. Clearly, this may merge clusters in a different order that the standard global greedy procedure. Does global–local equivalence hold? It turns out that GLE holds if and only if the cluster distance satisfies a property called *reducibility* [39, 40, 150].

Definition 1.2 (Reducibility in agglomerative hierarchical clustering)**.** A cluster distance $d(\cdot, \cdot)$ is *reducible* if for any three clusters $A, B,$ and C such that A and B are mutual nearest neighbors,

$$d(A \cup B, C) \geq \min\left(d(A, C), d(B, C)\right).$$

In words, the new cluster $A \cup B$ resulting from merging A and B is *not* closer to other clusters than both A and B were. The relevance of this property is that, if, say, C and D are MNN, merging A and B does not break that relationship (Lemma 1.3).

Lemma 1.3 ([150])**.** *If the cluster-distance metric is reducible, any pair of clusters that are or become mutual nearest neighbors during the execution of local greedy remain mutual nearest neighbors until they are merged together.*

The consequence of Lemma 1.3 is that MNN can be merged in any order and produce the same result (see [150] for a complete proof).

Many commonly used cluster-distance metrics are reducible, including minimum, maximum, and average distance, but others such as centroid and median distance are not [150, 152]. For example, it is easy to see that the minimum-distance metric (single-linkage) is reducible. This distance is defined as the minimum distance between a point in one cluster and a point in the other. Let $A, B,$ and C be clusters. Single-linkage satisfies Definition 1.2 because the point in $A \cup B$ realizing the

minimum distance to a point in C is one of the points in either A or B. Thus, $d(A \cup B, C) = \min(d(A, C), d(B, C))$, which satisfies the definition.

To see that reducibility is the key for GLE, Figure 1.2 illustrates an instance where GG and LG find different solutions with centroid distance. The centroid of a cluster is the point such that each coordinate is the average of that coordinate among all the points in the cluster. The points a and b in Figure 1.2 are the closest pair, while c and d are MNN. We can see that there is no reducibility because the centroid of c and d is closer to b than c and d are. In fact, it is closer to b than a, so, if LG starts merging c and d, the resulting hierarchies are different.

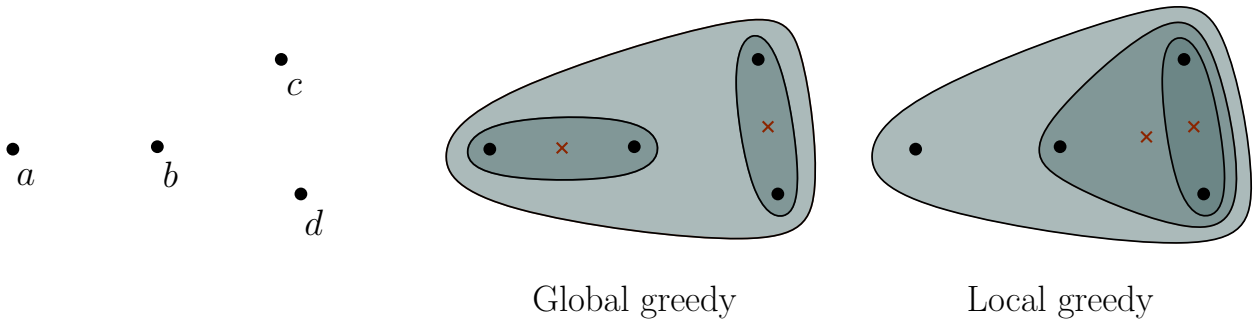


Figure 1.2: The hierarchies found by global greedy and a possible run of local greedy for the point set on the left, using centroid distance. The centroids of the inner clusters are indicated with red crosses.

1.4.2 Nearest-neighbor chain algorithm

The work on agglomerative hierarchical clustering also provides an algorithm to exploit global–local equivalence, the *nearest-neighbor chain algorithm* (NNC) [24, 121]. For extra background on NNC for hierarchical clustering, see [150, 151].

For simplicity, assume that there are no ties in the distances between clusters. The basic idea of NNC (Algorithm 1) is to maintain a stack[1], called the *chain*, of clusters. The first cluster is arbitrary. The chain is always extended with the nearest neighbor (NN) of the current cluster at

[1] A stack is a data structure where the last element inserted is the first element removed.

the top of the chain. Consequently, the distances between pairs of consecutive clusters in the chain keeps decreasing. Thus, no repeated clusters can occur, and the chain remains acyclic. Eventually, the chain reaches a pair of MNN, say, A and B. At this point, A and B are merged and removed from the chain. Crucially, after the merge happens, the rest of the chain is not discarded. Let C be one of the clusters that remain in the chain except the last one. Due to reducibility, the new cluster $A \cup B$ that results from the merge of A and B is not the NN of C. This is because reducibility states that the new cluster is not closer to C than one of A and B were, and neither A nor B were the NN of C. Thus, after the merge, the chain is still a chain of nearest neighbors: every cluster that remains in the chain is followed by its NN except the last one, which is not followed by anyone. The process continues from the new top of the chain.

Algorithm 1 Original nearest-neighbor chain algorithm for agglomerative hierarchical clustering.

```
Initialize an empty stack (the chain).
Initialize a base cluster for each input point.
while there is more than one cluster do
    if the chain is empty then
        Add an arbitrary cluster to it.
    else
        Let A be the cluster at the top of the chain.
        Find the NN B of A.
        if B is not in the chain then
            Add B to the chain.
        else                                  ▷ B must be the cluster immediately before A.
            Merge A and B into a single cluster.
            Remove A and B from the chain.
```

Runtime analysis. The most expensive part of NNC is the nearest-neighbor computations. Thus, we analyze the algorithm in terms of the number of NN computations needed as a function of the number of input points, n. Since there is one computation per iteration of the main loop (except when the chain is empty, in which case there are none) this amounts to counting the number of iterations. Let $T(n)$ be the cost of a NN computation.

Each merge reduces the number of clusters by 1, so there are $n - 1$ merges in total. Note that, for certain inputs, we could end up with a chain containing all the base clusters, so it may take $O(nT(n))$ time just to find the first pair of MNN and do the first merge. This suggests a runtime of $O(n^2T(n))$ to do all $n-1$ merges. However, thanks to not discarding the chain, we can see that the total runtime is only $O(nT(n))$.

Each iteration either adds a cluster to the chain or merges a pair of clusters. The number of iterations of the latter type is clearly $n - 1$. We only need to bound the number of iterations where a cluster is added to the chain. There are $2n - 1$ different clusters throughout the algorithm: n base ones and $n - 1$ that result from merges. Each cluster goes through the following life cycle: it comes to existence either as a base cluster or as a result of a merge. Eventually, it is added to the chain (except the very last cluster). Then, it remains in the chain until it is merged and becomes part of some other cluster. Thus, each of the $2n - 2$ clusters that are not the last one are added to the chain exactly once. Combining the two types of iterations, there are $3n - 3$ iterations in total. It follows that the runtime is $O(nT(n))$.

The runtime analysis of the nearest-neighbor chain algorithm (Algorithm 1) is relevant to our work because in Chapter 3 we give many NNC-inspired algorithms, and they all follow a similar analysis. In particular, they all finish in a linear number of iterations. However, since we will deal with different types of mathematical objects, the cost of a NN computation, $T(n)$, may differ from case to case.

Chapter 2

Global–Local Equivalence

We use the term global–local equivalence (GLE) to describe the fact that, for many greedy algorithms for optimization problems, a local greedy algorithm (LG) outputs the same solution as the normal greedy algorithm, which we call global greedy (GG). In this section, we prove GLE for several greedy algorithms of interest. To do this, we use a technique for proving GLE that we call the "hybrid method". In Chapter 3, we show how to implement LG for these problems by adapting the concept of nearest-neighbor chains.

Prior work. Global–local equivalence is known for greedy algorithms for graph problems such as maximum-weight matching [112], b-edge cover [129][2], and minimum dominating set [138]. In these prior works, GLE is exploited to design parallel and distributed algorithms. Global–local equivalence for these problems is a direct consequence of Theorem 2.9 in this chapter, but these prior works prove GLE for their specific greedy algorithms directly without pointing to any prior result along the lines of Theorem 2.9. To our knowledge, these results have not been connected in a single framework before.

[2]We adopt the term "locally dominant" from their paper.

As mentioned in Section 1.4.1, GLE is also known for agglomerative hierarchical clustering. Müllner [150] proved global–local equivalence in the context of agglomerative hierarchical clustering for reducible cluster distances, and introduced what we call the hybrid method in that setting.[3]

Maximum-weight matching. To motivate the hybrid method, first we give a simple proof of global–local equivalence for maximum-weight matching (Problem 1). A proof in the context of distributed protocols is given in [112]. Preis [162] proves that LG is a 2-approximation[4] directly without noting that the solution is actually the same as that of GG.

Recall the definition of local greedy for maximum-weight matching (Definition 1.1).

Lemma 2.1. *Let G be a weighted, undirected graph with unique positive weights. Let M_1, M_2 be any two matchings in G obtained by the local greedy algorithm for maximum-weight matching. Then, $M_1 = M_2$.*

Proof. Assume, for a contradiction, that $M_1 \neq M_2$. Let e be the heaviest edge in one (without loss of generality, M_1) but not the other (M_2). During the course of the algorithm, every edge is either added to the matching or removed from the graph because a heavier neighbor was picked. Since e is not in M_2, M_2 contains a neighbor edge e' of e that is heavier than e (see Figure 2.1). However, e' cannot be in M_1 because e is in M_1. This contradicts the assumption that e was the heaviest edge in only one of the matchings. □

Corollary 2.2 (Global-local equivalence in maximum-weight matching)**.** *The global greedy and local greedy algorithms for maximum-weight matching produce the same solution.*

[3] Even though global–local equivalence for agglomerative hierarchical clustering was known much earlier, Müllner [150] mentions that he could not find a formal proof. We also could not find an earlier proof, but it might be because some of the early papers on agglomerative hierarchical clustering are in French.

[4] An algorithm for an optimization problem is an α-approximation if, for every input, the optimal solution for that input is at most α times better than the solution outputted by the algorithm (e.g., see [182]).

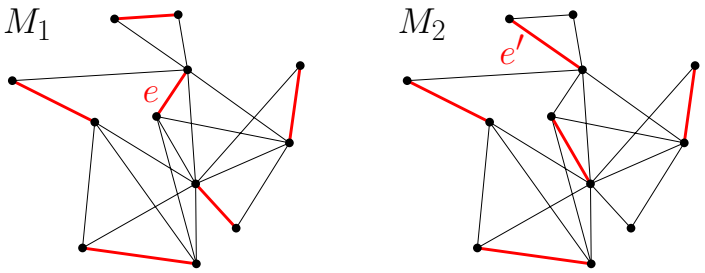


Figure 2.1: Setup in the proof of Lemma 2.1.

Proof. The maximum-weight valid edge is locally dominant, so GG is a special case of LG. By Lemma 2.1, they produce the same matching. □

Note that the proof of Lemma 2.1 relies on the uniqueness of the weights. This is always an important consideration for GLE. Without this assumption, GG itself could easily end with different matchings, and GLE would be more intricate to define.

Set cover. The simple argument in the proof of Lemma 2.1 is not powerful enough to prove GLE for greedy algorithms with more sophisticated evaluation functions. In particular, it relies on the ability to identify the heaviest edge where the matchings differ. However, many greedy algorithms use evaluation functions such that the evaluation of a choice depends on the current partial solution. Thus, the evaluations evolve over time. This makes it impossible to pinpoint the first discrepancy. Consider, for instance, the classic $O(\log n)$-approximation greedy algorithm for the set cover problem [56, 182], which is NP-complete [125].

Problem 2 (Set cover)**.** Given a set U and a collection X of subsets of U with positive weights, find a cover of U of minimum weight. A *cover* is a subset of X such that every element of U is contained in one of the sets. The weight of a cover is the sum of the weights of its sets.

Some famous problems are special cases of set cover, such as vertex cover (case where all the elements appear in only two sets) and edge cover (case where all the sets have size two).

The $O(\log n)$-approximation greedy algorithm for set cover uses an evaluation function known as *cost-per-element*. The cost-per-element of a set is the weight of the set divided by the number of still-uncovered elements in it (or infinite, if they are all covered). Note that the cost-per-element of a set increases throughout the course of the algorithm as the elements it contains are covered as part of other sets.

The normal greedy algorithm (GG) repeatedly picks the set minimizing the cost-per-element until all elements are covered. We define local greedy similarly.

Definition 2.3 (Local greedy for set cover)**.** Given a set cover instance, local greedy constructs a cover by repeatedly picking a locally-dominant set at each iteration. Given the partial solution picked by local greedy at a given iteration, a set s is locally dominant if it has a smaller cost-per-element than any set s' with an element in common with s. The constructed cover is returned when it covers every element.

To prove the equivalence between GG and LG for set cover, we use the hybrid method. In fact, we prove GLE for a more general class of combinatorial optimization problems.

2.1 Combinatorial optimization problems

We consider a class of combinatorial optimization problems (COPs) where there is a set from which we have to pick a subset. There is some constraint that subsets must satisfy to be valid solutions. There is also an objective function f that assigns values to the valid solutions. The goal might be to maximize or minimize f. In the specific examples considered in this section, f is *modular*, which means that the elements have positive weights, and the value of a solution is the sum of the weights of the elements in it. More generally, we consider monotonically-increasing objective functions: if S is a subset of T, then $f(S) \leq f(T)$.

Definition 2.4 (Class of combinatorial optimization problems)**.** A problem is a combinatorial optimization problem if it can be stated in the following terms. There is a goal, which can be maximization or minimization. Each input consists of a set, X, and the information needed to compute a monotonically-increasing objective function f and a validation function $\mathcal{V} : 2^X \to \{0, 1\}$ over the subsets of X.[5] The goal is to find the subset S of X maximizing or minimizing $f(S)$, depending on the goal, subject to $\mathcal{V}(S) = 1$.

It is clear from Definition 2.4 that maximum-weight matching and set cover (Problems 1 and 2) are COPs. The monotonicity of the objective function suggests a greedy strategy: start with the empty solution, and add one element at a time until no more can be added (for maximization problems), or until the solution becomes valid (for minimization problems). Note that if we commit to our picks—as per the greedy philosophy—then, once we reach the corresponding point in each case, there is nothing more we can add to the solution that will improve it.

In order to have a greedy algorithm for a COP, it only remains to provide an evaluation function to evaluate the unpicked elements and choose the best one. Formally, we need a function $h(a, S)$ which takes an element a of the input set, X, and a partial solution S not containing a, $S \subseteq X - a$.[6] For a maximization (resp. minimization) problem, the evaluation function h should map a and S to a real number measuring "how good (resp. how costly) it is to add a to S". If $S + a$ is not a valid solution nor a subset of any valid solution, we say a is invalid for S. A greedy algorithm should never pick an invalid element, so, if a is invalid for S, assume that $h(a, S) = -\infty$ in the case of maximization and $h(a, S) = +\infty$ in the case of minimization. Note that, by definition, h is insensitive to the order in which the elements of S were added to S, as the order does not affect the value of $h(a, S)$.

Given an evaluation function h for a maximization COP (resp. minimization COP) we can define the corresponding global greedy: start with an empty solution, $S = \emptyset$, and pick the globally-

[5]The notation 2^X indicates the set of all the subsets of X.

[6]When S is a set and a an element, we use $S + a$ and $S - a$ to denote $S \cup \{a\}$ and $S \setminus \{a\}$, respectively.

dominant element at each iteration. Given the partial solution, S, computed so far by global greedy at some iteration, we say that an element a is globally dominant if $h(a, S)$ is better than any other element b ("better" means $h(a, S) > h(b, S)$ for maximization COPs and $h(a, S) < h(b, S)$ for minimization COPs). The constructed solution is returned when no more elements can be added (for maximization COPs) or when the solution becomes valid (for minimization COPs).

For our purposes, we require GG to be deterministic, i.e., we require the globally-dominant element to be unique. Thus, we assume that there are no ties among the values of h. This can be achieved using a tie-breaking rule, such as associating indices from 1 to $|X|$ to the elements of X, and breaking ties according to the lowest index.

There can be multiple evaluation functions for the same combinatorial optimization problem. A natural but perhaps naive option is the *marginal benefit/cost* of adding a to S: $h(a, S) = f(S+a) - f(S)$. This is used in the 2-approximation greedy algorithm for maximum-weight matching that we discussed in Section 1.1. More sophisticated evaluation functions may also take into account other factors, such as the effect of picking a on the validity of other elements (e.g., see Section 2.1.1).

To define local greedy based on an evaluation function h, we first define the interaction graph induced by h.

Definition 2.5 (Interaction graph for combinatorial optimization problems)**.** Let X be the input set of a combinatorial optimization problem (Definition 2.4), and h an evaluation function for the problem. The interaction graph of X induced by h is an undirected graph where each node corresponds to an element of X and each edge corresponds to two elements that interact. Two elements a and b of X *interact* if there is a set $S \subseteq (X \setminus \{a, b\})$ such that $h(a, S) \neq h(a, S + b)$. That is, the presence of b in the solution S affects the evaluation of a.

We model the interaction between the elements of the interaction graph as undirected for simplicity (a and b interact if either affects the evaluation of the other). We can now define a generic local greedy based on h.

Definition 2.6 (Local greedy for combinatorial optimization problems)**.** Let h be an evaluation function for a combinatorial optimization problem (Definition 2.4). Local greedy starts with an empty solution, $S = \emptyset$, and picks a locally-dominant element at each iteration. Given the partial solution, S, computed so far by local greedy at some iteration, we say that an element a is locally dominant if $h(a, S)$ is better than $h(b, S)$ for any neighbor b of a in the interaction graph induced by h. The constructed solution is returned when no more elements can be added (for maximization COPs) or when the solution becomes valid (for minimization COPs).

Let h be an evaluation function for a COP. We want to characterize when the global and local greedy algorithms based on h are guaranteed to output the same solution. Next, we prove that, for GLE to hold, h needs to satisfy a condition which plays the same role as reducibility for cluster distances (Definition 1.2) in agglomerative hierarchical clustering. In the context of COPs, we will see that we have GLE if the evaluation of the elements cannot get better throughout the course of the algorithm. We say that h is *deteriorating*.

Definition 2.7 (Deteriorating evaluation function)**.** Let h be an evaluation function for of a maximization (resp. minimization) combinatorial optimization problem with input set X. We say that h is deteriorating if for all $a \in X, S \subseteq X - a$ and $S' \subset S$, we have $h(a, S) \leq h(a, S')$ (resp. $h(a, S) \geq h(a, S')$).

Lemma 2.8. *Let h be a deteriorating evaluation function for a combinatorial optimization problem. Any element that is or becomes locally dominant during the execution of local greedy remains locally dominant until it is picked by local greedy.*

Proof. Let a be a locally-dominant element at some iteration of LG. The neighbors of a in the interaction graph are not locally dominant due to the presence of a, so they cannot be picked by LG in the current iteration. Note that the evaluation of a can only change when one of its neighbors is picked. Thus, if LG does not pick a in the current iteration, it picks some element that is not a neighbor of a, and the evaluation of a does not change. The evaluation of the neighbors of

a may change, but, since h is deteriorating, they cannot become better than a. Thus, a is either picked or remains locally dominant after the current iteration. This reasoning applies to subsequent iterations, so, eventually, a is picked by LG. □

To prove GLE for COPs (Theorem 2.9), we introduce the main technique, the hybrid method. Recall the no-tie assumption for the evaluations. In the event of ties, the key is that both GG and LG break ties consistently. That is, both GG and LG favor the same element in case of a tie.

Theorem 2.9 (Global–local equivalence for combinatorial optimization problems)**.** *Let h be a deteriorating evaluation function for a combinatorial optimization problem. Then, the global and local greedy algorithms based on h output the same solution.*

Proof. Let $L = l_1, l_2, \ldots, l_m$ be the sequence of elements picked by an arbitrary run of local greedy. Consider a run of local greedy which is a hybrid between this specific run of LG and GG, denoted $hybrid_i$ for some $i \in \{1, \ldots, m+1\}$: $hybrid_i$ starts picking the elements in L, in order, up to l_{i-1}. Then, it switches to GG and, from iteration i onward, picks the globally-dominant element.

The main claim is that, for any $i \in \{1, \ldots, m\}$, $hybrid_i$ and $hybrid_{i+1}$ find the same solution. By applying this result repeatedly, we get that $hybrid_1$ and $hybrid_{m+1}$ find the same solution. Note that $hybrid_1$ is just GG, and that the solution of $hybrid_{m+1}$ is L. Thus, if we can prove the main claim, we get that the solution found by GG is the same as the solution found by an arbitrary run of LG (L).

Figure 2.2, Left, shows the setup for the main claim. We label the elements picked by $hybrid_i$ starting at iteration i with g_i, g_{i+1}, and so on. We label the elements picked by $hybrid_{i+1}$ starting at iteration $i+1$ with g'_{i+1}, g'_{i+2}, and so on. Figure 2.2, Right, shows what actually happens with the elements picked by the two hybrid runs.

First, note that $hybrid_i$ *eventually* picks l_i. That is, $l_i = g_j$ for some $j \geq i$. This follows from Lemma 2.8.

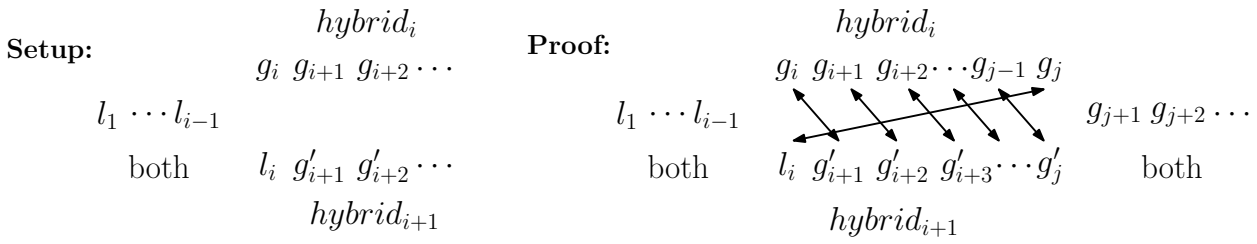


Figure 2.2: **Left:** the sequence of elements picked by $hybrid_i$ and $hybrid_{i+1}$. They agree up to and including iteration $i-1$, and then $hybrid_i$ switches to global greedy an iteration early. **Right:** what we prove about the elements picked by the two hybrid methods. The arrows indicate that these elements are the same despite being labeled differently. The two methods coincide up to and including iteration $i-1$ and after iteration j.

It is also clear that, after picking l_i, $hybrid_{i+1}$ picks g_i, i.e., $g'_{i+1} = g_i$. This is because g_i is globally dominant before picking l_i, and so it must still be after picking l_i. More generally, $g'_{k+1} = g_k$ for all $k \in \{i, \ldots, j-1\}$. This is easy to see by induction on k, with the case $k = i$ that we already mentioned as base case. For the recursive case, note that l_i is not a neighbor of any of $g_i, \ldots, g_{j-1}$, so "hoisting" the pick of l_i before them does not change their evaluation. Hoisting l_i may change the evaluation of their neighbors, but, again by Lemma 2.8, their evaluation only gets worse, so g'_{k+1} remains globally-dominant at iteration $k+1$.

After iteration j, both hybrids have picked exactly the same elements, and both have switched to GG. Thus, from that point on, they coincide in their picks. At the end, both have picked the same elements. □

Corollary 2.10. *Every possible run of a local greedy algorithm based on a given deteriorating evaluation function outputs the same solution.*

Proof. By Theorem 2.9, they all output the same solution as GG, which is deterministic. □

For maximum-weight matching (Problem 1), we proved GLE directly (Corollary 2.2). For set cover, recall that GG chooses sets minimizing the "cost-per-element": the weight of the set divided by the number of uncovered elements in the set. This rule is deteriorating, as the cost per uncovered element of a set can only go up. According to this rule, interacting sets are those with an element

in common (see Figure 2.3), and we see that the local greedy for set cover from Definition 2.3 is a special case of the general local greedy for COPs (Definition 2.6). Since the evaluation function is deteriorating, global–local equivalence follows from Theorem 2.9.

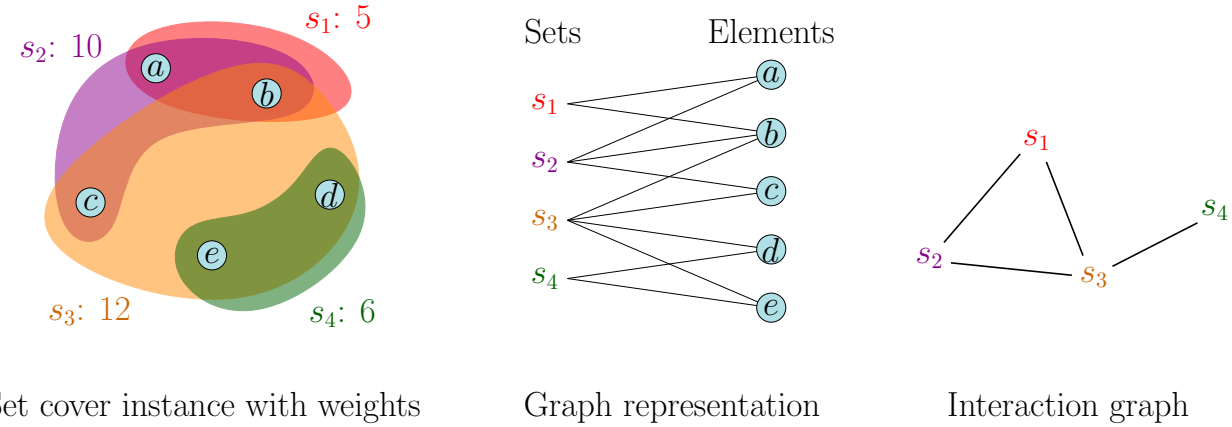


Figure 2.3: Interaction graph for a set cover instance.

2.1.1 Maximum-weight independent set

We use maximum-weight independent set, another classic NP-complete combinatorial optimization problem [125], to illustrate that not all evaluation functions are deteriorating.

Problem 3 (Maximum-weight independent set)**.** Given an undirected graph with positively-weighted nodes, find an independent set of maximum weight. An *independent set* is a subset of nodes such that no two nodes in the subset are neighbors. The weight of an independent set is the sum of the weights of the nodes in it.

Given an independent set S, a node u is valid for S if it is not in S nor a neighbor of a node in S. A greedy algorithm for maximum-weight independent set selects, at each iteration, a valid vertex u maximizing $h(u, S) = w(u)/(\delta(u, S) + 1)$, where $w(u)$ is the weight of u, S is the set of nodes selected so far, and $\delta(u, S)$ is the number of valid neighbors of u for S. This rule generalizes the intuition for the unweighted case that we should pick a node that invalidates the minimum number of other nodes. It achieves an approximation ratio of $1/\Delta$, where Δ is the maximum degree of the

graph. This ratio is tight, meaning that there are graphs for which the greedy algorithm outputs a solution of weight no better than $1/\Delta$ times the weight of the optimal solution [170].

Suppose that a, b, and c are valid nodes for a solution S such that a is adjacent to b and b to c, but a is not adjacent to c. If a is picked, the evaluation of c improves. That is, $h(c, S+a) > h(c, S)$. This is because the effect of picking a is that c invalidates one fewer nodes (b). The interaction graph induced by h connects pairs of nodes that are neighbors or have a neighbor in common. Thus, LG chooses any valid node with a higher value of h than any valid node at distance at most two. Since the evaluation function h is not deteriorating, there is no global–local equivalence. See Figure 2.4 for a graph where GG and LG may differ.

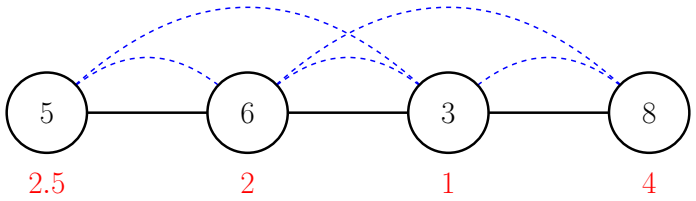


Figure 2.4: Graph where the nodes are labeled with their weights. Below each node u, the value $h(u, \emptyset)$ is shown in red. The edges of the interaction graph induced by h are shown dashed in blue. Initially, the locally-dominant nodes are the ones with weight 5 and 8. The valid run of LG that starts by picking the node with weight 5 outputs the solution $\{5, 8\}$, whereas GG outputs the solution $\{8, 6\}$.

We do not claim that local greedy algorithms are not interesting without global–local equivalence—only that the solution may be different from the global-greedy solution. In fact, it follows from the results of Sakai et al. [170] that the local greedy algorithm based on h achieves the same approximation ratio as GG, $1/\Delta$. Their result is even stronger: to achieve this approximation ratio, it suffices to pick any node u such that $h(u, S)$ is larger than the average value of $h(v, S)$ among the valid neighbors v of u in the input graph.

Another global greedy algorithm for this problem repeatedly selects the node u minimizing $h'(u) = w(u)/(\delta'(u, S)(\delta'(u, S) + 1))$, where $\delta'(u, S)$ is the number of neighbors of u that have not been selected yet. This global greedy ends when every edge is incident to at least one selected node, and outputs the set of non-selected nodes. We call this a *reverse greedy* algorithm, since it outputs the

complement of the selected elements. This reverse GG also achieves a $(1/\Delta)$-approximation [170]. Since its goal is minimization, the fact that the values of h' can only get higher over time means that h' is deteriorating. Thus, unlike for h, for h' we have global–local equivalence.

Similarly to the result for h, Sakai et al. [170] showed that, to achieve the approximation ratio of $1/\Delta$, we can relax the reverse GG algorithm and select any node u such that $h'(u, S)$ is smaller than the average value of h' among the neighbors of u. This policy further relaxes the policy of LG, so we call the algorithm *soft local greedy*. The soft local greedy loses global–local equivalence (See Figure 2.5).

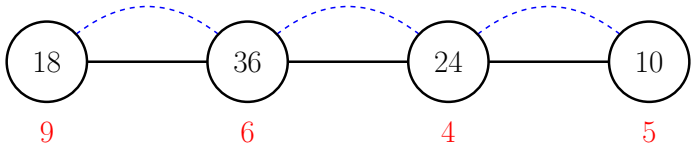


Figure 2.5: Graph where the nodes are labeled with their weights. Below each node u, the value $h'(u, \emptyset)$ is shown in red. The edges of the interaction graph are shown dashed in blue, and are the same as the input graph. The soft local greedy based on h' could pick (for elimination) the node with evaluation 6, because the average evaluation of its neighbors is 6.5. However, this node would not be picked by the local greedy algorithm based on h'.

Thus, we have the following hierarchy of algorithms based on the evaluation function h', where the more we relax the algorithm, the weaker the guarantee we get. Assuming no ties on the values of h':

- Global greedy: achieves a $(1/\Delta)$-approximation.
- Local greedy: outputs the same solution as GG, but may not compute it in the same order.
- Soft local greedy: may not output the same solution as GG, but still achieves a $(1/\Delta)$-approximation.

Since our focus is on global–local equivalence, we do not study the idea of relaxing local greedy further. It could be an interesting direction for future work.

2.2 Multi-fragment TSP

In this section, we prove GLE for a greedy algorithm for a famous NP-complete optimization problems, the traveling salesperson problem (TSP) [136].

Recall that a *path* in a graph is a non-empty sequence of nodes such that every two consecutive nodes are connected via an edge, and no node is visited twice. A *cycle* is a path that starts and ends at the same vertex.

Problem 4 (Traveling Salesperson Problem)**.** Given an undirected, complete graph with uniquely- and positively-weighted edges, find a cycle passing through all the nodes of minimum weight. The weight of a cycle is the sum of the weights of the edges in it.

Given two disjoint paths p and p', we define the cost of connecting them, $\text{cost}(p, p')$, as the weight of the cheapest edge between an endpoint of p and an endpoint of p'. We use $p \cup p'$ to denote the path resulting from connecting p and p' into a single path along that edge.

A greedy algorithm for this problem, known as the *multi-fragment algorithm*, works as follows [21]. We maintain a set of disjoint paths. We start with a single-node path for each node. While there is more than one path, we connect the two paths such that the cost of connecting them is minimum. Once there is a single path left, we connect its endpoints to form a cycle and return that cycle. More background on the multi-fragment algorithm is given in Section 3.3, where we implement a local greedy algorithm to construct the same tour.

The traveling salesperson problem can be modeled as a COP where the set of input elements, X, is the set of edges. However, the resulting interaction graph as defined in Definition 2.5 is not useful because it would be a complete graph. This is because any two edges, $\{a, b\}, \{c, d\}$, interact even if they do not have a common endpoint. To see this, note that a partial solution could contain paths from a to c and from b to d. Then, adding $\{a, b\}$ would make $\{c, d\}$ invalid, because c and d would be endpoints of the same path. If the interaction graph is a complete graph, then GG and LG are

the same. Hence, we model global–local equivalence for this problem differently. Instead of a static interaction graph like the one in Definition 2.5, we consider a graph that evolves with the constructed solution. Since the multi-fragment algorithm operates on paths rather than edges, the nodes of the interaction graph correspond to pairs of paths in the current set of paths.

Definition 2.11 (Multi-fragment local greedy for TSP)**.** Given an undirected, complete graph with positively-weighted edges, local greedy starts with a single-node path for each node. Then, it repeatedly connects a locally-dominant pair of paths until there is a single path left. We say a pair of paths, p, p' is locally dominant in a set of paths if the cost of connecting p and p' is lower than the cost of connecting either with a third path. Finally, local greedy connects the two endpoints of the remaining path and returns the cycle.

We call the pair of paths realizing the minimum connection cost globally dominant. As usual, GG is a special case of LG.

Note the similarity between multi-fragment TSP and hierarchical clustering. Instead of clusters, we merge paths. A difference is that, when determining the distance between clusters, it does not matter in which order the points were added to the clusters. In contrast, in a path, it is important which nodes are the endpoints. Nonetheless, in multi-fragment TSP we have a notion equivalent to reducibility in agglomerative hierarchical clustering (Definition 1.2).

Lemma 2.12 (Reducibility in multi-fragment TSP)**.** *Let $a, b,$ and c be paths in an undirected, complete graph with positively-weighted edges. Then, cost$(a \cup b, c) \geq \min$ (cost(a, c), cost(b, c)).*

Proof. The cost of connecting two paths is defined as the minimum weight among the edges connecting their endpoints. The claim is clear given that the two endpoints of $a \cup b$ are a subset of the four endpoints of a and b. □

We use this to give an analog of Lemma 2.8 for multi-fragment TSP.

Lemma 2.13. *Any pair of paths p, p' that is or becomes locally dominant during the multi-fragment local greedy algorithm remains locally dominant until they are connected to each other.*

Proof. By definition of locally dominant, it is cheaper to connect p and p' to each other than to any third path. By reducibility (Lemma 2.12), this does not change if LG connects any other pair of paths, as the resulting path will not be closer to p nor p' than at least one of the original paths were. Eventually, p and p' are connected by local greedy. □

Theorem 2.14 (Global-local equivalence in multi-fragment TSP)**.** *Given an undirected, complete graph with uniquely- and positively-weighted edges, the multi-fragment global and local greedy algorithms output the same solution.*

Proof. We use the hybrid method. Let $L = l_1, l_2, \ldots, l_m$ be the sequence of pairs of paths merged by a *specific* run of local greedy. Consider a run of local greedy, denoted $hybrid_i$ for some $i \in \{1, \ldots, m+1\}$: $hybrid_i$ starts merging the pairs in L, in order, up to and including l_{i-1}. Then it switches to merging globally-dominant pairs.

Let $i \in \{1, \ldots, m\}$. We show that the hybrid run that switches before iteration i ($hybrid_i$) finds the same solution as the hybrid run that switches after iteration i ($hybrid_{i+1}$). Using this result, by induction on i we get that $hybrid_1$ and $hybrid_{m+1}$ output the same solution. Since $hybrid_1$ corresponds to GG and $hybrid_{m+1}$ outputs L, the theorem follows.

We label the pairs merged by $hybrid_i$ starting at iteration i with g_i, g_{i+1}, and so on. We label the pairs merged by $hybrid_{i+1}$ starting at iteration $i+1$ with g'_{i+1}, g'_{i+2}, and so on (analogously to Figure 2.2 for the proof of GLE for COPs).

First, note that $hybrid_i$ *eventually* merges l_i. That is, $l_i = g_j$ for some $j \geq i$. This follows from Lemma 2.13.

Further, $g'_{k+1} = g_k$ for all $k \in \{i, \ldots, j-1\}$. This can be shown by induction on k. By Lemma 2.13 again, the path resulting from merging the pair l_i is not closer to any of the paths involved in any of the pairs $g'_{i+1}, \ldots, g'_j$, than one of the two paths in l_i. It follows that "hoisting" the merge of l_i before them does not change the fact that g'_k is globally dominant at iteration k.

After iteration j, both hybrid runs have merged exactly the same pairs, so they have the same partial solutions, and both have switched to GG. Thus, from that point on, they coincide in their merges, and finish with the same solution. □

Corollary 2.15. *Every possible run of the multi-fragment local greedy algorithm for TSP outputs the same solution.*

2.3 Shortest common superstring

Another classic optimization problem is the shortest common superstring problem (SCS) [166].

Problem 5 (Shortest common superstring)**.** Given a set of strings from an alphabet, find the shortest string that is a superstring of all of them. A string is a superstring of another if it contains it consecutively, that is, without additional characters intertwined.

The problem is NP-complete [166], but some approximation algorithms are known. Of these, the most famous is a greedy algorithm simply known as Greedy (in our context, we call it global greedy). This algorithm maintains a set of strings. Initially, this set is just the input set. Then, it repeatedly takes two strings in the set with the longest overlap and replaces them with the shortest superstring of both. We call this *merging* two strings. Global greedy finishes when there is a single string left, which is returned. This algorithm is similar to the multi-fragment algorithm for TSP. In fact, the problem can be reduced to a variant of TSP called maximum asymmetric TSP [159]: the input strings become nodes in a complete, directed graph, where the weight of the edges corresponds to the overlap between the end of a string and the beginning of another.

More background on the different algorithms for SCS is provided in Section 3.7. Here we focus on proving global–local equivalence.

Formally, the input is a string set, S_0, of size $m \geq 1$. We assume that no string in S_0 is a substring of another (in particular this also excludes duplicate strings and the empty string). If present, substrings can be handled in a preprocessing step. We call the strings in S_0 *input strings*. Below, we define a local greedy algorithm for SCS. Both GG and LG start with S_0 and operate by merging strings. Given some sequence of string merges, we use S_i to denote the string set after i merges. Regardless of the strategy used to merge strings, the set S_{m-1} consists of a single string, which is returned.

The issue of breaking ties. To be able to prove GLE, we need a tie-breaking rule for string overlaps that is used consistently by both GG and LG. The issue is not trivial because new strings appear as a result of merges. In addition, ties cannot simply be regarded a "corner case" because string overlaps are integer values.

For strings s_j, s_k in S_0, let $o(s_j, s_k)$ be the length of the maximum overlap between a suffix of s_j and a prefix of s_k. E.g., $o(ab, bc) = 1$ and $o(bc, ab) = 0$. We would like the overlap values to be unique among all string pairs in S_0, but that is generally not the case. Thus, we modify the overlap values in a way which breaks ties but preserves the true differences. More formally, we assume that each string s_j in S_0 has a unique index j from 1 to m. Let $\tau(j, k)$ be an arbitrary injective function from $\{1, \ldots, m\} \times \{1, \ldots, m\}$ to $(0, 1)$, such as $\tau(j, k) = j/(m+1) + k/(m+1)^2$. The *adjusted overlap* $o'(s_j, s_k)$ is $o'(s_j, s_k) = o(s_j, s_k) + \tau(j, k)$ (the specific function τ does not matter; the example given corresponds to breaking ties by the index of the first string, and then by the index of the second string). Note that all the adjusted overlaps are unique, and that $o(s, t) < o(u, v)$ implies $o'(s, t) < o'(u, v)$.

We extend the definition of adjusted overlap to pairs of strings u, v in the set S_i at any iteration, $i \in \{0, \ldots, m-1\}$. The strings u, v are the result of merging input strings (strings in S_0) zero or more times. For a string $s \in S_i$, let $first(s)$ and $last(s)$ denote the first and last input strings in s, respectively (if s itself is an input string, $first(s) = last(s) = s$). Then,

$$o'(u, v) = o(u, v) + \tau(index(last(u)), index(first(v))).$$

We use $o^*(u, v)$ to denote the maximum adjusted overlap between u, v without a specific order: $o^*(u, v) = \max(o'(u, v), o'(v, u))$. We use both $merge(u, v)$ and $merge(v, u)$ to denote the string that results from merging u and v in the order that maximizes the adjusted overlap between them (that is, the order of the arguments of $merge(\cdot, \cdot)$ is inconsequential). For example, if two strings u and v overlap by three letters in either order, but $o'(u, v) = 3.5$ and $o'(v, u) = 3.7$, they are merged such that v appears first.

Note that GG has some freedom of choice in what string pair to merge in case of a tie for the most overlap. This can easily lead to different outputs, e.g. with input $\{ab, bc, ca\}$. Henceforth, when we talk about GG, we assume that, in the case of ties, it chooses the pair of strings with the largest adjusted overlap, and merges them in the order that maximizes the adjusted overlap. We call this the globally-dominant pair. The rule of choosing the globally-dominant pair, as defined, makes GG completely deterministic, and corresponds to one of the possible sequences of choices that its non-deterministic version could make. When we define LG and show global–local equivalence, it will be with respect to this specific run of GG.

Local greedy. We define the *first choice* of a string s in S_i as the string in S_i (other than s itself) that overlaps the most with s. We define the first choices of the strings in terms of adjusted overlaps

so that they are unique and without ambiguity:

$$\text{first-choice}(s_j) = \arg\max_{s_k \in S \setminus \{s_j\}} o^*(s_j, s_k)$$

Definition 2.16 (Local greedy for shortest common superstring)**.** Let S_0 be a set of m strings such that no string is a substring of another. For $i = 0, \ldots, m-2$, let u, v be any locally-dominant pair of strings in S_i, and set $S_{i+1} = (S_i \setminus \{u, v\}) \cup \{merge(u, v)\}$. We say two strings in S_i are *locally dominant* if each one is the first choice of the other. Finally, return the single string remaining in S_{m-1}.

Proof of global–local equivalence. Henceforth, we use $\mathcal{S} = S_0, S_1, \ldots, S_{m-1}$ to denote a sequence of string sets such that S_0 is a valid input to SCS, of size $m \geq 1$, and each S_i, $i > 0$, can be obtained from S_{i-1} by merging a locally-dominant pair. In other words, $\mathcal{S}$ corresponds to the sequence of string sets in a valid run of local greedy (Definition 2.16). We now give some results about the properties of the sets S_i.

Lemma 2.17. *Let s, t be distinct strings in $S_i \in \mathcal{S}$. Then, s is not a superstring of $first(t)$ nor $last(t)$.*

Proof. We proceed by induction on i. The claim is true for S_0 by the assumption that no input string is a substring of any other. Suppose the claim is true for $i = k \geq 0$, and, on iteration k, the locally-dominant pair u and v are merged with u before v, so that $S_{k+1} = (S_k \setminus \{u, v\}) \cup \{merge(u, v)\}$. Let s, t be distinct strings in S_{k+1}. We show that s is neither a superstring of $first(t)$ nor $last(t)$. We must consider the following possible cases (the claim is immediate in the first two):

1. $s \neq merge(u, v)$ and $t \neq merge(u, v)$. Then, s and t were in S_k, so the claim follows by induction.

2. $t = merge(u, v)$. Then, s was in S_k and $last(t) = last(v)$, and, by induction hypothesis, $last(v)$ is not a substring of s. Similarly with $first(t)$.

3. $s = merge(u, v)$. Then, assume for a contradiction that $first(t)$ is a substring of $merge(u, v)$ (the case of $last(t)$ is symmetric). Then, since by the inductive hypothesis $first(t)$ is not a substring of u or v, $first(t)$ must contain the entire overlap of u and v as well as parts of both u and v which are not in the overlap (β and δ in Figure 2.6). This implies that, in S_k, $o(u, t) > o(u, v)$, contradicting the fact that u and v are locally dominant.

□

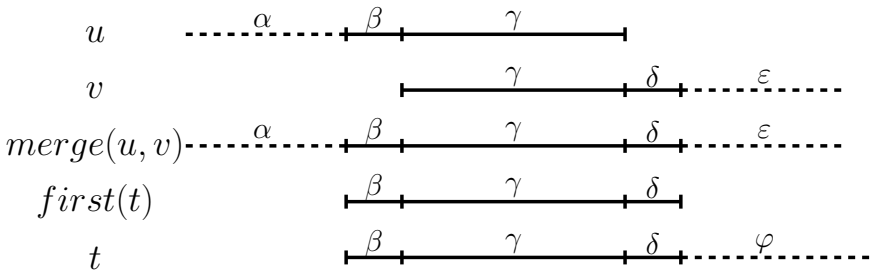


Figure 2.6: Setting in Case 3 of the proof of Lemma 2.17. The Greek labels represent equal substrings. Solid lines represent non-empty strings and dashed lines represent strings that may be empty.

Corollary 2.18. *Let s, t be distinct strings in $S_i \in \mathcal{S}$. Then, s is not a superstring of t.*

Proof. For t to be a substring of s, in particular $first(t)$ would have to be a substring of s. By Lemma 2.17, that is not the case. □

Lemma 2.19 shows that to find the first choice of a string at an arbitrary iteration S_i of LG, it suffices to look at the overlap between input strings.

Lemma 2.19. *Let s, t be distinct strings in $S_i \in \mathcal{S}$. Then,*

$$o(s,t) = o(last(s), first(t))$$
$$o'(s,t) = o'(last(s), first(t))$$
$$o^*(s,t) = \max(o'(last(s), first(t)), o'(last(t), first(s))$$

Proof. We prove the first equation. The other two follow immediately from it and from the corresponding definitions.

By Lemma 2.17, s is not a superstring of $first(t)$, so the maximum overlap between a suffix of s and a prefix of t is a strict substring of $first(t)$. Similarly, t is not a superstring of $last(s)$, so the maximum overlap between a suffix of s and a prefix of t is a strict substring of $last(s)$. Therefore, $o(s,t) = o(last(s), first(t))$. □

Lemma 2.20 shows that when a locally-dominant pair of strings u, v is merged, the new string $merge(u,v)$ does not overlap with the other strings more than either of u or v. It is analogous to reducibility for agglomerative hierarchical clustering and to Lemma 2.12 for the multi-fragment algorithm for TSP.

Lemma 2.20 (Reducibility for shortest common superstring)**.** *Let u, v, s be distinct strings in $S_i \in \mathcal{S}$ such that u and v are locally dominant. Then, $o^*(merge(u,v), s) \leq \max(o^*(u,s), o^*(v,s))$*

Proof. For convenience, let $w = merge(u,v)$. Assume, without loss of generality, that u appears before v in w. By Lemma 2.19,

$$o'(w,s) = o'(last(w), first(s)) = o'(last(v), first(s)) = o'(v,s),$$

$$o'(s,w) = o'(last(s), first(w)) = o'(last(s), first(u)) = o'(s,u).$$

Thus,

$$o^*(w,s) = \max(o'(w,s), o'(s,w)) =$$
$$\max(o'(v,s), o'(s,u)) \leq \max(o^*(v,s), o^*(s,u)).$$

□

Lemma 2.21. *Any pair of strings u, v that is or becomes locally dominant during the execution of local greedy remains locally dominant until they are merged together.*

Proof. Suppose some other pair, s and t, is merged when u and v are locally dominant. We show that after s and t are merged, u and v are still locally dominant. Let $r = merge(s,t)$. By Lemma 2.20,

$$\max(o^*(u,r), o^*(v,r)) \leq \max(o^*(u,s), o^*(u,t), o^*(v,s), o^*(v,t)). \tag{2.1}$$

By the definition of locally dominant,

$$max(o^*(u,s), o^*(u,t), o^*(v,s), o^*(v,t)) < o^*(u,v). \tag{2.2}$$

Combining inequalities (1) and (2) yields $\max(o^*(u,r), o^*(v,r)) < o^*(u,v)$, i.e., r is neither the first choice of u nor the first choice of v. Since r is the only new string after merging s and t, u and v are still locally dominant after the merge. □

Finally, we apply the hybrid method to prove GLE.

Theorem 2.22 (Global-local equivalence in shortest common superstring)**.** *Let S_0 be a set of strings such that no string is a substring of another. If global greedy and local greedy break ties according to the defined tie-breaking rule, they output the same solution.*

Proof. We use the hybrid method. Let $L = l_1, l_2, \ldots, l_{m-1}$ be the sequence of pairs of strings merged by a *specific* run of local greedy, and $S_0, \ldots, S_{m-1}$ the corresponding sequence of string sets. Consider a run of local greedy, denoted $hybrid_i$, for some $i \in \{1, \ldots, m\}$: $hybrid_i$ starts merging the pairs in L, in order, up to and including l_{i-1}. Then, it switches to merging globally-dominant pairs.

Let $i \in \{1, \ldots, m-1\}$. We show that the hybrid run that switches before iteration i ($hybrid_i$) finds the same solution as the hybrid run that switches after iteration i ($hybrid_{i+1}$). Using this result, by induction on i we get that $hybrid_1$ and $hybrid_m$ output the same solution. Since $hybrid_1$ corresponds to GG and $hybrid_m$ outputs L, the theorem follows.

We label the pairs merged by $hybrid_i$ starting at iteration i with $g_i, g_{i+1}, \ldots$. We label the pairs merged by $hybrid_{i+1}$ starting at iteration $i+1$ with $g'_{i+1}, g'_{i+2}, \ldots$ (analogously to Figure 2.2 for the proof of GLE for COPs).

First, note that $hybrid_i$ eventually merges l_i. That is, $l_i = g_j$ for some $j \geq i$. This follows from Lemma 2.21.

Further, $g'_{k+1} = g_k$ for all $k \in \{i, \ldots, j-1\}$. We show that $g'_{i+1} = g_i$. The same reasoning applies to subsequent pairs. The strings in g_i are the globally-dominant pair in S_{i-1}. To show the equivalence, we need to show that they are also globally-dominant in S_i, which is $(S_{i-1} \setminus l_i) \cup \{merge(l_i)\}$. By Lemma 2.20, no string in S_i overlaps more with $merge(l_i)$ than they did with one of the strings in l_i. Thus, since g_i is globally dominant in S_{i-1}, it is also in S_i.

After iteration j, both hybrid runs have merged exactly the same pairs, so they have the same partial solutions, and both have switches to merging globally-dominant pairs. Thus, from that point on, they coincide in their merges, and finish with the same solution. □

Corollary 2.23. *Every possible run of local greedy for shortest common superstring outputs the same solution.*

2.4 Conclusions

The examples in this chapter suggest that global–local equivalence is common in greedy algorithms for optimization problems, and that the hybrid method is a useful technique for proving GLE for greedy algorithms that have it.

The key property for GLE is that locally-dominant choices remain locally dominant when other locally-dominant choices are made (see Lemma 1.3 for agglomerative hierarchical clustering, Lemma 2.8 for COPs, Lemma 2.13 for multi-fragment TSP, and Lemma 2.21 for SCS). With this property, applying the hybrid method should be straightforward. The applications of the hybrid method in this chapter can serve as a footprint for future uses.

Global–local equivalence hinges on the assumption that there are no ties in the evaluations of the choices. If we wish to remove this assumption, we are forced to integrate a tie-breaking scheme that both GG and LG must use congruently. Perhaps it would be more general and elegant to state global–local equivalence differently, and say that every run of local greedy outputs the same solution as *some* run of global greedy. Then, we would not need to assume any specific tie-breaking scheme for GG and LG. Effectively, instead of handling the issue of ties at the algorithmic level, we would confine it to the analysis. This is left as future work. The main obstacle is how to define locally-dominant choices if we allow ties. For instance, if we allow two neighbors in the interaction graph to be both locally-dominant, then we need to reformulate the key property that locally-dominant choices remain locally dominant when other locally-dominant choices are made.

Chapter 3

Nearest-Neighbor Chain Algorithms

3.1 Preliminaries: nearest-neighbor data structures

In this chapter, we propose algorithms for several problems, all of which are inspired by the nearest-neighbor chain (NNC) algorithm from hierarchical clustering (Algorithm 1). As mentioned in Section 1.4.2, the cost of a nearest-neighbor computation is central to the analysis of NNC algorithms. In this section, we review nearest-neighbor and related data structures that we use in our algorithms. In subsequent sections, we refer back to these results as needed.

Knuth's discussion of the classic post office problem [133] started a long line of research on nearest-neighbor data structures. The goal is to maintain a collection of points, called *sites*, and answer queries asking for the closest site to a given query point. Data structures based on *Voronoi diagrams* [16, 58] are efficient when the set of sites is fixed. Given a set of sites, the Voronoi diagram partitions a space into regions such that the points in each region have a particular site as their nearest. However, for NNC-based algorithms, we need dynamic data structures, i.e., that allow sites to be inserted and removed.

For exact two-dimensional *dynamic nearest neighbors*, a data structure with $O(n^{\varepsilon})$ update and query time, for any $\varepsilon > 0$, was given by Agarwal et al. [1], improved to $O(\log^6 n)$ by Chan [45], to $O(\log^5 n)$ by Kaplan et al. [123], and, most recently, to $O(\log^4 n)$ by Chan [46]. Because of the high complexities of these methods, researchers have also looked at finding approximate nearest neighbors in dynamic point sets [85, 171].

We summarize the fastest known runtimes for several dynamic data structures related to finding nearest neighbors. In the following definitions, n is the total number of sites maintained by the data structures.

Definition 3.1 (Dynamic nearest-neighbor data structure)**.** Maintain a set of points, P, subject to insertions, deletions, and nearest-neighbor queries: given a query point q, return the point $p \in P$ closest to q.

Lemma 3.2 ([46])**.** *In $\mathbb{R}^2$ and for Euclidean distance, there is a dynamic nearest-neighbor data structure with $O(n \log n)$ preprocessing time, $O(\log^2 n)$ query time, $O(\log^2 n)$ amortized[7] insertion time, and $O(\log^4 n)$ amortized deletion time.*

Definition 3.3 (Dynamic ε-approximate k nearest-neighbor (k-ANN) data structure)**.** Maintain a set of points, P, subject to insertions, deletions, and *ε-approximate k nearest-neighbor queries*: given a query point q and an integer k with $1 \leq k \leq |P|$, return k points $p_1, \ldots, p_k \in P$ such that, for each p_i, $d(q, p_i) \leq (1 + \varepsilon) d(q, p_i^*)$, where p_i^* is the i-th nearest neighbor of q in P, and $\varepsilon > 0$ is a constant known at the time the data structure is initialized[8].

Lemma 3.4 ([15])**.** *In any fixed dimension, for any L_p metric, and for any $\varepsilon > 0$, there is a dynamic ε-approximate k nearest-neighbor data structure with $O(n \log n)$ preprocessing time, $O(k \log n)$ query time, and $O(\log n)$ insertion and deletion time.*

[7] An amortized time of $O(f(n))$ means that the average time per operation in any sequence of operations is $O(f(n))$, where n is the maximum size of the data structure throughout the sequence. See [178] for additional background.

[8] Some ε-approximate k nearest-neighbor data structures (e.g., [15]) do not need to know ε at construction time, and, in fact, allow ε to be part of the query and to be different for each query. Clearly, such data structures also qualify under the given definition.

Data structure	Metric space	Query	Insertion	Deletion	Reference
Exact NN	$(\mathbb{R}^2, L_2)$	$O(\log^2 n)$	$O(\log^2 n)^\dagger$	$O(\log^4 n)^\dagger$	[46]
k-ANN	$(\mathbb{R}^\delta, L_p)$	$O(\log n)$	$O(\log n)$	$O(k \log n)$	[15]
Closest pair	$(\mathbb{R}^\delta, L_p)$	$O(1)$	$O(\log n)$	$O(\log n)$	[25]
Bichromatic CP	$(\mathbb{R}^2, L_2)$	$O(1)$	$O(\log^2 n)^\dagger$	$O(\log^4 n)^\dagger$	[46]
Soft NN	$(\mathbb{R}^\delta, L_p)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	Section 3.2

Table 3.1: Summary of dynamic nearest-neighbor and related data structures. All the data structures have $O(n \log n)$ preprocessing time. The dimension δ is arbitrary but constant, and the metric L_p is for any $p \geq 1$. Chan [46] does not specify the metric, so we assume L_2. The "†" superindex indicates that the runtime is amortized.

Definition 3.5 (Closest-pair data structure)**.** Maintain a set of points, P, subject to insertions, deletions, and queries asking for the closest pair in P.

Lemma 3.6 ([25])**.** *In any fixed dimension and for any L_p metric, there is a closest-pair data structure with $O(n \log n)$ preprocessing time, $O(1)$ query time, and $O(\log n)$ insertion and deletion time.*

Definition 3.7 (Bichromatic closest-pair data structure)**.** Maintain two sets of points, P and Q, subject to insertions, deletions, and queries asking for the closest pair of points in different sets (one in P and one in Q).

Lemma 3.8 ([46])**.** *In $\mathbb{R}^2$ and for Euclidean distance, there is a dynamic bichromatic closest-pair data structure with $O(n \log n)$ preprocessing time, $O(1)$ query time, $O(\log^2 n)$ amortized insertion time, and $O(\log^4 n)$ amortized deletion time.*

See Table 3.1 for a summary of these results. The table also lists our new *soft nearest-neighbor data structure*, which we introduce in Section 3.2.

Clearly, proximity data structure have received a lot of attention in computational geometry. However, we also consider problems which deal with nodes in a graph instead of points in space. Shortest-path distances in an undirected graph establish a metric, so we can define the analogous of a dynamic nearest-neighbor data structure in this setting. In Chapter 5, we define proximity data structures in graphs (Definitions 5.1, 5.2, and 5.3) and design the first data structures of this kind

(to our knowledge). Table 5.2 is analogous to Table 3.1, but for graph-based data structures (refer to Chapter 5 for the relevant background).

3.2 Soft nearest-neighbor data structure

In this section, we present a new geometric data structure that we use in some of our NNC-inspired algorithms (in Sections 3.3 and 3.8).

Throughout this section, we consider points in $\mathbb{R}^\delta$, for some fixed dimension δ, and distances, denoted $d(\cdot, \cdot)$, measured under any L_p metric. Our data structure (Definition 3.9) is a variation of the traditional dynamic nearest-neighbor data structure (Definition 3.1).

Definition 3.9 (Dynamic soft nearest-neighbor data structure)**.** Maintain a set of points, P, subject to insertions, deletions, and *soft nearest-neighbor* queries: given a query point q, return either of the following:

- The nearest neighbor of q in P: $p^* = \arg\min_{p \in P} d(q, p)$.
- A pair of points p, p' in P satisfying $d(p, p') < d(q, p^*)$.

This section is dedicated to proving Theorem 3.10, which captures the main result (Theorem 3.10 is a consequence of Lemma 3.14).

Theorem 3.10. *In any fixed dimension, and for any L_p metric, there is a dynamic soft nearest-neighbor data structure with $O(n \log n)$ preprocessing time and $O(\log n)$ time per query, insertion, and deletion.*

We label the two types of answers to soft nearest-neighbor (SNN) queries as *hard* and *soft*. A "standard" NN data structure (Definition 3.1) is a special case of a SNN structure that always gives

hard answers. Comparing Lemma 3.2 and Theorem 3.10, we can see that allowing soft answers enables us to speed up the data structure and generalize it to any fixed dimension.

3.2.1 Implementation

Let P be the point set to be maintained in a soft nearest-neighbor data structure. We maintain the point set P in a dynamic k-ANN data structure (Definition 3.3) initialized with an approximation factor ε that will be determined later. The chosen ε will be a constant that depends only on the metric space. In this section, we show how to reduce each SNN query to a single query to the k-ANN data structure maintaining P. The query will have a constant k. Once we show this reduction, we will have proved that our data structure achieves the same runtime per operation as the k-ANN structure (Lemma 3.4).

In what follows, q denotes an arbitrary query point, p^* its nearest neighbor in P, and p_i^* the i-th closest point to q in P (in particular, $p_1^* = p^*$). We make a general position assumption: the distances from q to the points in P are all unique. For the k-ANN structure, we assume that the k returned points for a query, $p_1, \ldots, p_k$, are sorted by non-decreasing distance from the query point q. If that is not the case, note that the same set of points but in sorted order are also a valid answer to the same query. With this assumption, only the first returned point, p_1, can be p^*. Queries rely on the following lemma.

Lemma 3.11. *Let $p_1, \ldots, p_k$ be the answer given by a k-ANN structure, initialized with approximation factor ε, to a query (q, k). If $p_1 \neq p^*$, then for each p_i in $p_1, \ldots, p_k$, $d(q, p_i) \leq (1+\varepsilon)^{i-1} d(q, p_1)$.*

Proof. For $i = 1$, the claim is trivial. For $i = 2, \ldots, k$, note that $d(q, p_i^*) \leq d(q, p_{i-1})$. This is because there are at least i points within distance $d(q, p_{i-1})$ of q: $p^*, p_1, \ldots, p_{i-1}$. Thus, $d(q, p_i) \leq (1+\varepsilon) d(q, p_i^*) \leq (1+\varepsilon) d(q, p_{i-1})$. The claim follows by induction on i. □

We will use the contrapositive of Lemma 3.11, namely:

Corollary 3.12. *Let $p_1, \ldots, p_k$ be the answer given by a k-ANN structure, initialized with approximation factor ε, to a query (q, k). If $d(q, p_k) > (1+\varepsilon)^{k-1} d(q, p_1)$, then $p_1 = p^*$.*

In the following definition, we use the following terms. A closed *shell* with inner radius r_1 and outer radius r_2 is the set of points p satisfying $r_1 \leq p \leq r_2$. A region in space is *closed* if the boundary points are inside the region. Otherwise, the region is *open*.

Definition 3.13 (Valid SNN parameters)**.** We call a pair (ε, k) valid SNN parameters (for a given metric space) if every set of k points inside a closed shell with inner radius 1 and outer radius $(1+\varepsilon)^{k-1}$ contains two points, p and p', satisfying $d(p, p') < 1/(1+\varepsilon)$.

Suppose that (ε^*, k^*) are valid parameters. Initially, we construct the k-ANN structure using ε^* for the approximation factor. Then we answer queries as in Algorithm 2.

Algorithm 2 Soft nearest-neighbor query.

```
Ask query (q, k*) to the k-ANN structure maintaining P and initialized with ε*.
Measure the distance between each pair of the k* returned points, p_1, ..., p_{k*}.
if any pair (p_i, p_j) satisfies d(p_i, p_j) < d(q, p_1)/(1 + ε*) then
    return p_i, p_j.
else
    return p_1.
```

Lemma 3.14. *If (ε^*, k^*) are valid SNN parameters, Algorithm 2 is correct.*

Proof. The algorithm considers two cases. First, if a pair p_i, p_j of points returned by the k-ANN structure satisfies $d(p_i, p_j) < d(q, p_1)/(1+\varepsilon^*)$, p_i and p_j are a valid soft answer to the query, as $d(q, p_1)/(1+\varepsilon^*) \leq d(q, p^*)$.

In the alternative case, no pair among the returned points is at distance $< d(q, p_1)/(1+\varepsilon^*)$. Consider the closed shell centered at q, with inner radius $d(q, p_1)$, and outer radius $(1+\varepsilon^*)^{k^*-1} d(q, p_1)$. If we scale distances so that $d(q, p_1) = 1$, then this shell has inner radius 1 and outer radius

$(1+\varepsilon^*)^{k^*-1}$. Given that (ε^*, k^*) are valid SNN parameters, if all k^* of the returned points were inside this shell, at least two of them would be at a distance smaller than $1/(1+\varepsilon^*)$. However, without the scaling, this distance would be smaller than $d(q,p_1)/(1+\varepsilon^*)$, which corresponds to the case that we considered first. Thus, at least one of the returned points lies outside the shell. By Corollary 3.12, $p^* = p_1$. □

3.2.2 Choice of parameters

We left open the issue of finding valid SNN parameters for a metric space (Definition 3.13). Recall that (ε, k) are valid if every set of k points inside a shell with inner radius 1 and outer radius $(1+\varepsilon)^{k-1}$ contains two points, p and p', satisfying $d(p,p') < 1/(1+\varepsilon)$. To simplify the question, we can scale distances by $(1+\varepsilon)$, so that the inner radius is $1+\varepsilon$, the outer radius $(1+\varepsilon)^k$, and the required distance between the two points is < 1. As a further simplification, we can shrink the inner radius back to 1 (without scaling anything else). This makes the shell grow, and thus, if (ε, k) are valid parameters with this change, then they are also valid under the original statement. Hence, to clarify, the goal of this section is to show how to find, for any metric space $(\mathbb{R}^\delta, L_p)$, a pair of parameters (ε, k) such that any set of k points inside a shell with inner radius 1 and outer radius $(1+\varepsilon)^k$ contains two points, p and p', satisfying $d(p,p') < 1$.

This question is related to the *kissing number* of the metric space [148], which is the maximum number of points that can be on the surface of a unit sphere all at pairwise distance ≥ 1. For instance, it is well known that the kissing number is 6 in $(\mathbb{R}^2, L_2)$ and 12 in $(\mathbb{R}^3, L_2)$. It follows that, in $(\mathbb{R}^2, L_2)$, $(\varepsilon^* = 0, k^* = 7)$ are valid SNN parameters. Of course, we are interested in $\varepsilon^* > 0$, so that we can use a k-ANN structure. Thus, our question is more general than the kissing number in the sense that our points are not constrained to lie on a ball, but in a shell (and, to complicate things, the outer radius of the shell, $(1+\varepsilon)^k$, depends on the number of points).

Lemma 3.15. *There are valid SNN parameters in any metric space* $(\mathbb{R}^\delta, L_p)$.

Proof. Consider a closed shell with inner radius 1 and outer radius 2 (for the purposes of the proof, the number 2 here is arbitrary, and could be any value ≥ 1.5). A set of points in the shell at pairwise distance ≥ 1 corresponds to a set of disjoint open balls of radius $1/2$ with the center inside the shell. Consider the volume of the intersection of the shell with such a ball. This volume is bounded below by some constant, v, corresponding to the case where the ball is centered along the exterior boundary. Since the volume of the shell, v_s, is itself constant, the maximum number of disjoint balls of radius $1/2$ that fit in the shell is a constant smaller than v_s/v. This is because no matter where the balls are placed, at least v volume of the shell is inside any one of them, so, if there are more than v_s/v balls, there must be some region in the shell inside at least two of them. This corresponds to two points at distance < 1.

Set k to be $\lceil v_s/v \rceil$, and ε to be the constant such that $(1+\varepsilon)^k = 2$. Then, (ε, k) are valid parameters for $(\mathbb{R}^\delta, L_p)$. □

The dependency of k-ANN structures on $1/\varepsilon$ is typically severe. Thus, for practical purposes, one would like to find a valid pair of parameters with ε as big as possible. The dependency on k is usually negligible in comparison, and, in any case, k cannot be too large because the shell's width grows exponentially in k. Thus, we narrow the question to optimizing ε: what is the largest ε that is part of a pair of valid parameters?

We first address the case of $(\mathbb{R}^2, L_2)$, where we derive the optimal value for ε analytically. We then give a heuristic, numerical algorithm for general $(\mathbb{R}^\delta, L_p)$ spaces.

Parameters in $(\mathbb{R}^2, L_2)$. Let $\varepsilon_\varphi \approx 0.0492$ be the number such that $(1 + \varepsilon_\varphi)^{10} = \varphi$, where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. The valid SNN parameters with largest ε for $(\mathbb{R}^2, L_2)$ are $(\varepsilon^* < \varepsilon_\varphi, k^* = 10)$ (ε^* can be arbitrarily close to ε_φ, but must be smaller). This follows from the following observations.

- The kissing number is 6, so there are no valid parameters with $k < 6$.

- The thinnest annulus (i.e., 2D shell) with inner radius 1 such that 10 points can be placed inside at pairwise distance ≥ 1 has outer radius $\varphi = (1 + \varepsilon_\varphi)^{10}$. Figure 3.1, top, illustrates this fact. In other words, if the outer radius is any smaller than φ, two of the 10 points would be at distance < 1. Thus, any valid pair with $k = 10$ requires ε to be smaller than ε_φ, but any value smaller than ε_φ forms a valid pair with $k = 10$.

- For $6 \leq k < 10$ and for $k > 10$, it is possible to place k points at pairwise distance > 1 in an annulus of inner radius 1 and outer radius $(1 + \varepsilon_\varphi)^k$, and they are not packed "tightly", in the sense that k points at pairwise distance > 1 can lie in a thinner annulus. This can be observed easily; Figure 3.1 (bottom) shows the cases for $k = 9$ and $k = 11$. Cases with $k < 9$ can be checked one by one; in cases with $k > 11$, the annulus grows at an increasingly faster rate, so placing k points at pairwise distance > 1 of each other becomes increasingly "easier". Thus, for any $k \neq 10$, any valid pair with that specific k would require an ε smaller than ε_φ.

Parameters in $(\mathbb{R}^\delta, L_p)$. For other $(\mathbb{R}^\delta, L_p)$ spaces, we suggest a numerical approach. We can do a binary search on the values of ε to find one close to optimal. For a given value of ε, we want to know if there is any k such that (ε, k) are valid. We can search for such a k iteratively, trying $k = 1, 2, \ldots$ (the answer will certainly be "no" for any k smaller than the kissing number). Note that, for a fixed k, the shell with inner radius 1 and outer radius $(1 + \varepsilon)^k$ has constant volume. As in Lemma 3.15, let v be the volume of the intersection between the shell and a ball of radius $1/2$ centered on the exterior boundary of the shell. As argued before, if kv is bigger than the shell's volume, then (ε, k) are valid parameters. For the termination condition, note that if in the iterative search for k, k reaches a value where the volume of the shell grows more than v in a single iteration, no valid value of k will be found for that ε, as the shell grows faster than the new points cover it.

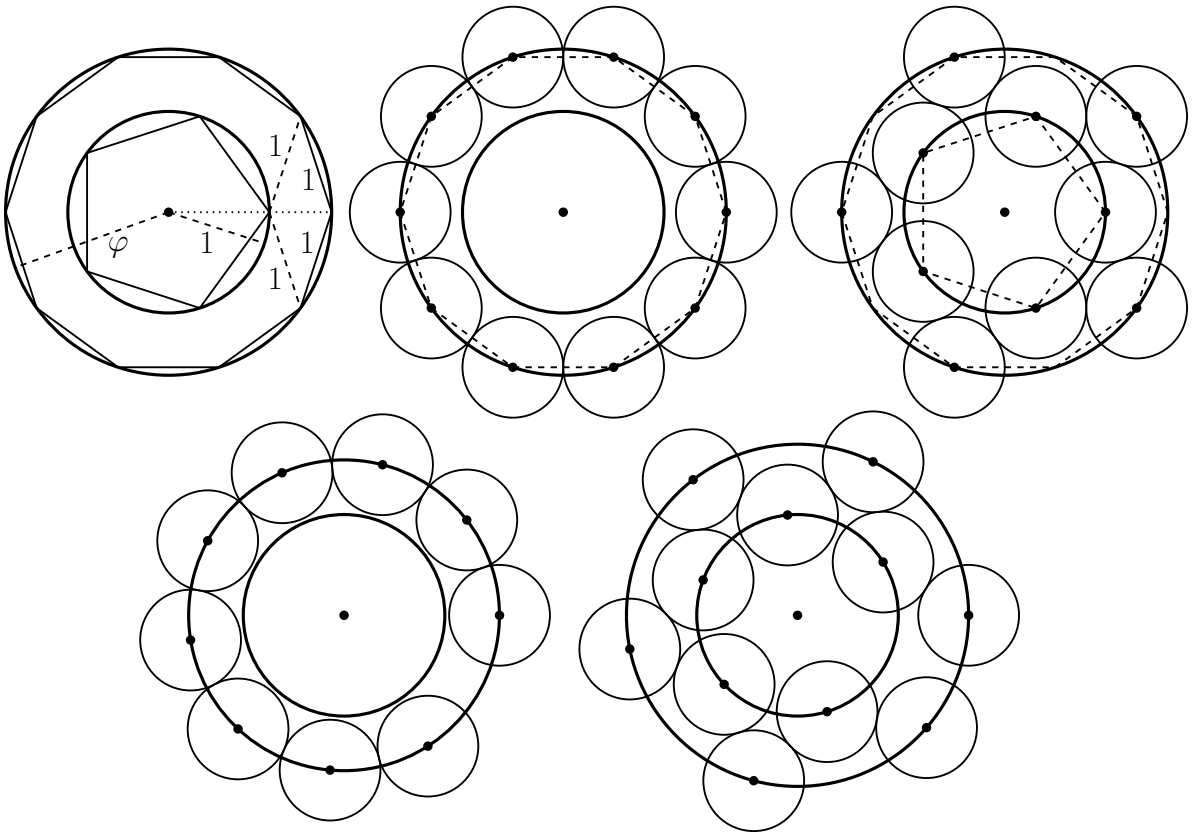


Figure 3.1: **Top:** The first figure shows two concentric circles of radius 1 and φ with an inscribed pentagon and decagon, respectively, and some proportions of these shapes. The other figures show two different ways to place 10 points at pairwise distance ≥ 1 inside an annulus of inner radius 1 and outer radius $(1+\varepsilon_\varphi)^{10} = \varphi$. Disks of radius $1/2$ around each point are shown to be non-overlapping. In one case, the points are placed on the vertices of the decagon. In the other, they alternate between vertices of the decagon and the pentagon. In both cases, the distance between adjacent disks is 0. Thus, these packings are "tight", i.e., if the annulus were any thinner, there would be two of the 10 points at distance < 1. **Bottom:** 9 and 11 points at pairwise distance ≥ 1 inside annuli of radius $(1+\varepsilon_\varphi)^9$ and $(1+\varepsilon_\varphi)^{11}$, respectively. These packings are not tight, meaning that, for $k = 9$ and $k = 11$, a valid value of ε would have to be smaller than ε_φ.

Besides the volume check, one should also consider a lower bound on how much of the shell's surface (both inner and outer) is contained inside an arbitrary ball. We can then see if, for a given k, the amount of surface contained inside the k balls is bigger than the total surface of the shell, at which point two balls surely intersect. This check finds better valid parameters than the volume one for relatively thin shells, where the balls "poke" out of the shell on both sides.

3.2.3 The m-way soft nearest-neighbor data structure

We can modify the specification of the SNN structure so that soft answers return m pairwise closer points instead of two. That is, a soft answer should return m points where each pair is closer to each other than the query point to its NN. We call this an *m-way* SNN structure. The normal SNN structure is a 2-way SNN structure.

Lemma 3.16. *In any fixed dimension, for any L_p metric, and for any constant $m \geq 2$, there is an m-way SNN structure with $O(n \log n)$ preprocessing time and $O(\log n)$ time per query, insertion, or deletion.*

Proof. To obtain an m-way SNN structure, we need to change the values of ε and k to make the shell smaller and k bigger, so that if there are k points in a shell of inner radius 1 and outer radius $(1+\varepsilon)^k$, then there must be at least m points at pairwise distance less than 1. The method described in Section 3.2.2 for finding valid parameters in (R^δ, L_p) also works here. It only needs to be modified so that the area (or surface) of the shell is accounted for over $m-1$ times, so that at some point m balls must intersect. Since k and ε are still constant, this does not affect the asymptotic runtimes in Theorem 3.10. □

3.2.4 The closest pair problem

To conclude this section, we give an illustrative application of the SNN structure, the *closest pair problem* [22, 173].

Problem 6 (Closest pair problem)**.** Given a set of points in a metric space, find the closest pair of points.

This problem can be solved in $O(n \log n)$ time in any metric space $(\mathbb{R}^\delta, L_p)$ with constant dimension [57]. For instance, it can be solved in this time bound using a dynamic closest-pair data structure (Lemma 3.6). With our SNN structure, we can match this result.

If a SNN structure is available, finding the closest pair in a point set P is simple: add all the points in P to the structure and do a query with each point in P (for each point, remove it temporarily from the structure to do the query, so that the answer is not itself). Since, there is no *closer* pair than the closest pair, the SNN structure will return a hard answer for the two points constituting the closest pair. Thus, the closest pair will be found among the hard answers to the queries.

This result sets a lower bound on the runtime of a SNN structure in the algebraic decision tree model of computation: it is well known that the closest pair problem requires $\Omega(n \log n)$ time in this model, shown by a reduction from the element-uniqueness problem [172]. Therefore, for a SNN structure in this model, either the preprocessing must take $\Omega(n \log n)$ time, or an insertion must take $\Omega(\log n)$ time, or doing a query must take $\Omega(\log n)$ time.

3.3 Geometric TSP

In Section 2.2, we introduced TSP (Problem 4) and the multi-fragment greedy algorithm. We then defined the corresponding local greedy algorithm (Definition 2.11) and showed that both output the same solution (Theorem 2.14). Recall that both algorithms work by merging paths, starting with paths that are just nodes, until there is a single path left. The global greedy algorithm merges the paths with the smallest cost, whereas local greedy merges any pair of paths such that the cost of connecting them is lower than the cost of connecting either with a third path. Here, the cost of connecting two paths is the minimum cost of connecting an endpoint of each.

In this section, we use a NNC-inspired algorithm to implement the local greedy algorithm. We are interested in a geometric setting (Problem 7). Since it is a special case of TSP, global–local equivalence holds.

Problem 7 (Geometric TSP)**.** Given a set of points in a $(\mathbb{R}^\delta, L_p)$, find a closed tour (a closed polygonal chain) through all the points of shortest length.

3.3.1 Related work

Note the following hierarchy of problems. In its most generic form, TSP has no restriction on the edge weights, and the optimal solution cannot be approximated to within any constant approximation factor (assuming $\mathbf{P} \neq \mathbf{NP}$) [182]. An important special case is *metric TSP*, where distances satisfy the triangle inequality. For this case, Christofides' algorithm achieves a $3/2$-approximation [52]. In turn, Problem 7, which we call geometric TSP, is a special case of metric TSP. A polynomial-time approximation scheme[9] is known for this case [14]. Finally, Euclidean TSP is a special case of Problem 7. In Euclidean TSP, the points lie in the plane and the metric is Euclidean distance. The problem is NP-hard even in the restricted case of Euclidean TSP [108].

The multi-fragment algorithm achieves an approximation ratio of $O(\log n)$ for metric TSP [35, 156]. It was proposed by Bentley [21] specifically for Euclidean TSP. Despite the fact that there are algorithms with better approximation ratios, it is used in practice due to its simplicity and empirical evidence that it tends to outperform other heuristics [23, 77, 120, 147, 149].

We are interested in the complexity of computing the same tour as the multi-fragment algorithm, which we call the multi-fragment tour. A straightforward implementation of the multi-fragment algorithm is similar to Kruskal's minimum spanning tree algorithm [134]: sort the $\binom{n}{2}$ edges by increasing weights and process them in order: for each edge, if the two nodes are endpoints of

[9] A polynomial-time approximation scheme is a parameterized family of algorithms such that, for every fixed $c > 0$, there is a $(1 + c)$-approximation algorithm that runs in time polynomial in the input size.

separate paths, connect them. The runtime of this algorithm is $O(n^2 \log n)$, where the bottleneck is sorting. Eppstein [79] uses a dynamic closest-pair data structure (for arbitrary distance matrices) to improve the runtime to $O(n^2)$ time.

In the geometric setting, we can leverage the additional structure to improve the runtime. Bentley [21] gives a K-d tree-based implementation and says that it appears to run in $O(n \log n)$ time on uniformly distributed points in the plane. Following our local greedy framework, we give an alternative algorithm for geometric TSP that computes the multi-fragment tour in $O(n \log n)$ time in any fixed dimension (Theorem 3.20). We do not know of any prior worst-case subquadratic algorithm.

3.3.2 Soft nearest-neighbor chain algorithm

In this section, we consider TSP in the geometric setting (Problem 7). Given that we have global–local equivalence (Theorem 2.14), we can adapt the NNC algorithm (Algorithm 1) to implement local greedy and compute the multi-fragment tour. The NNC algorithm maintains a chain of paths, each followed by its nearest-neighbor path, until we reach a locally-dominant pair (i.e., a pair of MNN). Then, remove both from the chain and connect them. The distance between paths is the minimum distance between their endpoints. We use $p \cup p'$ to denote the path resulting from connecting paths p and p'. They are connected by adding the line segment between their closest endpoints. To find the nearest neighbor of a path in a set of paths, we need to find the closest endpoint of another path from each endpoint. For this, we can use a nearest-neighbor data structure maintaining the set of path endpoints.

The dynamic NN structure by Chan [46] runs in $O(\log^4 n)$ amortized time per operation (Lemma 3.2). Pairing the NNC algorithm with this structure yields a runtime of $O(n \log^4 n)$ for points in $\mathbb{R}^2$ and Euclidean distance. We skip the proof of this result because we will prove a stronger one.

In order to improve upon the NNC algorithm, we need to relax the invariant that each path in the chain is followed by its nearest neighbor. Despite the name, the NNC algorithm does not rely on the fact that each element is followed by its nearest neighbor in the chain. It only requires that the distances between consecutive elements in the chain (in this case, paths) keeps decreasing. This is enough to converge to a locally-dominant pair of elements. In fact, we do not even need an element in the chain, say, p, to be followed by an element close to p. We can jump to a pair of completely different elements, as long as the distance between them is smaller than the distance between p and the element before p in the chain. This, too, suffices to converge to a locally-dominant pair.

Relaxing the chain in this manner is the central idea behind our algorithm for geometric TSP. We use a variation of the NNC algorithm that uses a SNN structure instead of the usual NN structure. We call it the *soft nearest-neighbor chain algorithm* (SNNC). For this, we need a SNN structure for paths instead of points. That is, a structure that maintains a set of (possibly single-point) paths, and, given a query path Q, returns the closest path to Q or two paths in the set which are closer to each other than Q to its closest path in the set.

A soft nearest-neighbor structure for paths. The distance between paths is measured as the minimum distance between an endpoint of one and an endpoint of the other, so, for the purposes of this data structure, only the coordinates of the endpoints are important.

Given a set of paths to store in the data structure, we maintain the set of path endpoints in a 3-way SNN structure for points (Lemma 3.16). Insertions and removals are straightforward: we add or remove, accordingly, both endpoints of the path.

Algorithm 3 shows the full algorithm for answering SNN queries about paths using a 3-way SNN structure for points. Given a query path Q with endpoints $\{q_1, q_2\}$, we do a SNN query from each endpoint of the path. If both answers are hard (assuming that the path has two distinct endpoints, otherwise, just the one), then we find the true NN of the path, and we can return it. However, there

is a complication with soft answers: the reason why we need three pairwise closer points instead of just two is that two points could be the endpoints of the same path. Thus, it could be the case that we find two closer points, but not two closer paths as we need. Using a 3-way SNN structure guarantees that even if two of the three endpoints belong to the same path, at least two different paths are involved.

Algorithm 3 Soft-nearest-neighbor query for paths.

Let q_1 and q_2 be the endpoints of the query path, Q.
Let S be a 3-way SNN structure containing the set of path endpoints.
Query S with q_1 and q_2.
if both answers are hard **then**
    Let p_1 and p_2 be the respective answers.
    return the closest path to the query path among the paths with endpoints p_1 and p_2.
else if one answer is hard and the other is soft **then**
    Let p be the hard answer to q_1 and (a, b, c) the soft answer to q_2 (without loss of generality).
    Let P and P' be the two closest paths among the paths with endpoints a, b, and c.
    if $d(q_1, p) < d(P, P')$ **then**
        return the path with endpoint p.
    else
        return (P, P').
else (both answers are soft)
    Let (a_1, b_1, c_1) and (a_2, b_2, c_2) be the answers to q_1 and q_2.
    return the closest pair of paths among the paths with endpoints $a_1, b_1, c_1, a_2, b_2, c_2$.

Lemma 3.17. *In any fixed dimension, and for any L_p metric, we can maintain a set of n paths in a SNN structure for paths with $O(n \log n)$ preprocessing time and $O(\log n)$ time per query, insertion, or deletion.*

Proof. All the runtimes follow from Lemma 3.16: we maintain the set of up to $2n$ path endpoints in a three-way SNN structure S. The structure S can be initialized in $O(n \log n)$ time. Insertions and deletions require two insertions or deletions to S, respectively, taking $O(\log n)$ time each. Algorithm 3 for queries clearly runs in $O(\log n)$ time. We argue that it returns a valid answer. Let Q be a query path with endpoints $\{q_1, q_2\}$, and consider the three possible cases:

- Both answers are hard. In this case, we find the closest path to each endpoint, and, by definition, the closest of the two is the NN of Q.

- One answer is soft and the other is hard. Let p be the hard answer to q_1 and (a, b, c) the soft answer to q_2 (without loss of generality). Let P and P' be the two closest paths among the paths with endpoints a, b, and c. If $d(q_1, p) < d(P, P')$, then, the path with p as endpoint must be the NN of Q, because there is no endpoint closer than $d(P, P')$ to q_2. Otherwise, P, P' is a valid soft answer, as they are closer to each other than either endpoint of Q to their closest endpoints.

- Both answers are soft. Assume, without loss of generality, that the NN of Q is closer to q_1 than q_2. Then, the soft answer to q_1 gives us two paths closer to each other than Q to its NN, so we return a valid soft answer. □

Note that we cannot build a closest-pair data structure for paths, which would allow us to implement GG directly, from a closest-pair data structure for points maintaining the set of endpoints of the paths. This is because the closest pair of endpoints could be the endpoints of the same path.

The algorithm. We use a SNN for paths (Lemma 3.17). In the context of this algorithm, let us think of a SNN answer, hard or soft, as being a set of two paths. If the answer is hard, then one of the paths returned in the answer is the query path itself, and the remaining path is its NN. Now, we can establish a comparison relationship between SNN answers (independently of their type): given two SNN answers $\{a, b\}$ and $\{c, d\}$, we say that $\{a, b\}$ is *better* than $\{c, d\}$ if and only if $d(a, b) < d(c, d)$.

See Algorithm 4. The input is a set of points, which are interpreted as single-point paths and added to the SNN structure for paths. We assume unique distances between the points. The algorithm maintains a stack (the chain) of *nodes*, where each node contains a pair of paths. In particular, each node in the chain is the best SNN answer among two queries for the two paths in the predecessor

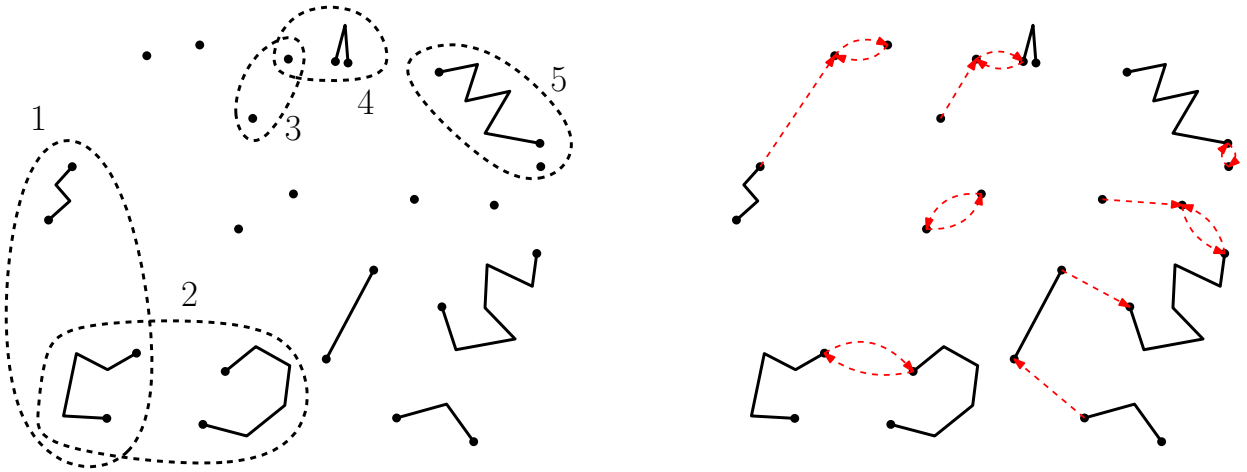


Figure 3.2: **Left:** a set of paths, including some single-point paths, and a possible chain, where the nodes are denoted by dashed lines and appear in the chain according to the numbering. **Right:** nearest-neighbor graph of the set of paths. For each path, a dashed/red arrow points to its NN. The arrows start and end at the endpoints determining the minimum distance between the paths.

node (when querying from a path, we remove it from the structure temporarily, so that the answer is not the path itself).

Figure 3.2 shows a snapshot of the algorithm. Nodes 2 and 4 are hard answers, whereas nodes 3 and 5 are soft answers. The two paths in the fifth node are the overall closest pair, so the SNN structures will return that pair when queried from each of them. The algorithm will connect them, remove the fifth node from the chain, and continue from the fourth node.

Correctness and runtime analysis.

Lemma 3.18. *The following invariants hold at the beginning of each iteration of Algorithm 4:*

1. *Each input point is an endpoint or an internal vertex of exactly one path in S.*
2. *If node R appears after node $\{s,t\}$ in the chain, then R is better than $\{s,t\}$.*
3. *Every path in S appears in at most two nodes in the chain, in which case they consist of consecutive nodes.*

Algorithm 4 Soft nearest-neighbor chain algorithm for geometric TSP.

Initialize an empty stack (the chain).
Initialize a SNN structure S for paths with the set of input points as single-point paths.
while there is more than one path in S **do**
  if the chain is empty **then**
    add a node containing an arbitrary pair of paths from S to it.
  else
    Let $U = \{u, v\}$ be the node at the top of the chain.
    Remove u from S, query S with u, and re-add u to S.
    Remove v from S, query S with v, and re-add v to S.
    Let A be the best answer.
    if $A \neq U$ **then**
      Add A to the chain.
    else
      Remove u and v from S and add $u \cup v$.
      Remove U from the chain.
      if the chain is not empty and the new last node, V, contains u or v **then**
        Remove V from the chain.
Connect the two endpoints of the remaining path in S.

4. *The chain only contains paths in S.*

Proof.

1. The claim holds initially. Each time two paths, u and v, are replaced by $u \cup v$, one endpoint of each becomes an internal vertex in the new path $u \cup v$, and the other endpoints become endpoints of $u \cup v$.

2. We show it for the specific case where R is immediately after $\{s, t\}$ in the chain, which suffices. Note that $R \neq \{s, t\}$, or it would not have been added to the chain. We distinguish between two cases:

   - s and t were MNN when R was added. Then, R had to be a soft answer from s or t, which would have to be better than $\{s, t\}$.

- s and t were not MNN when R was added. Then, s had a closer path than t (without loss of generality). Thus, whether the answer for s was soft or hard, the answer had to be better than $\{s, t\}$.

3. Assume, for a contradiction, that a path p appears in two non-consecutive nodes, $X = \{p, x\}$ and $Z = \{p, z\}$ (this covers the case where p appears more than twice). Let Y be the successor of X. By Invariant 2, Z is better than Y. It is easy to see that if z_1 and z_2 are the two endpoints of path z, then z_1 and z_2 were endpoints of paths since the beginning of the algorithm. Thus, the answer for p when X was at the top of the chain had to be a pair at distance at most $\min(d(p, z_1), d(p, z_2))$. However, $\min(d(p, z_1), d(p, z_2)) = d(p, z)$, contradicting that Z is better than Y.

4. Clearly, each node in the chain contains paths that were present in S at the time the node was added. Therefore, the invariant could only break when removing paths from S. In the algorithm, paths are removed from S when merging the paths in the top node. Thus, if a path p is removed from S, it means that p is in the top node. By Invariant 3, besides the top node, p can only occur in the second-from-top node. In the algorithm, when we merge the paths in the top node, we remove the top node from the chain, as well as its predecessor if has a path in common with the top node. □

Lemma 3.19. *Paths connected in the soft nearest-neighbor chain algorithm (Algorithm 4) are MNN in the set of paths in the SNN structure.*

Proof. Let $\{u, v\}$ be the node at the top of the chain at some iteration of the SNNC algorithm. Let A the best SNN answer among the queries from u and v. In the algorithm, u and v are connected when $A = \{u, v\}$. Thus, we need to show that if $A = \{u, v\}$, then u and v are MNN. We show the contrapositive: if u and v are *not* MNN, then $A \neq \{u, v\}$. If u and v are not MNN, u (without loss of generality) has a closer path than v, so the answer for u is *better* than $\{u, v\}$. □

Theorem 3.20. *The multi-fragment tour of a set of n points in any fixed dimension, and under any L_p metric, can be computed in $O(n \log n)$ time.*

Proof. We show that the SNNC algorithm computes the multi-fragment tour in $O(n \log n)$ time.

For its correctness, note that the output is a single tour that visits every input point (Invariant 1). This cycle is constructed by only merging pairs of paths that are MNN (Lemma 3.19). Thus, the SNNC algorithm implements local greedy. By global-local equivalence (Theorem 2.14), this produces the multi-fragment tour.

For the runtime, note that the chain is acyclic in the sense that each node contains a path from the current set of paths in S (Invariant 4) not found in previous nodes (Invariant 3). Thus, the chain cannot grow indefinitely, so, eventually, paths get connected. The main loop does not halt until there is a single path.

If there are n points at the beginning, there are $n - 1$ connections between different paths in total, and $2n - 1$ different paths throughout the algorithm. This is because each connection removes two paths and adds one new path. At each iteration, either two paths are connected or one node is added to the chain. There are $n - 1$ iterations of the first kind, each of which triggers the removal of one or two nodes in the chain. Thus, the total number of nodes removed from the chain is at most $2n - 2$. Since every node added is removed, the number of nodes added to the chain is also bounded by $2n-2$. Thus, the total number of iterations of the second kind is at most $2n-2$, and the total number of iterations is at most $3n-3$. Therefore, the total running time is $O(P(n)+nT(n))$, where $P(n)$ and $T(n)$ are the preprocessing and operation time of a SNN structure for paths. By Lemma 3.17, this can be done in $O(n \log n)$ time. □

3.3.3 Nearest-neighbor chain algorithm for general graphs

Consider that we are not in the geometric setting, so we do not have access to nearest-neighbor data structures. Recall from Section 3.3.1 that the best prior known runtime for computing the multi-fragment tour in this setting is $O(n^2)$ time, where n is the number of nodes [79]. If the non-input space is required to be $O(n)$, then the runtime increases to $O(n^2 \log^2 n)$ [79].

We can design a NNC algorithm that runs in $O(n^2)$ time, uses $O(n)$ extra space, and simplifies the required data structures. This is because Eppstein [79] did not exploit global–local equivalence, so it had to find closest pairs [79]. Using GLE makes the problem easier.

We use the traditional NNC algorithm. We only need to spell out how to find the nearest neighbor of a path P. Clearly, it can be found in $T(n) = O(n)$ time by scanning through the adjacency lists of the two endpoints of P, filtering non-endpoint nodes and the other endpoint of P. Using this linear search, we can easily compute the multi-fragment tour in $O(nT(n)) = O(n^2)$ time and $O(n)$ extra space.

3.4 Steiner TSP

Consider the Steiner TSP problem [62].

Problem 8 (Steiner TSP)**.** Given a weighted, undirected graph $G = (V, E)$ and a set of k nodes, $P \subseteq V$, called *sites*, find a minimum-weight tour (repeated vertices and edges allowed) in G that visits every site in P at least once. Nodes not in P do not need to be visited.

For instance, G could represent a road network, and the sites could represent the daily drop-off locations of a delivery truck. We consider the multi-fragment algorithm for this problem. An instance of Steiner TSP is equivalent to a TSP instance where the nodes are the sites, and the edge weights are shortest-path distances.

As mentioned, one way to implement the multi-fragment algorithm is to sort the $\binom{k}{2}$ pairs of sites by increasing distances, and process them in order: for each pair, if the two sites are endpoints of separate paths, connect them. In this setting, the bottleneck is computing the distances. By running Dijkstra's algorithm [70] from each site in a sparse graph, this takes $O(kn \log n)$. This can be improved to $O(kn)$ for planar graphs or, more generally, graphs belonging to hereditary graph classes with sublinear separators and for which a certain subdivision can be constructed in $O(kn)$ time [110] (see Chapter 5 for the relevant definitions). We do not know of any prior faster algorithm to compute the multi-fragment tour for Steiner TSP.

The proof of global–local equivalence for TSP (Theorem 2.14) also applies in this setting as long as the distances between sites are unique. Thus, we can use the NNC algorithm to construct the multi-fragment tour in $O(P(n,k) + kT(n,k))$ time, where $P(n,k)$ and $T(n,k)$ are the preprocessing and operation time of a dynamic nearest-neighbor structure maintaining k sites in an n-node graph. We omit the details of the NNC algorithm, as it is does not have any interesting modification over the standard NNC algorithm that we already discussed for TSP.

We use our NN data structure for graphs from Chapter 5. According to Theorem 5.4, using this data structure we get the following.

Theorem 3.21. *Let c be a constant with $0 < c < 1$, and let $\mathcal{G}$ be a hereditary graph class with $O(n^c)$-size separators which can be found in $O(n^{1+c})$ time. For an n-node graph from $\mathcal{G}$, the multi-fragment tour for the Steiner TSP problem can be computed in $O(n^{1+c} + kn^c \log k)$ time, where k is the number of sites.*

For instance, for planar graphs and graphs representing real-world road networks [87], we can construct the multi-fragment tour in $O(n^{1.5} + kn^{0.5} \log k)$ time. As a function of n alone, this is an improvement from $O(n^2)$ to $O(n^{1.5} \log n)$. In addition, in trees and graphs of bounded treewidth [167], which have separators of size $O(1)$ [55], our data structure from Chapter 5 has

$O(n \log n)$ preprocessing time and $O(\log n \log k)$ operation time, so we can construct a multi-fragment tour in $O(n \log n + k \log n \log k)$ time.

The Steiner TSP problem illustrates that the NNC algorithm is not only useful in a geometric setting. Rather, NNC is efficient in any setting where we can find nearest neighbors efficiently.

3.5 Motorcycle graphs

An important concept in geometric computing is the straight skeleton [8]. The straight skeleton of a polygon is a tree-like structure similar to the medial axis [31] of the polygon, but which consists of straight segments only. Given a polygon, consider a shrinking process where each edge moves inward, at the same speed, in a direction perpendicular to itself. The straight skeleton of the polygon is the trace of the vertices through this process. Some of its applications include computing offset polygons [82], medical imaging [59], polyhedral surface reconstruction [20,155], and computational origami [67]. It is a standard tool in geometric computing software [42].

Computing motorcycle graphs is a key step of straight skeleton algorithms.

Problem 9 (Motorcycle graph)**.** Given a set of n points in the plane, each with associated directions and speeds (the motorcycles), compute the corresponding motorcycle graph. Consider the process where all the motorcycles start moving at the same time, in their respective directions and at their respective speeds. Motorcycles leave a trace behind that acts as a "wall" such that other motorcycles crash and stop if they reach it. The motorcycle graph is the set of traces.

In the motorcycles graph problem, some motorcycles crash while others escape to infinity (see Figure 3.3, top).

The current fastest algorithms for computing straight skeletons consist of two main steps [50, 115, 116]. The first step is to construct a motorcycle graph induced by the reflex (i.e., concave) vertices

of the polygon. The second step is a lower envelope computation. With current algorithms, the first step is more expensive, but it only depends on the number of reflex vertices, r, which might be smaller than the total number of vertices, n. Thus, no step dominates the other in every instance. In this section, we focus on the first step. The second step can be done in $O(n \log n)$ time for simple polygons [33], in $O(n \log n \log r)$ time for arbitrary polygons [49], and in $O(n \log n \log m)$ time for planar straight line graphs with m connected components [33].

Most existing algorithms rely on three-dimensional ray-shooting queries. Consider a data structure that maintains a set P of two-dimensional polygons in three dimensions. A ray-shooting query for such a data structure consists of a given starting point, q, and a direction, $\vec{v_q}$, and asks for the first polygon in P, if any, encountered by the ray starting at q in the direction $\vec{v_q}$.

In the context of the motorcycle graph problem, if time is seen as the third dimension, the position of a motorcycle starting to move from (x, y), at speed s, in the direction (u, v), forms a ray (if it escapes) or a segment (if it crashes) in three dimensions, starting at $(x, y, 0)$ in the direction $(u, v, 1/s)$. Therefore, the impassable traces left behind by the motorcycles correspond to infinite vertical "curtains"—wedges or trapezoidal slabs, depending on whether they are bounded below by a ray or a segment. Thus, ray-shooting queries help determine which trace a motorcycle would reach first, if any. Of course, the complication is that, as motorcycles crash, their potential traces change. Early algorithms handle this issue by computing the crashes in chronological order [50, 82]. The best previously known algorithm, by Vigneron and Yan [183], is the first that computes the crashes in non-chronological order. Our NNC-based algorithm improves upon it by reducing the number of ray-shooting queries needed from $O(n \log n)$ to $3n$, and simplifies the required data structures significantly. It is also non-chronological, but follows a completely new approach.

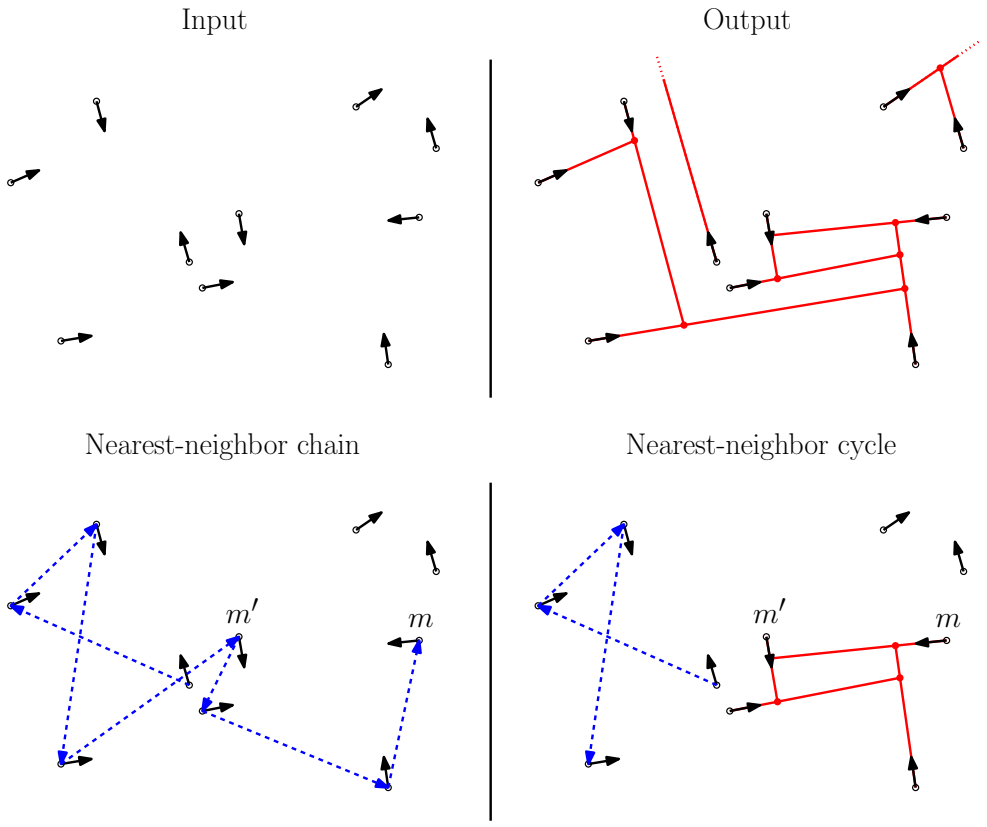


Figure 3.3: **Top:** an instance input with uniform velocities and its corresponding motorcycle graph. **Bottom:** snapshots of the NNC algorithm before and after determining all the motorcycles in a NN cycle found by the chain: the NN of the motorcycle at the top, m, is m', which is already in the chain. Note that some motorcycles in the chain have as NN motorcycles against the traces of which they do not crash in the final output. That is expected, because these motorcycles are still undetermined (e.g., as a result of clipping the curtain of m', the NN of its predecessor in the chain changes).

3.5.1 Algorithm description

In the algorithm, we distinguish between *undetermined* motorcycles, for which the final location is still unknown, and *determined* motorcycles, for which the final location is already known. We use a dynamic three-dimensional ray-shooting data structure to maintain a set of curtains, one for each motorcycle. In the data structure, determined motorcycles have wedges or slabs as curtains, depending on whether they escape or not. Undetermined motorcycles have wedge curtains like determined motorcycles that escape. Thus, curtains of undetermined motorcycles may reach points that the corresponding motorcycles never get to. When a motorcycle goes from undetermined to

determined, if it escapes, its curtain does not change. If it crashes, the curtain is "clipped" from a wedge to a slab.

For an undetermined motorcycle m, we define its "nearest neighbor" to be the motorcycle, determined or not, against which m would crash next according to the set of curtains in the data structure. Motorcycles that escape may have no NN. Finding the NN of a motorcycle m corresponds to one ray-shooting query. Note that m may actually not crash against the trace of its NN, m', if m' is undetermined and happens to crash early. On the other hand, if m' is determined, then m definitely crashes into its trace.

We begin with all motorcycles as undetermined. Our main structure is a chain (a stack) of undetermined motorcycles such that each motorcycle is the NN of the previous one. In contrast to typical applications of the NNC algorithm, here the notion of "proximity" used to define nearest neighbors is not symmetric, i.e., m may crash first into the trace of m', but m' crashes first into someone else's trace. Thus there may be no "mutual nearest neighbors". In fact, the only case where two motorcycles are mutual nearest neighbors is the degenerate case where two motorcycles reach the same point simultaneously. That said, mutual nearest neighbors have an appropriate analogue in the asymmetric setting: *nearest-neighbor cycles*, $m_1 \to m_2 \to \cdots \to m_k \to m_1$. Our algorithm relies on the following key observation: if we find a nearest-neighbor cycle of undetermined motorcycles, then each motorcycle in the cycle crashes into the next motorcycle's trace. This is easy to see from the definition of nearest neighbors, as it means that no motorcycle outside the cycle would "interrupt" the cycle by making one of them crash early. Thus, if we find such a cycle, we can determine all the motorcycles in the cycle at once. This can be seen as a type of chronological global–local equivalence, where global greedy would be to compute the crashes in strict chronological order.

See Algorithm 5. Starting from an undetermined motorcycle, following a chain of nearest neighbors inevitably leads to *(a)* a motorcycle that escapes, *(b)* a motorcycle that is already determined, or *(c)* a nearest-neighbor cycle. In all three cases, this allows us to determine the motorcycle at the

Algorithm 5 Nearest-neighbor chain algorithm for motorcycle graphs.

Initialize a ray-shooting data structure with the wedges for all the motorcycles (according to their status as undetermined).
Initialize an empty stack (the chain).
while there are undetermined motorcycles **do**
  if the chain is empty **then**
    Add an arbitrary undetermined motorcycle to it.
  Let m be the motorcycle at the top of the chain.
  Do a query for the NN of m. If there is any, denote it by m'. There are four cases:

(a) m does not have a NN: m escapes. Remove m from the chain and mark it as determined (its curtain does not change).

(b) m' is determined (i.e., its curtain is final): m crashes into it. Clip the curtain of m into a slab, mark m as determined, and remove m and the previous motorcycle from the chain. (We remove the second-to-last motorcycle because it had m as NN, and after clipping m's curtain, the previous motorcycle may have a different NN.)

(c) m' is undetermined and already in the chain: then, all the motorcycles in the chain, from m' up to m (which is the last one) form a nearest-neighbor cycle, and each of them crashes against the trace of the next motorcycle in the cycle. We clip all their curtains, mark them all as determined, and remove them and the motorcycle immediately before m' from the chain.

(d) m' is undetermined and not in the chain: add m' to the chain.

top of the chain, or, in Case (c), all the motorcycles in the cycle. See Figure 3.3, bottom. Further, note that we only modify the curtain of the newly determined motorcycle(s). Thus, if we determine the motorcycle m at the top of the chain, only the NN of the second-to-top motorcycle in the chain may have changed, and similarly in the case of the cycle. Consequently, the rest of the chain remains consistent.

3.5.2 Analysis

Clearly, every motorcycle eventually becomes determined, and we have already argued in the algorithm description that irrespective of whether it becomes determined through Case (a), (b), or (c), its final position is correct. Thus, we move on to the complexity analysis. Each "clipping"

update can be seen as an update to the ray-shooting data structure: we remove the wedge and add the slab.

Theorem 3.22. *Algorithm 5 computes the motorcycle graph in time $O(P(n)+nT(n))$, where $P(n)$ and $T(n)$ are the preprocessing time and operation time (maximum between query, insertion, and deletion) of a dynamic, three-dimensional ray-shooting data structure.*

Proof. Each iteration of the algorithm makes one ray-shooting query. At each iteration, either a motorcycle is added to the chain, or at least one motorcycle is determined (Cases (a—c)).

Motorcycles begin as undetermined and, once they become determined, they remain so. This bounds the number of Cases (a—c) to n. In Cases (b) and (c), one undetermined motorcycle may be removed from the chain. Thus, the number of undetermined motorcycles removed from the chain is at most n. Thus, there are at most $2n$ iterations where a motorcycle is added to the chain.

Overall, the algorithm takes at most $3n$ iterations, so it needs no more than $3n$ ray-shooting queries and at most n "clipping" updates where we change a triangular curtain into a slab. It follows that the runtime is $O(P(n) + nT(n))$. □

In terms of space, we only need a linear amount besides the space required by the data structure.

The previous best known algorithm runs in time $O(P(n) + n(T(n) + \log n) \log n)$ [183]. Besides ray-shooting queries, it also uses range searching data structures, which do not increase the asymptotic runtime but make the algorithm more complex.

Agarwal and Matoušek [3] give a ray-shooting data structure for curtains in $\mathbb{R}^3$ which achieves $P(n) = O(n^{4/3+\varepsilon})$ and $T(n) = O(n^{1/3+\varepsilon})$ for any $\varepsilon > 0$. Using this structure, both our algorithm and the algorithm of Vigneron and Yan [183] run in $O(n^{4/3+\varepsilon})$ time for any $\varepsilon > 0$. If both algorithms use the same ε in the ray-shooting data structure, then our algorithm is asymptotically faster by a logarithmic factor, as Vigneron and Yan need $O(n \log n)$ ray-shooting operations.

3.5.3 Special cases and remarks

Consider the case where all motorcycles start from the boundary of a simple polygon with $O(n)$ vertices, move through the inside of the polygon, and can crash against the edges of the polygon in addition to the traces of other motorcycles. In this setting, the motorcycle trajectories form a connected planar subdivision. There are dynamic ray-shooting data structures for connected planar subdivisions that achieve $T(n) = O(\log^2 n)$ [103]. Vigneron and Yan use this data structure in their algorithm to get an $O(n \log^3 n)$ time algorithm for this case [183]. Our algorithm brings this down to $O(n \log^2 n)$ time. Furthermore, their other data structures require that coordinates have $O(\log n)$ bits, while we do not have this requirement.

Vigneron and Yan also consider the case where motorcycles can only go in C different directions. They show how to reduce $T(n)$ to $O(C(\log n) \min(C, \log n))$ in this case. As a result, their algorithm runs in $O(Cn(\log^2 n) \min(C, \log n))$ time in this setting. Using the same data structures, the NNC algorithm improves the runtime to $O(Cn(\log n) \min(C, \log n))$.

A remark on the use of our algorithm for computing straight skeletons: degenerate polygons where two shrinking reflex vertices reach the same point simultaneously give rise to motorcycle graphs where two motorcycles collide head on. To compute the straight skeleton, a new motorcycle should emerge from the collision point. Our algorithm does not work if new motorcycles are added dynamically (such a motorcycle could, e.g., disrupt a NN cycle already determined), so it cannot be used in the computation of straight skeletons of degenerate polygons.

As a side note, the NNC algorithm for motorcycle graphs is reminiscent of Gale's *top trading cycle algorithm* [174] from the field of economics. This algorithm also works by finding "first-choice" cycles. We are not aware of whether they use a NNC-type algorithm to find such cycles. If they do not, they certainly can; if they do, then at least our use is new in the context of motorcycle graphs.

3.6 Server cover

Geometric *coverage* problems deal with finding optimal configurations for sets of geometric shapes that contain or "cover" another set of objects (for instance, see [4, 38, 160]). In this section, we propose an NNC-type algorithm for a problem in this category. We use NNC to speed up a greedy algorithm for a one-dimensional version of a *server cover* problem (Problem 10): given the locations of n clients and m servers, which can be seen as houses and telecommunication towers, the goal is to assign a "signal strength" to each communication tower so that they reach all the houses, minimizing the cost of transmitting the signals.

Problem 10 (Server cover)**.** Given two sets of points in $\mathbb{R}^\delta$, S (servers) and C (clients), assign a radius, r_i, to a disk centered at each server s_i in S, so that every client is contained in at least one disk. The objective function to minimize is $\sum r_i^\alpha$ for some parameter $\alpha > 0$.

The values $\alpha = 1$ and $\alpha = 2$ in Problem 10 are of special interest, as they correspond to minimizing the sum of radii and areas, respectively, in two dimensions.

3.6.1 Related work

Table 3.2 gives an overview of exact and approximation algorithms for the server cover problem. It shows that when either the dimension δ or α are larger than 1, there is a steep increase in complexity. We focus on the special case with $\delta = 1$ and $\alpha = 1$, which has received significant attention because it gives insight into the general problem.

Server cover was first considered in the one-dimensional setting by Lev-Tov and Peleg [137]. They gave an $O((n + m)^3)$-time dynamic-programming algorithm for the $\alpha = 1$ case, where n is the number of clients and m is the number of servers. They also gave a linear-time 4-approximation algorithm (assuming a sorted input). The runtime of the exact algorithm was improved to $O((n +$

Dim.	α	Approximation ratio	Complexity
2D	$\alpha > 1$	Exact	NP-hard [11]
2D	$\alpha = 1$	Exact	$O((n+m)^{881}T(n+m))$ [99]
		$1+\varepsilon$	$O((n+m)^{881}T(n+m))$ [99]
		$(1+6/k)$	$O(k^2(nm)^{\gamma+2})$ [137]
1D	$\alpha \geq 1$	Exact	Polynomial (high complexity) [29]
1D	$\alpha = 1$	Exact	$O((n+m)^2)$ [30]
		3	$O(n+m)$ [11]
		2	$O(m+n\log m)$ [11]
		2	$O(n+m)$ (our result)

Table 3.2: Summary of best known results on the server cover problem. In the table: n is the number of clients, m is the number of servers, $\varepsilon > 0$ is an arbitrarily small constant, $k > 1$ is an arbitrary integer parameter, $\gamma > 0$ is a parameter known to be a constant, and $T(n, m)$ is the cost of comparing the evaluation function for two sets of disks, which requires comparing sums of square roots to compute exactly. The exact algorithm of [99] is under the assumption that $T(n, m)$ can be computed, which depends on the computational model.

$m)^2)$ by Biniaz et al. [30]. In the approximation setting, Alt et al. [11] gave a linear-time 3-approximation algorithm and a $O(m + n \log m)$-time 2-approximation algorithm (also assuming a sorted input). Using NNC, we improve this to a linear-time 2-approximation algorithm under the same assumption that the input is sorted.

3.6.2 Global–local equivalence

The $O(m + n \log m)$-time 2-approximation by Alt et al. [11] can be described as follows: start with disks (which, in 1D, are intervals) of radius 0, and, at each step, make the smallest disk growth which covers a new client. This can be seen as a global greedy algorithm. Suppose that a client c lies outside the disk of a server s. We define the distance $d(c, s)$ between a client c and a server s as the distance between c and its closest boundary of the disk of s. Global greedy is the algorithm that repeatedly finds the closest uncovered-client–server pair, which can be seen as the globally-dominant pair, and grows the server's disk up to the client.

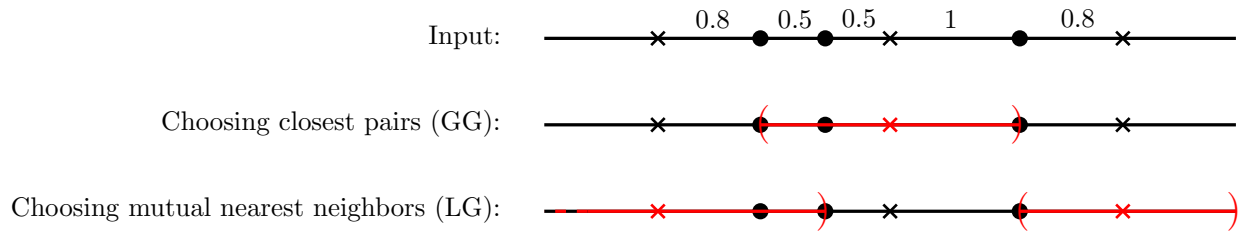


Figure 3.4: An instance (servers are crosses, clients are dots) where choosing MNN in a specific order does not result in the same solution as choosing closest pairs. Furthermore, the cost of the solution choosing MNN, 2.1, is not within a factor 2 of the optimal cost, 1.

Under this view, there is a natural notion of locally-dominant pairs/mutual nearest neighbors: an uncovered client c and a server s such that $d(c, s)$ is the smallest among all the distances involving c and s. In words, s is the server that can cover c for the least cost, and s can cover c more cheaply than any other client. Thus, we can consider a local greedy algorithm, which repeatedly finds MNN and grows the server's disk accordingly.

However, Figure 3.4 illustrates that this greedy algorithm does not satisfy global–local equivalence: matching MNN does not yield the same result as matching the closest pair. Furthermore, it shows that matching MNN loses the 2-approximation guarantee. We nevertheless use NNC to achieve a 2-approximation. This requires enhancing the algorithm so that it does not simply match MNN as defined. This illustrates that NNC may be useful even in problems where local greedy is worse than global greedy.

3.6.3 Algorithm description

Our algorithm takes a list of $n \geq 1$ clients and $m \geq 1$ servers already ordered left-to-right, without repeated coordinates, and outputs a radius for each server, which might be 0. In the algorithm, we group clients and servers into clusters. We distinguish between client clusters and server clusters. Each element starts as a base cluster, and we repeatedly merge them until there is a single cluster left.

A *client cluster* is a set of still-uncovered clients that appear continuously in the input, without servers in between. When the algorithm clusters clients together, it commits to covering them all simultaneously at a later iteration. Thus, we only need to keep track of the left-most and right-most ones, say, p and q. We represent such a cluster with the interval $[p, q]$. Each client p starts as a cluster $[p, p]$.

A *server cluster* is a set of servers and clients covered by servers in the cluster. Of all the servers in a cluster at a given iteration, only the ones with disks reaching furthest to the left and to the right may cover any new clients in future iterations. Let these servers be s_l and s_r, respectively (which might be the same). Let l be the left-most point covered by s_l, and r the right-most point covered by s_r. We represent the server cluster with the tuple $([l, r], s_l, s_r)$. Note that $l \leq s_l \leq s_r \leq r$. Each server s starts as a cluster $([s, s], s, s)$. In the algorithm, server clusters may overlap with other clusters, but when that happens they are merged immediately. The algorithm maintains the invariant that there are no uncovered client in the range $[l, r]$.

We describe the merge operation based on the cluster types.

- Two client clusters, $[p, q]$ and $[p', q']$, with $q < p'$, are merged into a single client cluster $[p, q']$.
- Two server clusters, $([p, q], s_p, s_q)$ and $([p', q'], s_{p'}, s_{q'})$, are merged into a single server cluster $([l^*, r^*], s_{l^*}, s_{r^*})$ where $l^* = \min(p, p')$, $r^* = \max(q, q')$, s_{l^*} is the server among s_p and $s_{p'}$ whose disk extends up to l^*, and s_{r^*} is the server among s_q and $s_{q'}$ whose disk extends up to r^*.
- A client cluster $[p, q]$ and a server cluster $([p', q'], s_{p'}, s_{q'})$ are merged into a single server cluster as follows. The merge involves covering all the clients in the client cluster by one of the servers in $\{s_{p'}, s_{q'}\}$, whichever is cheaper. Let that server be s, and d^* the new radius of the disk of s after it grows to cover $[p, q]$; we merge the client cluster and the server cluster into a server cluster $([l^*, r^*], s_{l^*}, s_{r^*})$, where $l^* = \min(p', s - d^*)$, $r^* = \max(q', s + d^*)$,

Algorithm 6 Nearest-neighbor chain algorithm for 1D server cover with $\alpha = 1$.

Initialize the base client and server clusters.
Initialize a stack (the chain) with the leftmost cluster.
while there is more than one cluster **do**
  Let a be the cluster at the top of the chain, and b its nearest neighbor.
  if b is to the right of a **then**
    Add b to the chain.
  else
    Merge a and b, remove them from the chain, and add $a \cup b$.
    if a server s grows to cover a client cluster c as a result of the merge **then**
      (Note that the disk of s grows on both sides of s. Thus, $C(s)$, which is $a \cup b$ at this point, might contain or overlap other clusters on the opposite side of c, as illustrated in Figure 3.5.)
      while $C(s)$ is not disjoint from other clusters **do**
        Traverse the list of clusters from $C(s)$ in the opposite direction from c.
        while the next cluster, e, is contained in, or overlaps with $C(s)$ **do**
          Merge e and $C(s)$, remove them from the chain (e might not be in the chain, if it is to the right of s, in which case only $C(s)$ is removed) and add the merged cluster to the chain.
        if the last cluster e partially overlaps $C(s)$ (i.e., e is not contained in $C(s)$) **then**
          Set c to e. (Their merge may cause the disk of s to expand on the opposite side from e, so again $C(s)$ might overlap with clusters on the opposite side of the new client c.)
        else
          ($C(s)$ is disjoint from other clusters, so we break out of the middle while loop.)

and s_{l^*} (resp. s_{r^*}) is the server among s and $s_{p'}$ (resp. s and $s_{q'}$) with the leftmost (resp. rightmost) extending disk.

The algorithm (Algorithm 6) works by building a chain (a stack) of clusters ordered from left to right. We use $a \cup b$ to denote the cluster resulting from merging clusters a and b, and $C(s)$ to denote the cluster containing a server s. The following invariant holds at the beginning of each iteration of the outer loop: no two clusters overlap, the chain contains a prefix of the list of clusters, and the distance between successive clusters in the chain decreases. We define the distance $d(a, b)$ between clusters as the distance between the closest endpoints of the clusters' intervals.

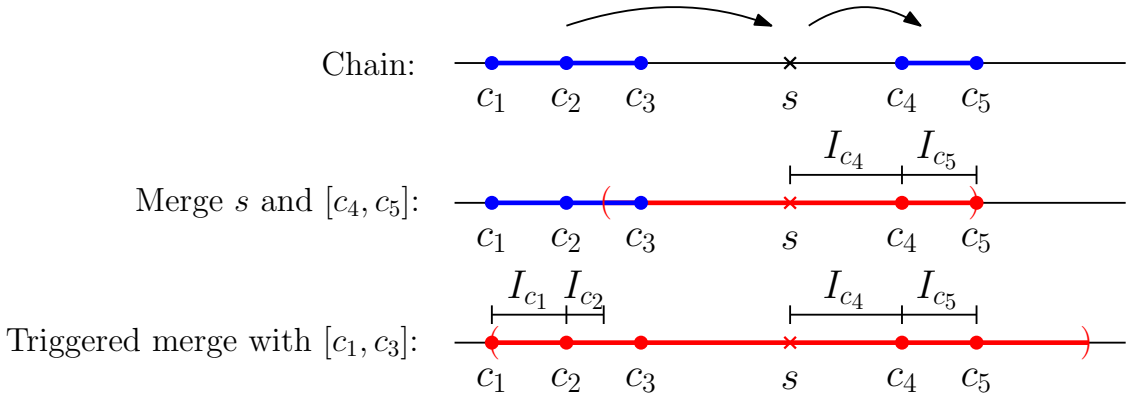


Figure 3.5: Illustration of the case where merging a server cluster and a client cluster causes the server cluster to expand on the opposite side and partially intersect another client cluster. This triggers another merge, causing the server cluster to expand again. Clients are denoted with points and servers with crosses. The coverage intervals I_c defined in the analysis are also shown.

3.6.4 Analysis

At the end of the algorithm, all the clusters have been merged into one, which is a server cluster. Thus, every client cluster has been merged with a server cluster, which means that some server grew its radius to cover the clients (or they became covered indirectly through an expansion). Therefore, the output is a valid solution. We turn our attention to the analysis of the runtime and of the 2-approximation factor. Throughout, we make an assumption that there are no ties between distances (or that they are broken consistently).

Lemma 3.23. *Algorithm 6 runs in $O(n + m)$ time, assuming the input is given in sorted order.*

Proof. Initially, there are $n + m$ clusters. Each merge operation reduces the number of clusters by one, so the number of merge operations is $n + m - 1$. A merge can be done in constant time, given that clusters have constant-size representations. Thus, the total time spent doing merges is $O(m + n)$. Each iteration of the main loop either causes at least one merge or adds a new cluster to the chain. Since clusters stay in the chain until they are merged, this can only happen $O(n+m)$ times, so there are $O(n+m)$ iterations. In 1D, finding the NN of a cluster simply involves checking the previous and next clusters, which can be done in constant time. □

Given an arbitrary problem instance, let NNC denote the solution output by Algorithm 6 and OPT denote the optimal solution. Next, we prove the approximation factor. We follow the proof idea for the greedy algorithm from [11]. We "charge" the disk radii in NNC to disjoint "coverage intervals", I_c, each of which is associated with a client c, and such that $\sum|I_c|=$ cost(NNC), where $|I|$ denotes the length of interval I.

Definition 3.24 (Coverage intervals)**.** Suppose that a server cluster S and a client cluster C are merged in Algorithm 6, and, as a result, server s is expanded to cover clients $c_1, \ldots, c_k$, in order of proximity to s. If C is to the right of s, define the coverage interval I_{c_1} as the open interval (s_b, c_1), where s_b is the right-most boundary of the disk of s before the expansion, and define I_{c_i} as (c_i, c_{i-1}) for $1 < i \leq k$. If C is to the left, the intervals are defined symmetrically.

See Figure 3.5, bottom, for an example of the coverage intervals.

Lemma 3.25. *The intervals $I_c, I_{c'}$ are disjoint if $c \neq c'$.*

Proof. If c and c' belong to the same client cluster at the time c is covered, it follows from the definition. Otherwise, let the cluster of c be the one merged with a server cluster first. Then, after c is covered, the interval I_c, if it exists, is inside a server cluster. Coverage intervals from clients covered later do not intersect existing server clusters. □

It is clear from the definition that the sum of the lengths of the coverage intervals equals the cost of NNC. If the union of these intervals (and therefore the sum of their lengths) were entirely contained within the disks in OPT, then NNC would trivially be at most double the sum of radii in OPT. If that is not the case, we show that the length of every coverage interval outside of the disks in OPT is accounted for by an equal or greater absence of coverage intervals inside a disk in OPT. For a server s, let $D_O(s)$ denote the disk of s in OPT. To offset the intervals I_c which occur outside of OPT disks, we need the following.

Remark 3.26. *Every I_c intersects or has a shared endpoint with a disk in OPT.*

This is because c must be covered by OPT.

Suppose that, for some c, I_c is not contained in any disk in OPT. Then, by Remark 3.26, I_c intersects or has a shared endpoint with a disk $D_O(s)$ in OPT. We consider the two cases where s is to the left and to the right of c separately. We show (Lemma 3.27) that, in either case, there is an interval J, between s and c and inside $D_O(s)$, which is disjoint from all coverage intervals (Figure 3.6). Note that at most one I_c may intersect $D_O(s)$ on each side on s, so the intervals J do not overlap.

Lemma 3.27. *If a client c belongs to $D_O(s)$ for some server s and I_c extends across the right (resp. left) boundary of $D_O(s)$, then there is an interval J in $D_O(s)$, to the right (resp. left) of s, free of coverage intervals, and such that $|J| > |I_c|$.*

Proof. **Right case.** Consider first the setting in Figure 3.6, right: suppose that in OPT, s covers some clients which in NNC are covered for the first time (the time where their coverage intervals are defined) from a server to the right of $D_O(s)$. Let $c_1, \ldots, c_k$, $k \geq 1$, be all such clients. Then, the coverage interval I_{c_k} of c_k extends across the right boundary of $D_O(s)$. Let x be the input element (client or server, possibly s) immediately to the left of c_1, and y the input element immediately to the right of c_k. Note that $d(c_k, y) \geq |I_{c_k}|$, since a coverage interval cannot extend past another input element.

We show that *(i)* $d(x, c_1) > d(c_k, y)$ (and thus, $d(x, c_1) > |I_{c_k}|$), and that *(ii)* the interval (x, c_1) is free of coverage intervals. Claim *(i)* follows from the fact that if $d(x, c_1) < d(c_k, y)$, then x and c_1 would be merged before y is added to the chain, which cannot happen: if x is a server, then c_1 would be covered from the left, and if x is a client, x and c_1 would be merged together, contradicting that c_1 is the left-most client covered from a server to the right of $D_O(s)$. For *(ii)*, note that c_1 was covered from the right (by definition) and x either was a client covered from the left (by definition of c_1) or a server which does not cover c_1. In the former case, I_x has its right endpoint at x, and in the latter case, x is not the client to cover c_1. Thus, there are no coverage intervals in (x, c_1).

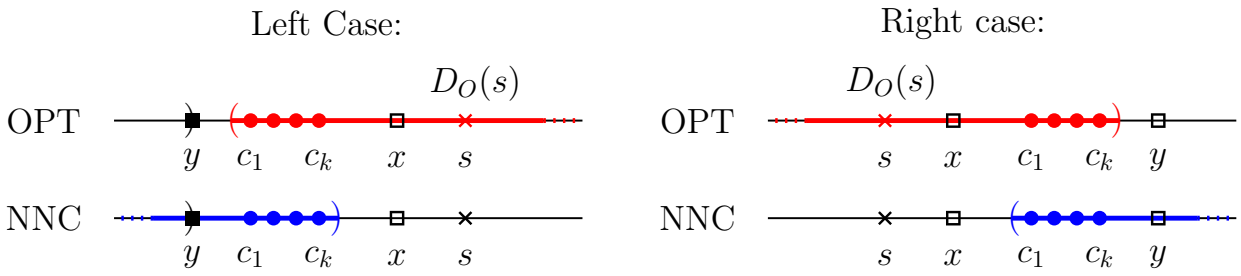


Figure 3.6: Illustration of the settings in the proof of Lemma 3.27.

Left case. Now consider the setting in Figure 3.6, left. The setting is similar, except that $c_1, \ldots, c_k$, $k \geq 1$, are to the left of s and are covered by a server to the left of $D_O(s)$, and it is the interval I_{c_1} that extends across the left boundary of $D_O(s)$. Define x as in the previous case but symmetrically: it is the input element immediately to the right of c_k. We define y slightly differently: it is the right boundary of the cluster preceding c_1 at the time c_1 is added to the chain (not necessarily an input element). Note that $d(y, c_1) \geq |I_{c_1}|$, since a coverage interval for c_1 would start at, or to the right of y.

Let u and v be the two consecutive elements among $c_1, \ldots, c_k, x$ maximizing $d(u, v)$ (note that x may be a server). We show that *(i)* $d(u, v) > d(y, c_1)$ (and thus, $d(u, v) > |I_{c_1}|$), and *(ii)* (u, v) is free of coverage intervals. For *(i)*, assume for a contradiction that $d(u, v) < d(y, c_1)$. Then, c_1 and x (and all the elements in between) would be clustered together before they are merged with y, because y would not be the NN of c_1 until all these merges between closer elements happen. However, this contradicts that c_k is the right-most client covered for the first time from the left of $D_O(s)$. Therefore, we have *(i)*.

For *(ii)*, note that if v is not x, then v and x (and all the elements in between) get clustered together before they are merged with u, because u would not be the NN of v until all these merges between closer elements happen. However, this contradicts that c_k is the right-most client covered (for the first time) from the left of $D_O(s)$. Therefore, v is x, and c_k is u. *(ii)* now follows by an analogous reasoning as in the symmetric case. □

Theorem 3.28. cost(NNC) $\leq$ 2cost(OPT).

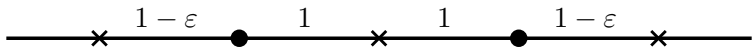


Figure 3.7: A tight example for the 2-approximation of NNC, as cost(NNC) $= 2-2\varepsilon$, cost(OPT) $=$ 1, and ε could be arbitrarily close to zero. This example is given by Alt et al. [11] also as a tight example for their greedy algorithm.

Proof. As mentioned, cost(NNC) $= \sum_c I_c$. The total length of the coverage intervals contained in OPT disks does not exceed twice the sum of the OPT radii (recall that, by Lemma 3.25, the coverage intervals are pairwise-disjoint). Consider now the parts of the coverage intervals outside the disks in OPT. For each OPT disk, there is at most one interval I_c overlapping the disk on each side, and by Remark 3.26, every I_c touches or overlaps a disk. Furthermore, by Lemma 3.27, for every length ℓ of coverage intervals in NNC outside the OPT disk of a server s to the right (resp. left) of s, there is at least ℓ length within s's disk to the right (resp. left) of s that is free of coverage intervals. Therefore the approximation ratio of two is preserved. □

See Figure 3.7 for an instance that shows that the 2-approximation is tight.

3.6.5 Global greedy in higher dimensions

As mentioned, the global greedy algorithm makes the smallest disk growth which covers a new client at each step. It achieves a 2-approximation in the 1D setting [11]. Does it achieve a good approximation ratio in higher dimensions? In this section, we give a negative answer. It performs poorly in two dimensions, even when servers are constrained to lie on a line (also known as the 1.5D case), and for $\alpha = 1$. We give an instance (Figure 3.8, left) where GG is a factor of $2m/\sqrt{5}$ worse than the optimal solution (and this ratio is not tight—it could be made worse).

In this instance, a set of m servers are placed along a horizontal line with a distance of 1 between each consecutive pair. Above each server, we place a "column" of clients stretching up to distance m above the servers. The clients in a column are evenly spaced and at distance $d = \sqrt{m^2+1} - m$

of each other. Thus, there are $m/d = m(m + \sqrt{m^2 + 1})$ clients in each column. The total number of clients is roughly $2m^3$.

Lemma 3.29. *The approximation ratio of the global greedy algorithm in the 1.5D setting with* $\alpha = 1$ *is no better than* $2m/\sqrt{5}$.

Proof. Consider the instance described above and illustrated in Figure 3.8.

The optimal solution is to cover all clients with a single server located at the center. By the Pythagorean theorem, the cost of the optimal solution is $\sqrt{5}m/2$. In contrast, we show that GG would choose to cover the clients in each column by the server at the bottom of it, resulting in a cost of m^2. Thus, GG is $2m/\sqrt{5}$ times worse than the optimal solution.

We assume that GG breaks ties by choosing clients closer to the horizontal line containing the servers first (alternatively, we can perturb the positions of the clients slightly to guarantee this tie breaking). Then, we can show that GG covers the clients by "layers", where a layer is the set of clients at a given height. To see this, assume, for the sake of an inductive argument, that GG has grown each disk to cover the clients up to a given layer. Then, some server s expands to cover a client p in next layer, which would be the one in the same column. Let s' and p' be the server and client next to s and p, respectively. We must argue that the disk of s is not closer to p' than the disk of s'. Note that it suffices to show that this does not happen when p is the very last client in the column above s. This is because the further away p is from s, the bigger the radius of the disk of s, resulting in a disk closer to p'. This is illustrated in Figure 3.8, right. The distance d between two consecutive points in a column is chosen precisely so that, in this scenario where p is the last point, the disk of s is exactly as close to p' as the disk of s'. Depending on the tie-breaking rule (or changing d to be slightly smaller), s' will grow to cover p' and not s. □

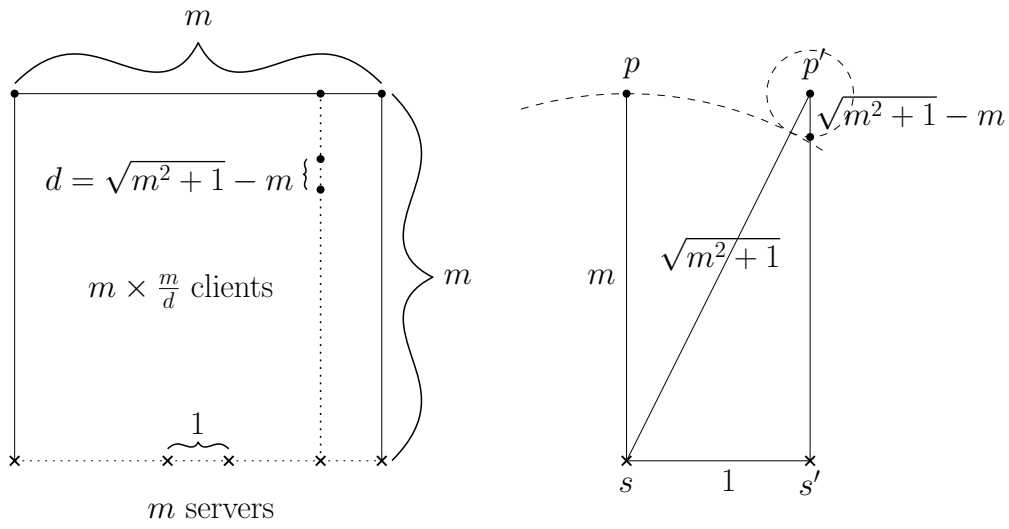


Figure 3.8: **Left:** Bad instance for the greedy algorithm for server cover. **Right:** Illustration (not to scale) that the disk of s is not closer to p' than the client below p'.

3.7 Shortest common superstring

In Section 2.3, we showed that the greedy algorithm for shortest common superstring (Problem 5) has global–local equivalence (Theorem 2.22). In this section, we develop a nearest-neighbor chain algorithm to implement local greedy (Definition 2.16). In the analysis, we argue why it may be implementable more efficiently than global greedy, as it requires a different type of string operations which seem simpler. We do not, however, provide the necessary string-based data structures. This is left as future work.

3.7.1 Background

Recall that GG repeatedly merges a pair of most-overlapping strings. The best proven approximation ratio for GG is 3.5 [124], and it is known to be no better than 2. In 1988, Tarhio and Ukkonen [177] conjectured that 2 is the true approximation ratio. Other algorithms have been shown to achieve a better approximation ratio: the best known approximation ratio is $2\frac{11}{30} \approx 2.3667$, which

builds on a $\frac{3}{4}$-approximation algorithm for maximum asymmetric TSP [159]. Unfortunately, this algorithm is relatively slow.

In practice, GG remains one of the most useful algorithms. It has been shown to perform arbitrarily close to optimum with random strings [94, 185]. However, this is not surprising because random strings barely overlap, so simply concatenating them in no particular order is already a good approximation. A much stronger result is that GG gets arbitrarily close to optimal under a smoothed analysis over random symbol changes that model DNA mutations [141]. This is important because DNA sequencing is one of the killer applications of SCS. More specifically, given any input, even one chosen by an adversary, if each symbol is changed randomly with an arbitrarily small probability, the expected approximation ratio of GG is $1 + o(1)$. This means that the approximation ratio is $1 + \varepsilon$ for any $\varepsilon > 0$, but only for sufficiently big instances where the size requirement depends on ε. Global greedy is an attractive algorithm for DNA sequencing and other applications because of the above guarantees, as well as its simplicity and faster running time.

3.7.2 First-choice chain algorithm

We call this variant of the NNC algorithm the first-choice chain (FCC) algorithm (Algorithm 7). Figure 3.9 shows a snapshot of the algorithm. The algorithm maintains a chain of strings where each string is the first choice of the previous one, with the first string being arbitrary. By following a chain of first choices, the algorithm reaches a pair of strings that is locally dominant, and merges them. After $m - 1$ merges, the algorithm returns the single string left. As in Section 2.3, we use S_0 to denote the input, and $S_1, \ldots, S_{m-1}$ to denote the intermediate string sets in the FCC algorithm after each merge of a pair of locally-dominant strings.

The functions $first(s)$ and $last(s)$ refer to the first and last *input* strings that appear in a string that is the result of "merging" input strings. E.g., if $s = aabbcc$ because it is the result of merging input strings aab, abb, and bcc, then $first(s) = aab$ and $last(s) = bcc$.

Algorithm 7 First-choice chain algorithm for SCS.

Input: A set S of non-empty strings where no string is a substring of another.
Initialize an empty stack C (the chain).
Assign a unique index from 1 to $|S|$ to each input string.
For each input string s, $first(s) \leftarrow s$, $last(s) \leftarrow s$.
$S_0 \leftarrow S$.
$i \leftarrow 0$.
while $|S_i| > 1$ **do**
    if C is empty **then**
        Add an arbitrary string from S_i to C.
    Let s be the string at the top of C.
    Find the first choice of s, i.e., find the two strings u and v in S_i that maximize the overlap between a suffix of $last(u)$ and a prefix of $first(v)$, subject to one of them being s ($s \in \{u, v\}$ and $u \neq v$). (By Lemma 2.19, the maximum overlap between a suffix of $last(u)$ and a prefix of $first(v)$ is the same as the maximum overlap between a suffix of u and a prefix of v themselves.)

    - Break ties by the index of $last(u)$.
    - Break an additional tie by the index of $first(v)$.

    Let t be the string among u and v that is not s. ($\{s, t\}$ and $\{u, v\}$ are the same strings, but s and t are defined in terms of their order in the chain, and u and v are defined in terms of the order in which they maximize overlap.)
    if t is not in C **then**
        Add t to C.
    else (t must be second-from-top in C, as shown in the analysis.)
        Remove s and t from C.
        $w \leftarrow$ the shortest superstring of s and t with u first and v last.
        $first(w) \leftarrow first(u), last(w) \leftarrow last(v)$.
        $S_{i+1} \leftarrow (S_i \setminus \{u, v\}) \cup \{w\}$.
        $i \leftarrow i + 1$
return The single string in S_i.

3.7.3 Analysis

Note the importance of breaking ties consistently: otherwise, the chain could enter an infinite loop like $abc \to bcd \to cdab \to abc \to \ldots$. Breaking ties using the adjusted overlaps from Section 2.3 guarantees that this type of cycle cannot happen: there is a unique highest adjusted overlap in a cycle, and both involved strings prefer each other over their other adjacent strings in the cycle.

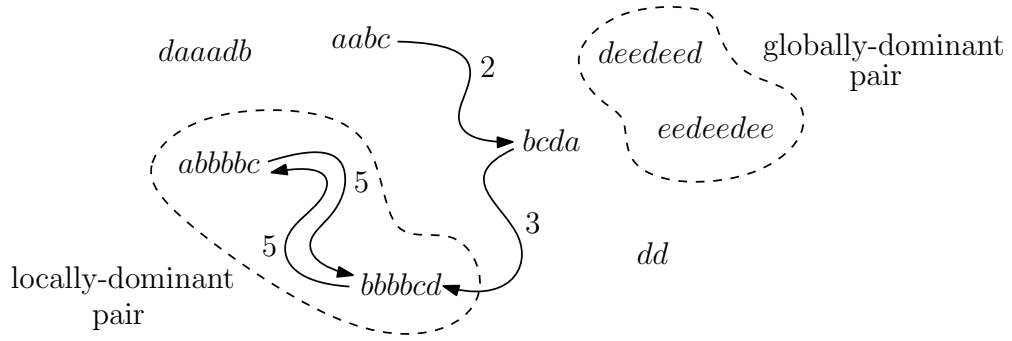


Figure 3.9: Snapshot of the FCC algorithm. The arrows show the chain of first choices, and are labeled with the overlap between strings. In this snapshot, the chain has reached a locally-dominant pair, which is then merged by the algorithm. Note that it is not the globally-dominant pair of strings that would be merged by global greedy.

We maintain the invariant that each string in the chain is followed by its first choice in the current set of strings. The reason why t must be the second-from-top in C in the 'else' clause is that, by the invariant, the adjusted overlap between pairs of consecutive strings in the chain strictly increases. Thus, if $c_1, \ldots, c_k$ were the strings in the chain, and t appeared earlier in the chain, say $t = c_j$ for some $j < k-1$, then c_{j+1} would not be the first choice of c_j, as c_k overlaps more with it, breaking the invariant.

Note that the invariant is maintained throughout. Assume it is true for a set S_i, and then the string c_k at the top of the chain is merged with a string t. This removes c_k and t from the chain. Every string that remains in the chain is also a string in S_{i+1}. Furthermore, by Lemma 2.20, their first choices do not change to be $merge(c_k, t)$. Thus, every string in the chain after the merge is a string from S_{i+1} that points to its first choice in S_{i+1}.

It is clear that every pair of merged strings is locally dominant. Thus, by Theorem 2.22, the FCC algorithm produces the same solution as global greedy. Each iteration either adds a string to the chain or merges two strings. In total, $m-1$ merges happen, so there are $m-1$ iterations of the second kind. There are $2m-1$ different strings throughout the algorithm: m input ones and $m-1$ merged ones. Each of them is added exactly once to the chain (except the very last one, which is

not added at all), as they are not removed until they are merged. Thus, the algorithm does $\leq 2m-2$ iterations of the first kind. The total number of iterations is $\leq 3m - 3$.

3.7.4 Discussion

The main algorithmic primitive of global greedy is finding the most-overlapping pair in a dynamic set of strings. It needs $m - 1$ such operations. In contrast, the algorithmic primitive in FCC is to find the first choice of a given string in a dynamic string set. It needs fewer than $3m$ such operations. Lemma 2.19 shows that, in order to find the first choice of a string at an arbitrary iteration S_i of local greedy, it suffices to look at the overlap between *input* strings. This may be useful for implementation purposes because it shows that we do not need to explicitly construct the merged strings.

The runtimes of GG and FCC are similar: both have $O(m)$ merge operations and $O(m)$ primitive operations. However, the primitives are different. We think that the primitive in FCC may be implementable more efficiently than the primitive in GG. This is because the primitive of GG has $O(m^2)$ candidate pairs, while the FCC primitive only has $O(m)$ candidate strings.

However, there is a long line of research on optimizing GG. In fact, under certain assumptions, GG can be made to run in linear time (see details below). Thus, it is not clear whether there is room for asymptotic improvement. Perhaps, FCC is only asymptotically superior in specific situations, such as under certain assumptions about the input—or about the computation model—which accentuate the advantages of the FCC primitive. Perhaps the space complexity can be improved. Developing string-based data structures for the FCC primitive which make FCC competitive or better than GG is left as future work. We conclude with some background on the analysis of the runtime of the global greedy algorithm.

Runtime of global greedy. We are interested in how fast, asymptotically, an implementation of GG can be. We analyze it in terms of the following parameters: m is the number of input strings, n is the total length of the strings, and σ is the length of the input alphabet.

To our knowledge, the fastest implementation of GG is by Ukkonen [180], which relies on the Aho–Corasick string-matching automaton (the Aho–Corasick automaton is a finite-state machine structurally similar to a trie but with additional links between its internal nodes; it was designed by Aho and Corasick [7] for string matching). It runs in $O(n)$ randomized time (hashing is used on the symbols of the alphabet) or in $O(n \min(\log m, \log \sigma))$ deterministic time and it requires $O(n \log n)$ bits of space (in the RAM model). If the alphabet is small enough to allow perfect hashing without asymptotically increasing space complexity, the algorithm runs in $O(n)$ deterministic time. For most biological purposes this is good enough, since these utilize small alphabets (e.g., DNA nucleotides).

The fastest previous algorithm, by Turner [179], is based on suffix trees and lexicographic splay trees. Its runtime is $O(n \log n)$ randomized time (or deterministic time, if the alphabet is small enough) or $O(n \log m)$ deterministic time. Recently, Alanko and Norri [10] combined Ukkonen's method [180] with an enhanced FM-index (an FM-index is a compressed substring index designed by Ferragina and Manzini; see [90]) into an algorithm that runs in $O(n \log \sigma)$ time and space (measured in bits), under the RAM model. Although this algorithm is slower for small alphabets, it improves on the $O(n \log n)$ bits of space necessary in Ukkonen's algorithm.

3.8 Geometric matching

We introduced maximum-weight matching (Problem 1) and showed global–local equivalence for it (Corollary 2.2). Hence, we can compute the matching found by GG with a NNC-type algorithm.

Problem 11 (Geometric matching)**.** Given an even-sized set of points in $(\mathbb{R}^\delta, L_p)$, find a perfect matching between the points minimizing the sum of the distances between matched points. A matching over a point set is *perfect* if every point is matched.

Geometric matching can be considered a special case of maximum-weight matching because it is equivalent to maximizing the negated distances.

Consider the global greedy algorithm for matching in the geometric setting: given a point set P of even size, repeatedly match the closest pair. This algorithm can be implemented in $O(n \log n)$ time using a closest-pair data structure (Lemma 3.6). In this section, we show that the soft nearest-neighbor chain algorithm can also achieve the same runtime. While this is not an improvement, it illustrates another application of the SNNC algorithm and provides a new way to compute the greedy matching in $O(n \log n)$ time.

3.8.1 Soft nearest-neighbor chain algorithm

We discuss how to modify the SNNC algorithm for geometric TSP (Algorithm 4) for the matching problem. See Algorithm 8 for the the SNNC algorithm for geometric matching. The input is a point set P in a metric space $(\mathbb{R}^\delta, L_p)$ with fixed dimension, where we assume that all pairwise distances are distinct. We think of a SNN answer as being a set of two points, analogously to the version of SNNC for geometric TSP.

Analysis. We omit the correctness and runtime analysis of Algorithm 8 because they are very similar to, but simpler than, those for geometric TSP (Section 3.3). They are simpler because the SNN structure only needs to maintain points instead of paths, and matched points are removed permanently (unlike connected paths, which are re-added to the set of paths). It follows from an analogous argument to the one for geometric TSP that points matched in Algorithm 8 are MNN in

Algorithm 8 Soft nearest-neighbor chain algorithm for geometric matching.

```
Initialize an empty stack (the chain).
Initialize a SNN structure S with the input points.
while there are unmatched points do
    if the chain is empty then
        Add a node with an arbitrary pair of points from S to it.
    else
        Let U = {u, v} be the node at the top of the chain.
        Remove u from S, query S with u, and re-add u to S.
        Remove v from S, query S with v, and re-add v to S.
        Let A be the best answer.
        if A = U then
            Match u and v, remove them from S, and remove U from the chain.
            if the chain is not empty and the new last node, V, contains u or v then
                Remove V from the chain.
        else
            Add A to the chain.
```

the set of unmatched points. Likewise, it is easy to see that the total number of iterations is linear. Again, we get a runtime of $O(P(n) + nT(n))$ time, where $P(n)$ and $T(n)$ are the preprocessing time and operation time (maximum between query and deletion) of a SNN data structure. Using our implementation, we get the following.

Lemma 3.30. *In any fixed dimension and for any L_p metric, the soft nearest-neighbor algorithm for geometric matching outputs the same solution as the global greedy algorithm in $O(n \log n)$ time.*

3.9 Combinatorial optimization problems

We conclude the chapter by revisiting the COPs that we discussed in Section 2.1. Recall the class of COPs that we considered (Definition 2.4), the interaction graph induced by the evaluation function for a COP (Definition 2.5), the local greedy algorithm for COPs (Definition 2.6), and Theorem 2.9 on global–local equivalence, which states that if the evaluation function is deteriorating (Definition 2.7), then LG outputs the same solution as GG.

In Section 2.1, we did not provide an algorithm for actually finding locally-dominant elements. If the evaluation function is deteriorating, we can adapt the nearest-neighbor chain algorithm to implement local greedy. In this context, we call it the *best-neighbor chain* (BNC) algorithm.

Algorithm 9 is the BNC algorithm for a maximization or minimization COP. It does not make explicit how to traverse the interaction graph or how to evaluate the elements, as this is better considered on a case-by-case basis. For instance, depending on the problem, it may be better to recompute the evaluations as needed, or to compute them once at the beginning and maintain them as the solution evolves. Recall that an element u is valid for a partial solution S if $S + u$ is a valid solution or a subset of one.

Algorithm 9 Best-neighbor chain algorithm for COPs with deteriorating evaluation functions.

Input: a set X and any information needed to compute the evaluation function.
Initialize the solution, S, empty.
Initialize an empty stack (the chain).
Let $G = (X, E)$ be the interaction graph.
while S is not maximal (for maximization) or S is invalid (for minimization) **do**
    if the chain is empty **then**
        Add an arbitrary valid element to it.
    Let u be the element at the top of the chain.
    if u has no valid neighbors in G **then**
        Add u to S.
        Remove u from the chain.
    else
        Let v be the best valid neighbor of u in G.
        if v is better than u **then**
            Add v to the chain.
        else
            Add u to S.
            Remove u from the chain.
            if the chain is not empty and the new top element has become invalid or worse **then**
                Remove the new top element from the chain.
return S.

Correctness. The BNC algorithm builds a chain of elements of X. We maintain the invariant that each element in the chain is valid and the best neighbor of its predecessor. With this invariant, the chain cannot grow indefinitely, so eventually a locally-dominant element is found and added to

the solution. We need to argue that the invariant is maintained when a locally-dominant element is added to the solution. Suppose that the last three elements in the chain are $a, b,$ and c, with a before b and b before c, and that c is locally dominant. Since the evaluation function, h, is deteriorating, when we pick c, the neighbors of c in the interaction graph become worse or invalid. If the evaluation of an element that is not in the chain becomes worse or invalid, the invariant remains intact. The invariant can only break if an element in the chain becomes worse or invalid and stops beings the best neighbor of its predecessor. However, we can see that the only neighbor of c in the chain is b. This is because c is the best element in the entire chain, so, if c were a neighbor of a prior element in the chain, then that element would not be followed by its best neighbor. Thus, of all the elements in the chain, only the evaluation of b can change. It may stop being the best neighbor of a. This is why, when we add c to the solution, we also remove its predecessor from the chain, b, if its evaluation changes. Doing so preserves the invariant.

Note that the BNC algorithm does not work if h is not deteriorating, because the invariant is not preserved when adding an element to the solution. A locally-dominant element c at the top of the chain can have neighbors in common with elements anywhere in the chain. Thus, if h is not deteriorating, when adding c to the solution, the neighbors of nodes deep in the chain could get better and become the new best neighbors of these nodes in the chain. The algorithm works when h is deteriorating because only the penultimate link in the chain can get "corrupted".

Runtime analysis. Let $n = |X|$. There are two types of iterations: (1) an element is added to the chain or (2) an element is added to the solution. The number of iterations of type (2) is at most n. For each element added to the solution, at most two elements are removed from the chain. This bounds the number of removed elements, and, consequently, the number of iterations of type (1), to $2n$. It follows that the total number of iterations is at most $3n$. The total runtime is $O(P(n) + nT(n))$, where $T(n)$ is the cost of finding the best neighbors of an element in the interaction graph, and $P(n)$ is the time needed for any preprocessing.

We can refine this analysis to an *output-sensitive* bound, i.e., we can bound the runtime more precisely as a function of not only the input size, but also the output size. The runtime of the BNC algorithm for a maximization COP is actually $O(P(n) + kT(n))$, where k is the size of the solution found. The number of iterations of type (2) is k by definition. Thus, the number of elements removed from the chain is bounded by $2k$. For a maximization COP, the chain always ends empty, because the chain contains only valid elements, so, as long as the chain is not empty, the solution is not maximal. This bounds the number of iterations of type (1) to $2k$, and the total number of iterations to $3k$. Similarly, for minimization COPs, the number of iterations is bounded to be linear on the size of the solution plus the maximum chain length, since the chain does not necessarily end empty.

3.9.1 Problems

Unlike for geometric problems, we do not have data structures to speed up the best-neighbor search. In a COP, the most straightforward way to find the best neighbor of an element is to scan all its neighbors in the interaction graph. Assuming that we can iterate through the neighbors of an element and compute (or maintain) their evaluations in constant time per element, we get that $T(n)$ is the maximum degree of the interaction graph.

The following examples show that it is more difficult to obtain a speedup over global greedy in COPs compared to geometric problems. We only obtain speedups in special cases.

Maximum-weight independent set. For the reverse greedy algorithm for maximum-weight independent set (Section 2.1.1), we can find the best neighbor of a node in time $T(n) = O(\Delta)$, where n is the number of nodes and Δ is the maximum degree of the input graph. Thus, the BNC algorithm computes the same solution as GG in $O(n\Delta)$ time. By comparison, a straightforward implementation of GG takes $O(m \log n)$ time, where m is the number of edges (note that m is

always smaller than $n\Delta$). This time can be achieved maintaining a min-heap of the nodes, using their evaluations as keys. The globally-dominant node is found with a extract-min operation, which is done $O(n)$ times. Each time a node is added to the solution, we increase the evaluation of its neighbors. There are $O(m)$ such updates. In total, we do $O(n + m)$ min-heap operations, each of which takes $O(\log n)$ time[10]. Thus, the BNC algorithm is faster for graphs with constant maximum degree ($O(n)$ instead of $O(n \log n)$).

Set cover. For set cover (Problem 2), let n be the number of sets, m the sum of the sets' sizes, k the maximum set size, and f the maximum frequency among the elements, i.e., every element appears in at most f sets. In the graph representation (Figure 2.3, Center), m is the total number of edges, k is the maximum degree among the sets, and f is the maximum degrees among the elements. Each set interacts with the sets with which it shares an element, so the number of neighbors of a set in the interaction graph is at most kf. We get $T(n) = O(kf)$, so we can compute the same solution as GG in $O(nkf)$ time.

By comparison, a min-heap implementation of GG takes $O(m \log n)$ time. As in the case of maximum-weight independent set, GG needs a linear number of min-heap operations, while tne BNC algorithm needs a linear number of best-neighbor searches, neither of which dominates the other in all instances.

Maximum-weight matching. Let n and m be the number of nodes and edges in a graph, respectively. For maximum-weight matching (Problem 1), the BNC algorithm runs in $O(n\Delta)$ time, where Δ is the maximum degree of the input graph. This uses the output-sensitive bound, since there can be up to $\Theta(n^2)$ edges, but any matching has at most $n/2$ edges. We can find the best neighbor of an edge in $T(n) = O(\Delta)$ time by scanning the adjacency lists of the two endpoints of the edge.

[10]Some min-heap implementations, such as Fibonacci heaps [93], allow $O(1)$-time reduce-key operations, but not increase-key operations.

For graphs where the average degree is of the same order as the maximum degree, such as graphs with constant maximum degree or dense graphs ($m = \Theta(n^2)$), the BNC algorithm runs in linear time ($O(n + m)$). For any other type of graph, Preis' algorithm [162] is a superior alternative, as it also implements local greedy but runs in linear time for every graph.

Preis' algorithm [162] is an interesting variant of the BNC algorithm. It also builds a chain of adjacent and increasingly heavier edges. However, since finding best neighbors is too expensive, it continues the chain as soon as it finds a *better* neighbor of the edge at the top of the chain, even though it may not be the *best* neighbor. This suffices to converge to a locally-dominant edge (our motivation for the soft nearest-neighbor chain algorithm (Algorithm 4) is similar). The price to pay for relaxing the chain-successor rule like that is that, when a locally-dominant edge is found, it may be a neighbor of any number of edges in the chain—not just the second-from-top. Thus, when the top of the chain backtracks to a previous edge $\{u, v\}$, it may find that u or v, or both, have already been matched, and we are left with an edge at the top of the chain which is invalid (note that we avoided this complication in all our NNC-based algorithms). Preis' elegant solution relies on the fact that the neighbors of an edge form two cliques in the interaction graph, one for each endpoint, and that choosing any edge in one of these cliques invalidates all the edges in that clique. When scanning for a better neighbor of an edge $\{u, v\}$, Preis' algorithm alternates between checking edges incident to u and edges incident to v. Further, edges already checked from a previous edge in the chain are not checked again, for if they were not better than a previous edge in the chain, they cannot be better than the current one. As a result, if the chain backtracks to $\{u, v\}$ and $\{u, v\}$ is invalid because, e.g., u has been matched, then at least half the edges checked from $\{u, v\}$ can be discarded permanently (those incident to u). Since the number of discarded edges over the entire graph is bounded by m, the total time spent at the edges checking for better neighbors is $O(m)$.

Preis' approach can be generalized to problems where the neighbors of every element in the interaction graph form a constant number of cliques of mutually-exclusive elements (e.g., maximum-

weight matching in hypergraphs with constant hyperedge size), but is not powerful enough for arbitrary COPs.

The assignment problem. The assignment problem [135] is a special case of maximum-weight matching (Problem 1), and thus a COP. It corresponds to the case where the input graph is a complete bipartite graph. A graph is *bipartite* if the nodes can be partitioned into two sets, A and B, such that all the edges have an endpoint in A and the other in B. There are exact algorithms especially for this case, such as the Hungarian algorithm [135].

Problem 12 (Assignment problem)**.** Given two sets, A and B, of n elements each, and a weight function $w : A \times B \to \mathbb{R}^+$, find a perfect matching between the elements of A and B of maximum weight.

Since the assignment problem is a special case of maximum-weight matching in a dense graph with $\Theta(n^2)$ edges, the BNC algorithm runs in linear time ($O(n^2)$). This matches the runtime of Preis' algorithm [162] while being simpler to implement.

3.10 Conclusions

Before this work, the nearest-neighbor chain algorithm had only been used in agglomerative hierarchical clustering. The new applications in this chapter (and the next) showcase the versatility of global–local equivalence and the nearest-neighbor chain algorithm. Here are some guidelines for designing NNC-based algorithms.

When dealing with a greedy algorithm, one may check if a form of global–local equivalence holds. The hybrid method described in Chapter 2 is a useful tool to do so. If it does, one should then consider using the NNC algorithm. To design a NNC-type algorithm: *(i)* each link in the chain should be better than the previous, so that progress towards a locally-dominant element is made

with each step; *(ii)* to avoid infinite loops, the chain should remain acyclic. For this, one should be careful to break ties consistently; *(iii)* after finding and processing a locally-dominant element, one should check that all the remaining links in the chain stay valid.

These simple ingredients are likely to lead to an algorithm with a runtime of the form $O(P(n) + nT(n))$, as seen throughout this chapter. Here, $P(n)$ and $T(n)$ are the preprocessing and operation time of a dynamic nearest-neighbor data structure, which might take different forms depending on the problem.

A direct way to speed up NNC algorithms is to improve the underlying data structures. We have mostly used fully dynamic data structures that allow insertions and deletions, which is "overkill" for our needs. NNC-type algorithms typically only use deletions. Further, all the potential query points are generally known at construction time. This motivates research on specialized data structures with these considerations in mind.

The NNC algorithm is not necessarily the only efficient way to implement local greedy. In parallel and distributed settings, it makes sense to try to detect several locally-dominant elements simultaneously. However, even in the sequential setting that we have considered there may be other algorithms besides NNC. Finding new approaches is left as future work. As a first step, our soft nearest-neighbor chain algorithm (Algorithm 4) relaxes the condition that each element in the chain must be followed by its nearest neighbor. Preis' algorithm [162] also uses a chain and relaxes it in a different context (see the discussion in Section 3.9.1). It goes without saying that chains need not be used at all.

Chapter 4

Symmetric Stable Matching

4.1 Background

The theory of *stable matchings* studies how to match entities in two sets, each of which has its own preferences about the elements of the other set, in a "stable" manner. It is a central concept in market design. Some surveys or books on the subject include [107, 118]. One of its original uses was to match hospitals and medical students starting their residencies in the US [153]. It is also used in on-line advertisement auctions [6]. It was originally formulated by Gale and Shapley [97] in the context of establishing marriages between n heterosexual men and women, where each man ranks the women by preference, and the women rank the men. This is why the stable matching problem is also known as the stable marriage problem. The main definition of stability is that a matching between the men and women is *stable* if there is no *blocking pair*: a man and woman who prefer each other over their assigned partners under the matching. In this case, stability is necessary (and more important than, e.g., total utility) to prevent extramarital affairs.

Gale and Shapley [97] show that at least one stable matching exists for any set of preferences, and it might not be unique. They also provided an algorithm that runs in $O(n^2)$ time (where n is the

size of the sets) for computing a stable matching. This is known as the Gale–Shapley algorithm or the deferred-acceptance algorithm.

Briefly, the deferred-acceptance algorithm works as follows. Initially, everyone is unmarried. One of the sets (e.g., the men) is in charge of proposing to the other set. The algorithm runs until everyone is matched. At each step, an unmatched man proposes to his most preferred woman who has not rejected him yet. If the woman is unmatched, she accepts him for the time being. If the woman is matched, she accepts him, also for the time being, only if she prefers him over her current match. In this case, the man that was her current match becomes unmatched. Gale and Shapley proved that this process terminates with everyone matched in a stable matching.

For arbitrary preference lists, the runtime of the deferred-acceptance algorithm is worst-case optimal. Note that storing all the preferences already requires $\Theta(n^2)$ space. Thus, just reading the input requires $\Theta(n^2)$ time, but the lower bound holds even when the preferences are already in memory. In fact, quadratic lower bounds are also known for "simpler" questions, like verifying stability of a given matching [101]. This inspired work on finding subquadratic algorithms in restricted settings where preferences can be specified in subquadratic space. Such models are collectively called *succinct stable matching* because they require less space [132].

The question is whether stable matching requires $O(n^2)$ time simply because of the raw amount of data, or because of some other intrinsic difficulty about finding stable matchings. Künnemann et al. [132] showed that some succinct models have a rich enough structure to allow for faster algorithms, while others cannot be solved faster than $O(n^2)$ time even though their preferences can be specified in much less space (more details in Section 4.5). In this section, we present some stable matching models that fall in the category of succinct stable matching. We will see that, in our case, it is possible to achieve subquadratic time.

When generalized to the one-to-many setting, the stable matching problem is also known as the *college admission problem* [169] and can be formulated as an assignment of n students to $m \leq$

n colleges, where each student has a preference ranking of the colleges and each college has a preference ranking of the students and a *quota* indicating how many students it can accept. For simplicity, we assume here that the sum of all the quotas is n, although this constraint is not essential. The deferred-acceptance algorithm can be adapted to this setting, taking $O(nm)$ time.

Stable matching in the local greedy framework. The core mechanic of the deferred-acceptance algorithm is that accepted proposals are *not* permanent. Thus, it is not possible to match agents permanently in a greedy fashion in a stable matching algorithm. In this chapter, we introduce a special case of stable matching, which we call *symmetric stable matching*. We show that, unlike the general case, symmetric stable matching can be solved with a greedy algorithm. We also define a local greedy algorithm, which we call the soul-mate algorithm, and show that global–local equivalence holds in this setting. Then, we discuss three special cases of symmetric stable matching, which we call geometric stable matching, geographic stable matching, and a "narcissistic" variant of the k-attribute model [26]. In each case, we adapt the nearest-neighbor chain algorithm from hierarchical clustering to implement the soul-mate algorithm and improve upon the runtime of the deferred-acceptance algorithm.

4.2 Symmetric stable matching

We present symmetric stable matching in the one-to-many context, and often use "colleges" and "students" to refer to the sets. Naturally, the results also apply to the more restricted one-to-one case.

In the original setting, preferences are ordinal: each agent (college or student) ranks the agents in the other set by preference. In order to formulate the symmetric stable matching problem, consider this alternative way to derive ordinal preferences: each agent gives a unique numeric score to each

agent from the other set, and ranks them in increasing order of these scores. For instance, a set of scores such as $(a \leftarrow 7, b \leftarrow 2, c \leftarrow 10)$ corresponds to the list of preferences $c > a > b$.

We call the scores *symmetric* when every pair of agents agree on their reciprocal score. Formally:

Definition 4.1. A stable matching problem between two sets X and Y is *symmetric* if *(i)* the preferences of each agent $x \in X$ are consistent with the ranking of the agents of Y according to an injective function $score_x : Y \mapsto \mathbb{R}$, and analogously for the agents in Y; *(ii)* for each pair of agents $x \in X, y \in Y$, $score_x(y) = score_y(x)$.

4.2.1 Greedy algorithm

Assume that all the scores are different except for reciprocal pairs. Let s and x be the student and college such that $score_x(s)$ is maximum among all the scores. It is easy to see that s and x must be matched together in every stable matching. Assuming that we have access to the score functions and not just to the corresponding ordinal preferences, this suggests a greedy algorithm: repeatedly match the pair realizing the highest score among the unmatched students and the colleges with unfulfilled quota. This produces a unique matching, which we call the greedy matching.

Lemma 4.2. *Given a stable matching instance with symmetric preferences, the greedy matching is stable.*

We omit the proof of Lemma 4.2, as it is a direct consequence of Theorem 4.6 below.

Lemma 4.3. *Every symmetric stable matching instance has a unique solution.*

Proof. Let S be any stable solution for a given instance with symmetric preferences. At least one solution exists, as proved by Gale and Shapley (or, by Lemma 4.2). Let s and x be the pair realizing the maximum score. Clearly, s and x are matched to each other in S; if they were matched to other agents, they would both prefer each other more, so they would form a blocking pair.

If we remove s from the given instance and reduce the quota of x by one, we obtain a smaller instance such that the restriction of S to the smaller instance is still stable and such that the solution of the greedy algorithm to the smaller problem agrees with its solution to the whole problem. The result follows by induction on the size of the problem. □

Uniqueness (Lemma 4.3) is a special property of symmetric stable matching compared to the general setting. It also relies on the assumption of no ties in the scores. Incidentally, Eeckhout [76] states a sufficient condition for a unique solution in the one-to-one setting. It can be shown that symmetric preferences satisfy this condition, and hence uniqueness in the one-to-one setting also follows from their result.

4.2.2 Local greedy algorithm

Using the terminology from Chapter 2, the pair with the overall highest score is the globally-dominant pair, and the greedy algorithm is global greedy. We now define a local greedy algorithm.

Definition 4.4. In a stable matching instance, a pair of *soul mates* are a college and a student who have each other as first choice.

Remark 4.5. *If preferences are symmetric, a pair of soul mates exists.*

In particular, the agents in the globally-dominant pair are soul mates, but there may be other pairs of soul mates. Remark 4.5 would not hold without symmetric preferences. However, we do not need to have access to the score functions in order to determine if two agents are soul mates, only to their preference lists.

Our local greedy algorithm, also called the *soul-mate algorithm*, repeatedly matches soul mates (Algorithm 10).

Algorithm 10 Soul-mate algorithm for symmetric stable matching.

Input: n students and m colleges with symmetric preferences, and college quotas adding up to n.
Initialize the matching empty.
while there is an unmatched student **do**
    Find soul mates s, x.
    Match s and x, remove s from the pool of unmatched students, reduce the quota of x by one, and remove x from the pool of unmatched colleges if its quota reached zero.

Theorem 4.6. *Given a stable matching instance with symmetric preferences, the soul-mate algorithm computes a stable matching.*

Proof. Due to Remark 4.5, the algorithm never fails to find soul mates, so it terminates. Let s be an arbitrary student, and x the college that s is assigned to in the matching computed by an arbitrary run of the soul-mate algorithm. Let y be some other college such that s prefers y over x. We argue that y does not prefer s back over its assigned students, and, thus, s and y are not a blocking pair. When s and x were matched by the algorithm, s and x were soul mates, so, in particular, x was the first choice of s among the remaining colleges. This means that y had already fulfilled its quota. Therefore, y was matched with all of its students while s was available. Clearly, y prefers those students over s. □

Using the terminology from Chapter 2, soul mates can be seen as locally-dominant pairs. As usual, global greedy is a special case of local greedy, which has some non-determinism: if there are multiple pairs of soul mates, any of them may be chosen. By Lemma 4.3, every possible run outputs the same (unique) solution. This is in contrast to the traditional deferred-acceptance algorithm, where there is also some freedom of choice, and it can actually affect the final matching. For instance, it is known that the deferred-acceptance algorithm favors the side making the proposals [97].

Algorithm 10 leaves open how to actually find soul mates—it only shows that any strategy that finds soul mates will produce the unique stable solution regardless of the order. Thus, there may be different strategies for finding soul mates. Global greedy is a naive but clearly valid strategy

for finding soul mates. We propose a different algorithm based on the the nearest-neighbor chain algorithm from hierarchical clustering (Algorithm 1).

4.2.3 First-choice chain algorithm

Algorithm 11, which we call the *first-choice chain* (FCC) algorithm, is based on the NNC algorithm. It relies on a *first-choice data structure*. This structure should be able to maintain a set of agents of the same type (students or colleges) and answer queries asking for the first choice of a query agent of the opposite set. Moreover, it should support deletions, that is, allow to remove elements from the set. This is called a semi-dynamic data structure, as it does not need to support insertions for this algorithm.

Algorithm 11 First-choice chain algorithm for symmetric stable matching.

Input: n students and m colleges with symmetric preferences, and college quotas adding up to n.
Initialize the matching as empty.
Initialize a dynamic first-choice structure containing the students, and one containing the colleges.
Initialize an empty stack S (the chain).
while there is an unmatched student **do**
    if S is empty **then**
        Add any unmatched student to it.
    else
        Let p be the agent at the top of the stack.
        Query the first-choice structure of the opposite set to find q, the first choice of p.
        if q is not already in S **then**
            Add q to S.
        else (q is be the penultimate element in S, as justified below.)
            Match p and q.
            Remove the student from the first-choice structure of students.
            Reduce the quota of the college by one and remove it from the first-choice structure of colleges if its quota reached zero.
            Remove p and q from S.

Note that, in the chain in Algorithm 11, the scores between consecutive elements in S strictly increase (recall that the score functions are required to be injective). That is why, if q is already in

the stack, it must be the second-from-top; if q were anywhere else, p would prefer its predecessor in S over q, contradicting that q is the first choice of p.

Theorem 4.7. *Given a first-choice data structure with $P(n)$ preprocessing time and $T(n)$ operation time (maximum between query and deletion), a symmetric stable matching problem can be solved in $O(P(n) + nT(n))$ time.*

Proof. Since the algorithm only matches soul mates, it implements the soul-mate algorithm; correctness follows by Theorem 4.6.

For the runtime, note that each iteration that pushes a new element in the stack can be charged against a later pop operation in the stack and its associated match. Since the number of matches is n, the total number of iterations is $2n$. Each iteration takes $O(T(n))$ time, as everything else takes constant time. □

A possible optimization that does not affect the asymptotic analysis is to note that, if a match happens and the matched college is below the matched student in the stack and it still has positive quota, we can keep the college in the stack. That college would be added to the stack again in the next iteration, as it would still be the first choice of the previous student in the stack.

Next, we discuss several stable matching models with symmetric preferences, and discuss the appropriate first-choice data structures in each case. For simplicity, we define them in the one-to-one setting.

4.3 Geometric model

Consider a setting where the two sets of agents are points in a metric space, and they rank the agents of the other set by proximity. This is a naturally occurring model: in markets, connecting nearby

buyers and sellers reduces transportation costs. Some assignments of students to public schools already take into account proximity, and so do dating apps. Political redistricting, discussed in Chapter 7, can also use distance-based stability to draw fair and compact districts. This geometric model of stable matching was studied by Arkin et al. [12].

Problem 13 (Geometric stable matching)**.** Find a stable matching between two sets of n points in a metric space, where a point p prefers q over q' if and only if $d(p, q) < d(p, q')$.

By definition, distances in a metric space are symmetric. That is, $d(x, y) = d(y, x)$. Thus, this model is symmetric. The globally-dominant pair is the closest pair of agents of different sets. Assuming that distances are unique, we can find the globally-dominant pair using a bichromatic closest pair data structure (Definition 3.7). This data structure also allows us to remove the agents once they are matched. By Lemma 3.8, we can implement global greedy in $O(n \log^4 n)$ time for points in the plane and under Euclidean distance.

Alternatively, we can use the FCC algorithm (Algorithm 11). In this setting, the first-choice data structure is a dynamic nearest-neighbor data structure. By Theorem 4.7 and Lemma 3.2, we can implement the FCC algorithm in the same runtime of $O(n \log^4 n)$ for points in the plane and under Euclidean distance[11]. Comparatively, the Gale–Shapley algorithm takes $O(n^2 \log n)$ time. As mentioned, the Gale–Shapley algorithm typically requires $O(n^2)$ time. However, in this setting we are not given the preferences explicitly, so the bottleneck of the algorithm is actually sorting the n students by distance from each of the n colleges, and vice-versa, in order to know their preferences.

4.4 Geographic model

Instead of the geometric setting, we could consider a graph-based setting. Here, the agents of the two sets are nodes in an underlying graph. For example, the graph could represent a road

[11] Before recent data structures improvements by Chan [46], the FCC algorithm was faster than global greedy by a factor of $\Theta(\log^2 n)$.

network. This model is also symmetric because, in an undirected graph, shortest-path distances are symmetric.

Problem 14 (Geographic stable matching)**.** Given a connected, undirected, n-node graph G with positively-weighted edges, find a stable matching between two subsets of nodes in G of size k each, where a node p prefers q over q' if and only if $d(p, q) < d(p, q')$, and d denotes shortest-path distance.

If G is from a hereditary graph class with $O(n^c)$-size separators which can be computed in $O(n^{1+c})$ time, where $c < 1$, we can use our nearest-neighbor data structure for graphs from Chapter 5 (see the chapter for the relevant definitions). By combining Theorem 4.7 and Theorem 5.4, a stable matching based on proximity preferences for such graphs can be found in $O(n^{1+c} \log k)$ time. For instance, this is $O(n^{1.5} \log k)$ for planar graphs and graphs that represent real-world road networks accurately (i.e., graphs with sparse crossing graphs [87]).

By comparison, the Gale–Shapley algorithm takes $O(kn \log n)$ time for sparse graphs (all graphs for which we can use the data structure from Chapter 5 are sparse). In this setting, the Gale–Shapley algorithm requires computing the preferences, that is, the shortest-path distances from each college to the students and vice-versa. This step dominates the $O(k^2)$ time for performing the Gale–Shapley algorithm itself. The preferences can be computed by applying a single-source shortest-path algorithm starting from each agent, such as Dijkstra's algorithm [70].

The runtime of Gale–Shapley can be improved to $O(kn)$ for planar graphs or, more generally, graphs from a hereditary graph class with sublinear separators and for which a certain subdivision can be constructed in $O(kn)$ time [110] (see Chapter 5). This subdivision allows us to compute the distances from a given node to every other node in linear time. In any case, the first-choice chain algorithm improves upon Gale–Shapley for sufficiently large values of k.

4.5 Narcissistic k-attribute model

We introduce the *narcissistic k-attribute* model, a special case of the k-attribute model, and show that it has symmetric preferences. We use this fact to solve it efficiently using the first-choice chain algorithm.

The k-attribute model [26] is defined as follows. Each agent p has a vector $\vec{p_a}$ of k numerical attributes, and a vector $\vec{p_w}$ of k weights according to how much p values each attribute in a match. Each agent p ranks the agents in the other set according to the score function $f_p(q) = \vec{p_w} \cdot \vec{q_a}$, the linear combination of the attributes of q according to the weights of p.

"Narcissistic" stable matching is not a concrete model. Instead, the term narcissistic is used to describe models where the preferences of each agent reflect their own qualities in some way (e.g., see [48, 132]). We consider the natural narcissistic interpretation of the k-attribute model, where $\vec{p_a} = \vec{p_w}$ for every agent. That is, each agent weights each attribute according to its own value in that attribute. To illustrate this model, consider a centralized dating service where k attributes are known for each person, such as income, intelligence, and so on. In the general k-attribute model, each person assigns weights to the attributes according to their preferences. The narcissistic assumption that $\vec{p_a} = \vec{p_w}$ implies that someone with, say, a high income, values income more than someone with a relatively smaller income.

Here, we consider the one-to-one setting and make a general position assumption that there are no ties in the preference list of each agent. In addition, in this model each agent is uniquely determined by its attribute vector, so we do not distinguish between the agents themselves and their k-dimensional vectors. We obtain the following formal problem.

Problem 15 (Narcissistic k-attribute stable matching)**.** Find a stable matching between two sets of n vectors in $\mathbb{R}^k$, where a vector $\vec{p}$ prefers $\vec{q}$ over $\vec{q'}$ if and only if $\vec{p} \cdot \vec{q} > \vec{p} \cdot \vec{q'}$.

The runtime of the first-choice chain algorithm for this model, with the data structures we will discuss, can be made to be $O(n^{2-4/(k+2+\varepsilon)})$ for any $\varepsilon > 0$. Without the narcissistic assumption, the k-attribute model becomes less tractable. Künnemann et al. [132] showed that no strongly subquadratic-time algorithm exists if $k = \omega(\log n)$ assuming the Strong Exponential Time Hypothesis [43], even if the weights and attributes take Boolean values. Similarly to us, Künnemann et al. also studied some restricted cases that can be solved by subquadratic algorithms. They presented a $O(C^{2k}n(k + \log n))$ time algorithm for the case where the attributes and weights may have only C different values, and a $\tilde{O}(n^{2-1/\lfloor k/2 \rfloor})$ time algorithm for the asymmetric case where one of the sets has a single attribute and the other has k.[12]

It is easy to see that our setting is symmetric: due to narcissistic preferences, for every two agents p and q, we have $f_p(q) = \vec{p_w} \cdot \vec{q_a} = \vec{p_a} \cdot \vec{q_w} = f_q(p)$. It follows that the problem has a unique stable matching, and that it can be found with the first-choice chain algorithm (Algorithm 11).

It remains to provide an adequate first-choice data structure. In our case, the first-choice data structure should maintain a set of vectors, and, given a query vector, return the vector maximizing the dot product with the query vector. Under a dual transformation, such queries become ray shooting queries: each vector becomes a hyperplane, and a query asks for the first hyperplane hit by a vertical ray from the query point [132]. We use the data structure from Matoušek and Schwarzkopf [144], the runtime of which is captured in the following lemma (see [2] for a summary of ray-shooting data structures).

Lemma 4.8. ([144, Theorem 1.5]) *Let $\varepsilon > 0$ be a constant, $k \geq 4$ a fixed dimension, and m a parameter with $n \leq m \leq n^{\lfloor k/2 \rfloor}$. Then, there is a dynamic data structure for ray-shooting queries requiring $O(m^{1+\varepsilon})$ space and preprocessing time, $O(m^{1+\varepsilon}/n)$ insertion and deletion time, and $O(\frac{n}{m^{1/\lfloor k/2 \rfloor}} \log n)$ query time.*

Theorem 4.9. *For any $\varepsilon > 0$, the narcissistic k-attribute stable matching problem can be solved in $O(n \log n)$ time for $k = 2$, $O(n^{4/3+\varepsilon})$ time for $k = 3$, and $O(n^{2-4/(k(1+\varepsilon)+2)})$ time for $k \geq 4$.*

[12] The $\tilde{O}$ notation omits polylogarithmic factors.

Proof. Since preferences are symmetric, the problem can be solved in $O(P(n) + nT(n))$ time, given a dynamic data structure for ray-shooting queries with $P(n)$ preprocessing time and $T(n)$ operation time (Theorem 4.7).

For $k \geq 4$, using the data structure for ray-shooting queries cited in Lemma 4.8 results in a runtime of $O(m^{1+\varepsilon} + \frac{n^2 \log n}{m^{1/\lfloor k/2 \rfloor}})$ for any $\varepsilon > 0$. The optimal runtime is achieved when the parameter m is chosen to balance the two terms, i.e., so that $m^{1+\varepsilon} = \frac{n^2 \log n}{m^{1/\lfloor k/2 \rfloor}}$. This gives $m = (n^2 \log n)^{1/(1+\varepsilon+1/\lfloor k/2 \rfloor)}$. For the sake of obtaining a simple asymptotic expression, we set m to $(n^2 \log n)^{1/(1+\varepsilon+2/k)}$ (which is the same for even k, and bigger for odd k). Then, the $O(m^{1+\varepsilon})$ term dominates. Also note that if $\varepsilon < 1 - 2/k$, this value of m is between n and $n^{\lfloor k/2 \rfloor}$, so the condition in Lemma 4.8 is satisfied.

Thus, the problem can be solved in $O(m^{1+\varepsilon}) = O((n^2 \log n)^{(1+\varepsilon)/(1+\varepsilon+2/k)})$ time, which further simplifies to the claimed runtime of $O(n^{2-4/(k(1+\varepsilon')+2)})$ (where ε' needs to satisfy $\varepsilon' > \varepsilon$). For $k = 3$, we use the same data structure, but raising the problem to four dimensions, so that Lemma 4.8 applies. For $k = 2$, see Lemma 4.12. □

4.5.1 The 2-attribute case

In the special case where $k = 2$, we can design a simple first-choice data structure with $P(n) = O(n \log n)$ preprocessing time and $T(n) = O(\log n)$ query and deletion time. Note that, for a vector $\vec{p}$ in $\mathbb{R}^2$, all the points along a line perpendicular to $\vec{p}$ are equally preferred, i.e., have the same dot product with $\vec{p}$ (because their projections onto the supporting line of $\vec{p}$ are the same). In fact, the preference list for $\vec{p}$ corresponds to the order in which a line perpendicular to $\vec{p}$ encounters the vectors in the other set as it moves in the direction opposite from $\vec{p}$ (see Figure 4.1, left). We get the following lemma (where the vectors in one set are interpreted as points).

Lemma 4.10. *Given a point set P and a vector $\vec{q}$, in $\mathbb{R}^2$, the point p^* in P maximizing $\vec{q} \cdot p^*$ is in the convex hull of P.*

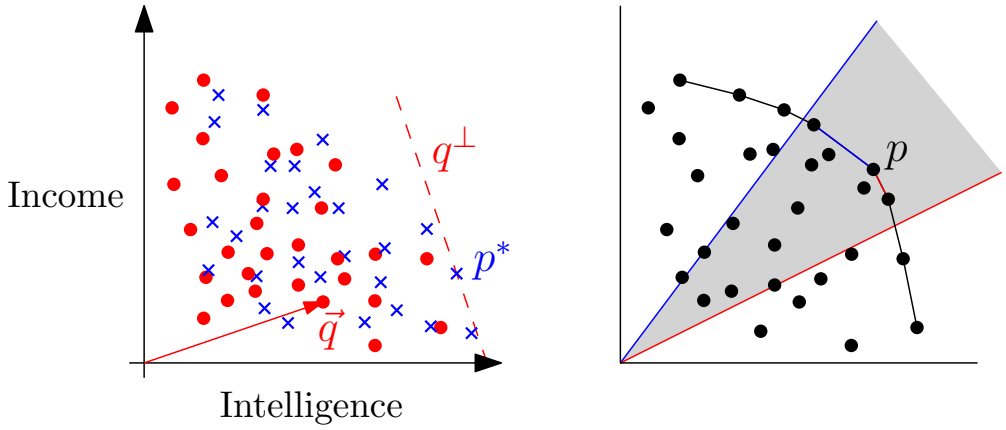


Figure 4.1: **Left:** an instance of narcissistic 2-attribute stable matching. The two sets of vectors are represented as red dots and blue crosses, respectively, in a plane where the axes correspond to the two attributes. For a specific red vector, $\vec{q}$, its first choice in the other set (the vector maximizing the dot product), p^*, is shown. The dashed line passing through p^* is perpendicular to $\vec{q}$. **Right:** the point p is the point among the black points maximizing $q \cdot p$ for all the points q in the gray wedge. The wedge is delimited by two rays starting at the origin and perpendicular to the two edges of the convex hull incident to p.

Proof. Consider a line perpendicular to $\vec{q}$. Move this line in the direction of $\vec{q}$, until all points in P lie on the same side of it (behind it). Note that any line orthogonal to $\vec{q}$ has the property that all points lying on the line have the same dot product with $\vec{q}$. The point p^* is the last point in P to touch the line, since moving the line in the opposite direction from $\vec{q}$ decreases the dot product of $\vec{q}$ with any point on the line (and by the general position assumption, it is unique). Clearly, p^* is in the convex hull. □

Our first-choice data structure is a semi-dynamic convex hull data structure, where deletions are allowed but not insertions [111]. We handle queries as in Lemma 4.11.

Lemma 4.11. *Given the ordered list of points along the convex hull of a point set P, and a query vector $\vec{q}$, we can find the point p^* in P maximizing $\vec{q} \cdot p^*$ in $O(\log n)$ time, where n is the number of points in the convex hull.*

Proof. By Lemma 4.10, the point p^* is in the convex hull. For ease of exposition, assume that all the points in P and $\vec{q}$ have positive coordinates (the alternative cases are similar). Then, p^* lies in

the top-right section of the convex hull (the section from the highest point to the rightmost point, in clockwise order).

Note that points along the top-right convex hull are ordered by their y-coordinate, so, we say *above* and *below* to describe the relative positions of points in it. Each point p in the top-right convex hull is the point in P maximizing $p \cdot \vec{q'}$ for all the vectors $\vec{q'}$ in an infinite wedge, as depicted in Figure 4.1, right. The wedge contains all the vectors $\vec{q'}$ whose perpendicular line touches p last when moving in the direction of $\vec{q'}$, so the edges of the wedge are perpendicular to the edges of the convex hull incident to p. Thus, by looking at the neighbors of p along the convex hull, we can calculate this wedge and know whether $\vec{q}$ is in the wedge for p, below it, or above it. Based on this, we discern whether the first choice of $\vec{q}$ is p itself or above or below it. Hence, we can do binary search for p^* in $O(\log n)$ time. □

Lemma 4.12. *The narcissistic 2-attribute stable matching problem can be solved in $O(n \log n)$ time.*

Proof. We can use the first-choice chain algorithm (Algorithm 11) coupled with a first-choice data structure which is a semi-dynamic convex hull data structure. Updating the convex-hull can be done in $O(n \log n)$ time throughout the algorithm [111]. Queries are answered in $O(\log n)$ time (Lemma 4.11). Thus, the total running time is $O(n \log n)$. □

This result illustrates how fully-dynamic data structures are not needed for NNC-based algorithms.

4.6 The stable roommates problem

The *stable roommates problem* [12, 117, 118] is a variant of stable matching where the agents are not split in two sets and anyone can be matched to anyone. Symmetric preferences and soul mates can be defined analogously, and the soul-mate algorithm also works for the stable roommates

problem when preferences are symmetric. The only change in the algorithm is that all the agents are put in a single first-choice data structure. With arbitrary preferences, an instance of the stable roommates problem may have no solution [117], but with symmetric preferences there is always a unique solution.

Our algorithmic results for the symmetric models that we considered also apply to this setting. In the geometric model, we can even improve the runtime from $O(n \log^4 n)$ to $O(n \log n)$. This is because when the agents are points in space and preferences are from closest to farthest, the unique stable solution can be found by repeatedly matching the closest pair. This coincides with the greedy algorithm for geometric matching (Problem 11). In Section 3.8, we showed how to find this matching in $O(n \log n)$ time using a closest pair data structure or the soft nearest-neighbor chain algorithm (Algorithm 8).

4.7 Conclusions

We have studied three stable-matching models: the geometric, geographic, and narcissistic k-attribute models. These models have in common that they have symmetric preferences. We have shown how symmetric preferences have special properties. First, with symmetric preferences there exists a unique stable matching. Further, it can be found with a greedy algorithm, and we have global–local equivalence. In turn, this allows us to use the nearest-neighbor chain algorithm to find the stable matching efficiently.

Are there other natural stable matching models with symmetric preferences? Here is a suggestion for a model: in a matching market, all the agents are given a set of options, and they are asked to rank the options (not the agents) by importance. The goal is to match agents with similar priorities. This can be quantified by counting the number of inversions (pairs of options that are in different order) between the orderings of two agents. This model is symmetric because the number

of inversions is the same from the perspective of both agents. It is also narcissistic because each agent has zero inversions with its own list.

From the models studied so far, it seems that symmetric and narcissistic preferences always go hand in hand. To see that this is not the case, we suggest another model. Consider that each agent is identified with a region in space. For instance, this region could denote the territory where they have a license to operate. We consider three options.

1. Each agent x gives a score of $area(x \cap y)$ to each agent y (i.e., the area of the region where they are allowed to do business together). This model is symmetric and narcissistic (each agent gives the highest possible score to itself).

2. Each agent x gives a score of $area(x \cup y)$ to each agent y (i.e., how much territory they reach together). This model is symmetric but not narcissistic (each agent gives the lowest possible score to itself).

3. Each agent x gives a score of $area(y \setminus x)$ to each agent y (i.e., how much new territory x can reach as a result of partnering with y). The scores are not symmetric, and the model is not narcissistic. Nonetheless, there is a unique stable matching (assuming no ties in the scores). This is because the preference lists corresponding to this model are precisely the same as for the second model: $area(y \setminus x) = area(x \cup y) - area(x)$, and subtracting a constant ($area(x)$) to every score does not change the preference list of x.

The stable roommates problem in the geometric setting can be solved in $O(n \log n)$ time, while for geometric stable matching we only know how to solve it in $O(n \log^4 n)$ time. We leave it as an open problem to close the gap between the runtimes of these two related problems. The reason for this gap is that we cannot use the soft nearest-neighbor chain algorithm (Algorithm 8) when the agents are split in two sets. This is because if we use a soft nearest-neighbor data structure for the colleges and one for the students, a soft answer from one of those structures, e.g., the one containing the

students, would give us two students. This does not help us in making progress towards a locally-dominant pair, which must consist of a student and a college. We leave it as an open problem to design a bichromatic variant of the soft nearest neighbor data structure (Definition 3.9). This structure should maintain two dynamic point sets, A and B, and, given a query point from one of the sets, either return its nearest neighbor of the opposite set or a pair of points of different sets closer to each other than the query point to its nearest neighbor of the other set. Is it possible to match the $O(\log n)$ time per operation of the monochromatic case (Theorem 3.10)?

Chapter 5

Proximity Data Structures In Graphs

5.1 Introduction

Proximity data structures maintain a set of objects of interest, called *sites*, and support queries concerned with minimizing some distance involving the sites, such as nearest-neighbor or closest-pair queries. They are well known in computational geometry [63], where sites are points in space and distance is measured by Euclidean distance or some other metric (e.g., see Section 3.1). In this chapter, we are interested in proximity data structures that deal with nodes in a graph rather than points in space. We consider that there is an underlying, fixed graph G, such as a road network for a geographic region, and sites are a distinguished subset P of the vertices of G. Distance is measured by shortest-path distance in G. We consider updates (additions and deletions) to and from the set P of sites. Our goal is to design efficient data structures for the following problems.

Definition 5.1 (Reactive nearest-neighbor data structure in graphs)**.** Given a fixed, undirected graph $G = (V, E)$ with positively-weighted edges, maintain a subset of nodes $P \subseteq V$, subject to insertions to P, deletions from P, and nearest-neighbor queries: given a query node $q \in V$, return the node $p \in P$ closest to q in shortest-path distance.

Definition 5.2 (Reactive closest-pair data structure in graphs)**.** Given a fixed, undirected graph $G = (V, E)$ with positively-weighted edges, maintain a subset of nodes $P \subseteq V$, subject to insertions to P, deletions from P, and queries asking for the closest pair in P.

Definition 5.3 (Reactive bichromatic closest-pair data structure in graphs)**.** Given a fixed, undirected graph $G = (V, E)$ with positively-weighted edges, maintain two subsets of nodes $P, Q \subseteq V$, subject to insertions to P or Q, deletions from P or Q, and queries asking for the closest pair of nodes in different sets (one in P and one in Q).

5.1.1 Background

The data structures that we study fall into the area of *dynamic graph algorithms*, the subject of extensive study [83]. Traditionally, dynamic data structures in graphs, e.g., for shortest-path computations, allow updates on the underlying graph G itself, such as vertex or edge insertions and deletions [41, 44, 69, 71, 83, 130, 168]. We call our data structures *reactive* to distinguish the kind of updates we allow. For us, G is fixed, but we allow updates on P.

Previous work on dynamic graph algorithms has focused on the setting where G can change. Exceptions are the work of Eppstein on maintaining a dynamic subset of vertices in a sparse graph and keeping track of whether it is a dominating set [80], and the work of Italiano and Frigioni on dynamic connectivity for subsets of vertices in a planar graph [96]. Despite the applications that we mention in Section 5.1.3, to our knowledge, no one has considered proximity data structures for graphs subject to updates on the set of sites.

We design data structures that only work for graphs from certain hereditary graph classes. A *graph class* is a (generally infinite) set of all graphs with some defining property. A graph class is *hereditary* if it contains every induced subgraph of every graph in the class. An *induced subgraph* of a graph G is a graph obtained by removing any number of vertices (and their incident edges)

from G. For instance, the class of planar graphs is hereditary because every induced subgraph of a planar graph is planar.

Our data structures work for graphs from hereditary graph classes with separators of sublinear size. A *separator* of a graph $G = (V, E)$ is a subset of V whose removal from G splits G into two disjoint subgraphs, each with at most $\frac{2}{3}|V|$ nodes, and with no edges between them. We say a graph class has $O(n^c)$-size separators if every n-node graph in the class has a separator of size $O(n^c)$. A graph class has sublinear separators if it has $O(n^c)$-size separators for some $c < 1$. For instance, the planar separator theorem states that planar graphs have $O(n^{0.5})$-size separators [140].

A hereditary graph class has sublinear separators if and only if it has polynomial expansion [74]. Thus, any graph from a class of polynomial expansion is suitable for our data structures.

If $G = (V, E)$ is a graph from a hereditary graph class with sublinear separators, then G is *sparse*, which means that $|E| = O(|V|)$. The converse is not necessarily true. For instance, bounded-degree expander graphs are sparse but do not have sublinear separators [114]. Nonetheless, many important sparse graph families are hereditary and have sublinear separators. One of the first classes that was shown to have sublinear separators is the class of planar graphs, which have $O(n^{0.5})$-size separators [140]. Separators of the same asymptotic size have also been proven to exist for k-planar graphs [73], bounded-genus graphs [100], minor-closed graph families [126], and the graphs of certain four-dimensional polyhedra [81]. In addition, trees have separators of size one. More generally, graphs with bounded treewidth [167] have constant-size separators [55].

The importance of having sublinear separators in our data structures is that it allows us to construct a *separator hierarchy*. A separator hierarchy is the result of recursively partitioning a graph into disjoint subgraphs using separators. Separator hierarchies are useful to solve many graph problems [95, 102]. An important application is the *single-source shortest path* (SSSP) problem: finding the distance from a node to every other node in the graph. This problem can be solved in linear time given a type of separator hierarchy called a *recursive division* [110]. For graphs for

which we can construct this hierarchy in linear time, such as planar graphs [110], the SSSP problem can be solved in linear time. This improves upon the $O(n \log n)$ time required by Dijkstra's algorithm in sparse graphs [70].

We emphasize that the graph class must be hereditary because, otherwise, a graph G could have a small separator but not a separator hierarchy. Consider, for instance, that G consists of two equal-sized cliques plus a node connected to a node on each clique. This graph has a separator of size 1, but we cannot build a separator hierarchy for G because cliques do not have sublinear separators.

In many applications (see Section 5.1.3), the underlying graph G represents a real road network. A road network can be represented by a graph where each node is an intersection, and each edge is a stretch of road connecting two intersections. Edge weights represent road lengths. Road networks are often modeled as planar graphs. However, they are not quite planar because of bridges and underpasses [84]. Thus, we are particularly interested in a class of graphs which has been shown to be a better model for road networks: the class of *graphs with sparse crossing graphs* [87]. Given an embedding of a graph G in the plane, the *crossing graph* of the embedding is a graph H where each node in H represents an edge of G, and two nodes in H are connected if the corresponding edges in G cross in the embedding. Clearly, a graph is planar if it has an embedding such that the corresponding crossing graph has no edges. More generally, it is k-planar if it has an embedding such that the crossing graph has maximum degree k. Graphs with sparse crossing graphs further generalize k-planar graphs: a graph has a sparse crossing graph if it has an embedding such that the corresponding crossing graph has bounded *degeneracy*, a notion of sparcity used in graph theory [139]. Bounding the degeneracy of the crossing graph instead of the maximum degree accounts for, e.g., long tunnels that go under many street-level roads. Like planar graphs, the class of graphs with sparse crossing graphs is also hereditary and has $O(n^{0.5})$-size separators [87]. This is fortunate, because it means that we can use our data structures in applications dealing with road networks.

5.1.2 Our contributions

We design a new reactive nearest-neighbor data structure (Definition 5.1) with the aim to balance between query and update times. If we only cared about one of these, the data structure would be trivial. For instance, if we only cared about query time, there is a well known solution: the graph-based Voronoi diagram, which maintains the closest site to each node in the graph. Erwig [88] shows that Voronoi diagrams can be adapted to graphs and that they can be constructed using a modification of Dijkstra's algorithm. With this information, queries can be answered in constant time. However, the Voronoi diagram is not easy to update, requiring $O(n \log n)$ time in sparse graphs with n nodes—the same time as for creating the diagram from scratch.

If, instead, we optimize for update time only, we could avoid maintaining any information and answer queries directly using a shortest-path algorithm from the query node. Updates would take constant time; queries could be answered using Dijkstra's algorithm [61], which runs in $O(n \log n)$ time in sparse graphs. As mentioned, this could be improved to $O(n)$ time for graphs for which we can construct a recursive subdivision during a preprocessing stage [110].

Our reactive nearest-neighbor data structure finds a "sweet spot" between fast queries and fast updates. Table 5.1 summarizes its runtime as a function of the size of the separators (the data structure is the same when $c = 0$, but an extra logarithmic factor appears in the analysis). For planar graphs and, more generally, graphs with sparse crossing graphs, $c = 1/2$. For graphs with bounded treewidth, $c = 0$.

To construct a reactive nearest-neighbor data structure for planar graphs specifically, we could also consider using an *exact-distance oracle*. This is a static data structure that admits queries asking for the distance between any two nodes. If k is the number of sites, with an exact-distance oracle we can find the closest site to a query node with k queries. The recent oracle from Gawrychowski et al. [98] answers queries in $O(\log n)$ time when it uses $O(n^{1.5})$ space like our data structure. With this oracle, we could answer queries for our data structure in $O(k \log n)$ time and do updates in

Sep. size	Space	Preprocessing	Query	Insertion	Deletion
$0 < c < 1$	$O(n^{1+c})$	$O(n^{1+c})$	$O(n^c)$	$O(n^c)$	$O(n^c \log k)$
	$O(n^{1+c})$	$O(n^{1+c} \log n)$	$O(n^c)$	$O(n^c \log\log n)$	$O(n^c \log\log n)$
$c = 0$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$	$O(\log n)$	$O(\log n \log k)$
	$O(n \log n)$	$O(n \log^2 n)$	$O(\log n)$	$O(\log n \log\log n)$	$O(\log n \log\log n)$

Table 5.1: Runtimes of our reactive nearest-neighbor data structure when it maintains k sites on an n-node graph from a hereditary graph class with $O(n^c)$-size separators. The preprocessing time is under the assumption that a separator can be found in $O(n^{1+c})$ time. Possible trade-offs between preprocessing and update times are shown.

Data structure	Query	Insertion	Deletion
Exact NN	$O(n^c)$	$O(n^c)$	$O(n^c \log k)$
Closest pair	$O(1)$	$O(n^c \log^2 k)^\dagger$	$O(n^c \log^3 k)^\dagger$
Bichromatic CP	$O(1)$	$O(n^c \log^2 k)^\dagger$	$O(n^c \log^3 k)^\dagger$

Table 5.2: Runtimes of our reactive proximity data structures when they maintain k sites on an n-node graph from a hereditary graph class with $O(n^c)$-size separators, where $0 < c < 1$ (we omit the case of $c = 0$ for brevity). All the data structures require $O(n^{1+c})$ space. The preprocessing time is $O(n^{1+c})$ assuming that a separator can be found in $O(n^{1+c})$ time. The "†" superindex indicates that the runtime is amortized.

constant time. This approach has better runtimes when k is small, but the preprocessing time is $O(n^2)$.

We combine our reactive nearest-neighbor data structure with preexisting data structures [78,79] to obtain other proximity data structures. Table 5.2 shows our new family of proximity data structures. Each data structure has a similar preprocessing–update time trade-off as shown in Table 5.1. For brevity, we only show the versions of the data structures that optimize the preprocessing time.

5.1.3 Applications

We use our reactive nearest-neighbor data structure as part of algorithms for Steiner TSP (Section 3.4), geographic stable matching (Section 4.4), and political redistricting (Section 7.2)).

Reactive proximity data structures in graphs can also be useful in several logistical problems in geographic information systems dealing with real-time data. Consider, for instance, an application

to connect drivers and clients in a private-driver service, such as Uber or Lyft, or even a future self-driving car service. A reactive nearest-neighbor data structure could maintain the set of cars waiting at various locations in a city to be put into service. When a client requires a driver, she queries the data structure to find the car nearest to her. This car is then removed from P (i.e., it is no longer available) until it completes the trip for this client, at which point the car is then added to P (i.e., it is available) at this new location. Alternatively, we could consider a similar application in the context of police or emergency dispatching, where the data structure maintains the locations of a set of available first responder vehicles. In Section 5.4, we experiment with this type of system emulating random queries in a real road network.

Reactive proximity data structures can also be useful in other domains, such as content distribution networks, like the one maintained by Akamai. For instance, a reactive nearest-neighbor data structure could maintain the set of nodes that contain a certain file of interest, like a movie. When another node in the network needs this information, the data structure could be used to find the closest node that can transfer it. Updates allow us to model how copies of such a file migrate in the network, e.g., for load balancing, so that we add a node to P when it gets a copy of the file and remove a node from P when it passes the file to another server.

5.2 Nearest-neighbor data structure

Initially, we are given an n-node graph $G = (V, E)$ and a subset $P \subseteq V$ of sites. As mentioned, the runtime analysis depends on the size of the separators. Henceforth, we consider that G is undirected, has positive edge weights, and comes from a hereditary graph class with $O(n^c)$-size separators for some constant c with $0 < c < 1$ (the analysis is slightly different when $c = 0$).

We begin by reviewing the concept of a separator hierarchy. Recall that a *separator* in a given n-vertex graph is a subset S of nodes such that the removal of S (and its incident edges) partitions the

remaining graph into two disjoint subgraphs (with no edges from one to the other), each of size at most $2n/3$. It is allowed for these subgraphs to be disconnected; that is, removing S can partition the remaining graph into more than two connected components, as long as those components can be grouped into two subgraphs that are each of size at most $2n/3$. A *separator hierarchy* is the result of recursively subdividing a graph by using separators. Since children have size at most $2/3$ the size of the parent, the separator hierarchy is a binary tree of $O(\log n)$ height.

5.2.1 Preprocessing

The creation of our data structure consists of two phases. The first phase does not depend on P, while the second phase incorporates our knowledge of P. Note that there are two kinds of nodes of interest: separator nodes and sites. The two sets may intersect, but should not be confused.

Site-independent phase. First, we build a *separator hierarchy* of the graph. This hierarchy can be constructed in $O(n)$ time and space in planar graphs [102] and graphs with sparse crossing graphs [87]. However, we do not need the construction to take linear time, as this is not the bottleneck of the preprocessing. It suffices that the hierarchy can be computed in $O(n^{1+c})$ time. In fact, it suffices that a single separator can be found in $O(n^{1+c})$ time in an n-node graph (as opposed to the entire hierarchy). This is because the hierarchy is built recursively so, if a separator can be found in $O(n^{1+c})$ time, the construction time of the entire hierarchy is captured by the recurrence

$$T(n) \leq T(x) + T(y) + O(n^{1+c}), \tag{5.1}$$

where x and y are the sizes of two subgraphs, chosen so that $x + y \leq n$, $\max(x, y) \leq 2n/3$, and, among all x and y obeying these constraints, $T(x) + T(y)$ is maximum. It is easy to prove that this recurrence is dominated by its top-level $O(n^{1+c})$ term, so $T(n) = O(n^{1+c})$.

Second, we compute, for each graph in the hierarchy, the distance from each separator node to every other node. Consider the work done for the graph at the root of the hierarchy, G itself. We need to compute $O(n^c)$ SSSP problems, one for each separator node. As mentioned, each such problem can be solved in linear time given a recursive subdivision [110]. A recursive subdivision is a type of separator hierarchy that is also built by finding separators recursively. Thus, if we can find a separator in $O(n^{1+c})$ time, we can construct the entire recursive subdivision, and compute all the distances for the separators in the top-level graph, in $O(n^{1+c})$ time. We do the same for all the remaining graphs in the separator hierarchy. The total runtime follows Equation 5.1 again, so it is also $O(n^{1+c})$.

Site-dependent phase. For each graph $H = (V_H, E_H)$ in the separator hierarchy, for each separator node s in H, we initialize a priority queue Q_s. The elements stored in Q_s are the sites in H, $P \cap V_H$. Their priorities are their distances from s in H.

We use an implementation of a priority queue that supports insertions and find-minimum operations in constant worst-case time, and deletions in logarithmic worst-case time. For instance, we can use a strict Fibonacci heap [37] or a Broadal queue [36]. Then, constructing each queue Q_s takes time linear on the number of sites in H. Thus, the time at the top level of the hierarchy is $O(|P|)$ per separator node, and $|P| = O(n)$, so in total it is $O(n^{1+c})$. The total time analysis of this phase is $O(n^{1+c})$ as before.

Adding the space and time for the two phases together gives $O(n^{1+c})$ space and preprocessing time for graphs for which we can find a separator in $O(n^{1+c})$ time.

5.2.2 Queries

Given a query node q, we find two sites: (a) the closest site to q with paths restricted to the same side of the top-level partition as q, and (b) the closest site to q with paths containing at least one

separator node. The paths considered between both cases cover all possible paths, so one of the two found sites is the overall closest site to q.

- To find the site satisfying Condition (a), we can relay the query to the subgraph of the separator hierarchy containing q. This case does not arise if q is a separator node.
- To find the site satisfying Condition (b), we need to find the shortest path from q to any site, but only among paths containing separator nodes. Note that if the shortest path goes through a separator s, it should end at the site closest to s. Therefore, the length of the shortest path starting at q, going through s, and ending at any site, is $d(q, s) + d(s, \min(Q_s))$, where $\min(Q_s)$ denotes the element with the smallest key in Q_s. We can find the site satisfying Condition (b) by considering all the separator nodes and retaining the one minimizing this sum.

The time to find the site satisfying Condition (b) is $O(n^c)$, since there are $O(n^c)$ separator nodes to check and we do a find-minimum operation on a priority queue for each. We do not need to do any distance computation, as we precomputed all the needed distances. Therefore, the time to find the two paths satisfying Conditions (a) and (b) can be analyzed by the recurrence

$$T(n) \leq T(2n/3) + O(n^c),$$

where the $T(2n/3)$ term dominates the actual time for recursing in a single subgraph of the separator hierarchy. The solution to this recurrence is $O(n^c)$, so queries take $O(n^c)$ time.

We can implement a heuristic optimization for queries so that we do not need to check every separator node when searching for the site satisfying Condition (b). During the preprocessing stage, we can sort, for each node u in each graph H of the separator hierarchy, all the separators in H by distance from u. This increases the space used by the data structure by a constant factor. Then, during a query, after obtaining the site satisfying Condition (a), to find the site satisfying

Condition (b), we consider the separator nodes in order by distance from the query node q. Suppose that p is the closest site found so far. As soon as we reach a separator node s such that $d(q, s) \geq d(q, p)$, we can stop and ignore the rest of separator nodes, since any site reached through them would be further from q than p. In our experiments (Section 5.4), this optimization reduces the average query runtime by a factor between 1.5 and 9.5, depending on the number of sites. It is more effective when there are many sites, as then the closest site is likely to be closer than many separators at the upper levels of the hierarchy.

5.2.3 Updates

Suppose that we wish to insert or delete a node p to or from the set of sites P. Note that, when we perform such an update, the structures computed during the site-independent preprocessing phase (the separator hierarchy and the computation of distances) do not change. However, we need to add or remove p (according to the type of update) to or from the priority queue Q_s for every separator node s in the top-level separator. Moreover, if p is not a separator node, we also need to update the priority queues for the subgraph containing p, recursively.

The time for an insertion is the same as for a query, since our priority queues support constant time insertions. For deletions, the time to remove p in all top-level priority queues is $O(\log k)$ time per priority queue, where k is the number of sites, for a total time of $O(n^c \log k)$. Again, if we formulate and solve a recurrence for the running time at all levels of the separator hierarchy, these times are dominated by the top-level term.

Next, we discuss how to improve the update time to $O(n^c \log \log n)$ with additional preprocessing. For each separator node s, instead of using the distance from s to p as the key for a site p in the priority queue Q_s, we can use the index of p in the list of nodes sorted by distance from s. That is, if the set of distances in sorted order from s to the other nodes are $d_1, d_2, d_3, \ldots$, with $d_1 < d_2 < d_3 < \cdots$, we could replace these numbers by the numbers $1, 2, 3, \ldots$, without

changing the comparison between any two distances. This replacement would allow us to use a faster integer priority queue in place of the priority queue. For instance, a van Emde Boas tree [181] maintains the minimum in a set of integer numbers between 1 and n in $O(\log \log n)$ time per insertion and deletion. In order to use this optimization, we need to add the time to sort the distances in the preprocessing time, which increases to $O(n^{1+c} \log n)$ (assuming an $O(n \log n)$ time sorting algorithm is used).

We have completed the description and analysis of the data structure. Theorem 5.4 captures its runtime.

Theorem 5.4. *Let c be a constant with $0 < c < 1$, let $\mathcal{G}$ be a hereditary graph class with $O(n^c)$-size separators, and let $T(n)$ be the time needed to find a separator in an n-node graph from $\mathcal{G}$. Then, for any n-node graph from $\mathcal{G}$, there is a reactive nearest-neighbor data structure that uses $O(n^{1+c})$ space, with $\max(O(n^{1+c}), T(n))$ preprocessing time, $O(n^c)$ query and insertion time, and $O(n^c \log k)$ deletion time, where k is the number of sites. Alternatively, the data structure could have $\max(O(n^{1+c} \log n), T(n))$ preprocessing time, $O(n^c \log \log n)$ insertion and deletion time, and the same space and query time.*

5.3 Extensions and related data structures

If we reformulate and solve the recurrence equations for the case where there is constant number of separator nodes ($c = 0$), we obtain the space and runtimes shown in Table 5.1.

The conga-line data structure [79] is a closest-pair data structure with $O(1)$ query time, $O(T(k) \log k)$ amortized insertion time, and $O(T(k) \log^2 k)$ amortized deletion time, where $T(k)$ is the time per operation (maximum between query and update) of a nearest-neighbor data structure maintaining k sites. Another data structure [78] achieves the same runtimes, but for the bichromatic closest-pair problem. Combined with our reactive nearest-neighbor data structure, we get the following result.

Lemma 5.5. *Let c be a constant with $0 < c < 1$, and let $\mathcal{G}$ be a hereditary graph class with $O(n^c)$-size separators. For any n-node graph from $\mathcal{G}$, there is a reactive closest-pair data structure and a reactive bichromatic closest-pair data structure with the space and runtimes shown in Table 5.2.*

Finally, our reactive nearest neighbor data structure can be extended to directed graphs with the same asymptotic runtimes. The only required change is to compute distances *from* and *to* every separator node. To obtain the latter, we can compute the distances in the *reverse graph*, i.e., the graph obtained by reversing the directions of all the edges.

5.4 Experiments

In this section, we evaluate our data structure empirically on a real road network, the Delaware road network from the DIMACS data set [68]. We consider the biggest connected component of the network, which has 48812 nodes and 60027 edges. This data set has been planarized: overpasses and underpasses have been replaced by artificial intersection nodes. Each trial in our experiment begins with a number of uniformly distributed random sites, and then performs 1000 operations. We consider the cases of only queries, only updates, and a mixture of both (see Figure 5.1). The updates alternate between insertions and deletions, and the operations in the mixed case alternate between queries and updates. We compare the performance of our data structure against a basic data structure that simply uses Dijkstra's algorithm for the queries.

5.4.1 Implementation details

We implemented the algorithms in Java 8.[13] We then executed them and timed them as run on an Intel(R) Core(TM) CPU i7-3537U 2.00GHz with 4GB of RAM, on Windows 10.

[13]The source code is available at `https://github.com/nmamano/NearestNeighborInGraphs`.

We implemented the optimization for queries described in Section 5.2.2, and compared it with the unoptimized version in order to evaluate if its worth the extra space. For updates, we used a normal binary heap, as these tend to perform better in practice than more sophisticated data structures.

A factor that affects the efficiency of the data structure is the size and balance of the separators. Our hierarchy for the Delaware road network had a total of 504639 nodes across 8960 graphs up to 13 levels deep. Among these graphs, the biggest separator had 81 nodes. Rather than implementing a full planar separator algorithm to find the separators (recall that the data had been planarized), we choose the smallest of two simply-determined separators: the vertical and horizontal lines partitioning the nodes into two equal subsets. While these are not guaranteed to have size $O(\sqrt{n})$, past experiments on the transversal complexity in road networks [86] indicate that straight-line traversals of road networks should provide separators with low complexity, making it unnecessary to incorporate a full planar graph separator algorithm.

When a separator partitions a graph in more than two connected components, we made one child per component. Thus, our hierarchy is not necessarily a binary tree, and may be shallower. We set the base case size to 20. At the base case, we perform Dijkstra's algorithm. Experiments with different base-case sizes did not affect the performance significantly.

5.4.2 Results

Figure 5.1 depicts the results. Table 5.3 shows the corresponding data for the case of mixed operations, which is the case of interest in a reactive model.

- The runtime of Dijkstra's algorithm is roughly inversely proportional to the number of sites, because with more sites it requires less exploration to find the closest one. Moreover, initialization and updates require virtually no time. Thus, this choice is superior for large numbers

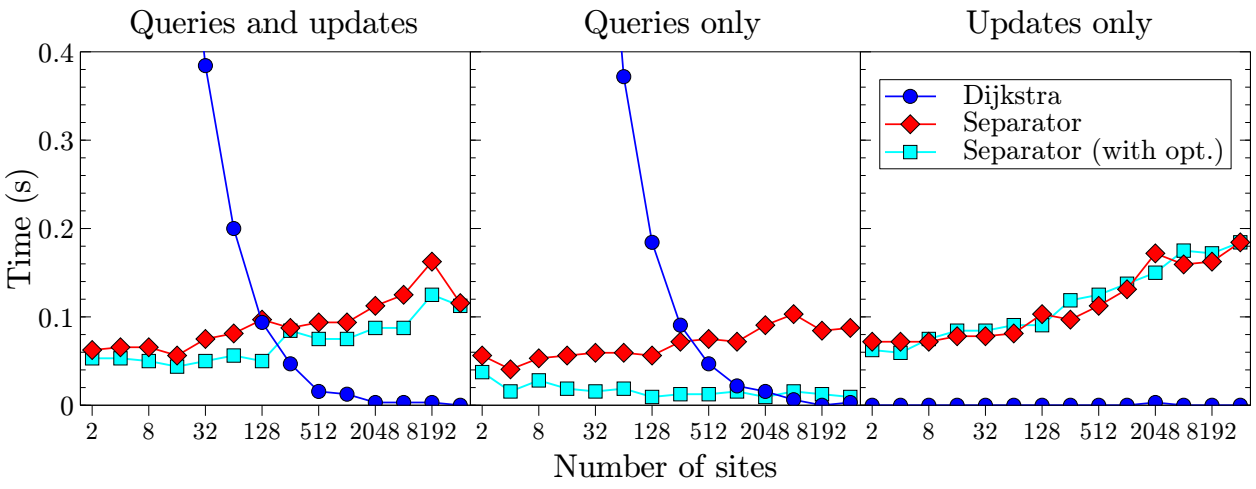


Figure 5.1: Time needed by the data structures to complete 1000 operations in the Delaware road network [68] for a range of number of sites (in a logarithmic scale), excluding preprocessing time. Each data point is the average of 5 runs with different sets of random sites (the same sets for all the algorithms).

of sites, while being orders of magnitude slower when the number of sites is low (see Table 5.3).

- Our data structure based on a separator hierarchy is not affected as much by the number of sites. The update runtime only increases slightly with more sites because of the operations on larger heaps, as expected from its asymptotic runtime. The optimization, which reduces the number of separators needed to be checked, can be seen to have a significant effect on queries, especially as the number of sites increases: it is up to 9.5 times faster on average with 2048 sites. However, since it has no effect on updates, in the mixed model with the same number of updates and queries the improvement is less significant.

- The data structure requires a significant amount of time to construct the hierarchy. Our code constructed the hierarchy for the Delaware road network in around 15 seconds. Fortunately, this hierarchy only needs to be built once per road network. The limiting factor is the space requirement of approximately $O(n^{1+5})$, which caused us to run out of memory for other road networks from the DIMACS data set with over 10^5 nodes.

# sites	Dijkstra	Separator	Separator (with opt.)
2	3797 (3672 – 3906)	63 (47 – 94)	53 (31 – 94)
4	2303 (2203 – 2359)	66 (63 – 78)	53 (47 – 63)
8	1272 (1250 – 1297)	66 (47 – 78)	50 (47 – 63)
16	694 (641 – 781)	56 (47 – 63)	44 (31 – 47)
32	384 (359 – 406)	75 (63 – 94)	50 (47 – 63)
64	200 (172 – 219)	81 (63 – 94)	56 (47 – 63)
128	94 (94 – 94)	97 (94 – 109)	50 (47 – 63)
256	47 (47 – 47)	88 (78 – 94)	84 (78 – 109)
512	16 (16 – 16)	94 (94 – 94)	75 (63 – 78)
1024	13 (0 – 16)	94 (94 – 94)	75 (63 – 78)
2048	3 (0 – 16)	113 (94 – 125)	88 (78 – 94)
4096	3 (0 – 16)	125 (109 – 156)	88 (78 – 94)
8192	3 (0 – 16)	163 (125 – 188)	125 (94 – 156)
16384	0 (0 – 0)	116 (109 – 125)	113 (94 – 156)

Table 5.3: Time in milliseconds needed by the data structures to complete 1000 operations (mixed queries and updates) in the Delaware road network for a range of number of sites (in a logarithmic scale). Each data point is the average, minimum, and maximum, of 5 runs with different sets of random sites (the same sets for all the algorithms).

5.5 Conclusions

We have studied reactive proximity problems in graphs, giving a family of data structures for such problems. Tables 5.1 and 5.2 summarize our theoretical results. While we have focused on applications in geographic systems dealing with real-time data, the problems are primitive enough that they may arise in other domains of graph theory, such as network protocols.

We would like to explore other applications in the future. New applications may require designing reactive proximity data structures for more general graph classes, i.e., classes without sublinear separators. If finding exact nearest neighbors turns out to be too complex without sublinear separators, it would be interesting to design a reactive data structure supporting *approximate* nearest-neighbor queries.

As discussed in Section 5.4.1, an important factor in the runtime of any data structure based on separator hierarchies is the choice of separators. It may be of interest to compare the benefits

of a simpler but lower-quality separator construction algorithm versus a slower but higher-quality separator construction algorithm in future experiments.

Chapter 6

Stable-Matching Voronoi Diagrams

6.1 Introduction

The *Voronoi diagram* is a well known geometric structure with a broad spectrum of applications in computational geometry and other areas of Computer Science, e.g., see [16, 18, 27, 34, 131, 145, 161, 176]. The Voronoi diagram partitions the plane into regions. Given a finite set S of points, called *sites*, each point in the plane is assigned to the region of its closest site in S. Although the Voronoi diagram has been generalized in many ways, its standard definition specifies that each *Voronoi cell* or *region* of a site s is the set $V(s)$ defined as

$$\{p \in \mathbb{R}^2 \mid d(p,s) \leq d(p,s') \quad \forall s' \neq s \in S\}, \tag{6.1}$$

where $d(\cdot,\cdot)$ denotes the distance between two points. The properties of standard Voronoi diagrams have been thoroughly studied (e.g., see [16, 18]). For example, it is well known that in a standard Voronoi diagram for point sites in the plane every Voronoi cell is a connected, convex polygon whose boundaries lie along perpendicular bisectors of pairs of sites.

We introduced the stable matching problem in Chapter 4. In this chapter, we are interested in studying the algorithmic and combinatorial complexity of the diagrams that we call *stable-matching Voronoi diagrams*, which combine the notions of Voronoi diagrams and the one-to-many stable matching problem. These diagrams were introduced by Hoffman, Holroyd, and Peres [113], who provided existence and uniqueness proofs for such structures for potentially countably infinite sets of sites, but did not study their algorithmic or combinatorial complexities. A stable-matching Voronoi diagram is defined with respect to a set of sites in $\mathbb{R}^2$, which in this chapter we restrict to finite sets of n distinct points, each of which has an assigned finite numerical *quota* (which is also known as its "*appetite*") indicating the area of the region of points assigned to it. A preference relationship is defined in terms of distance, so that each point p in $\mathbb{R}^2$ prefers sites ordered by distance, from closest to farthest, and each site likewise prefers points ordered by distance. The stable-matching Voronoi diagram, then, is a partition of the plane into regions, such that (i) each site is associated with a region of area equal to its appetite, and (ii) the assignment of points to sites is stable in the sense that there is no blocking pair, defined as a site–point pair whose members prefer each other over their assigned matches. This is formalized in Definition 6.1. The regions are defined as closed sets so that boundary points lie on more than one region, analogously to Equation 6.1. See Figure 6.1.

Definition 6.1. Given a set S of n points (called sites) in $\mathbb{R}^2$ and a numerical appetite $A(s) > 0$ for each $s \in S$, the *stable-matching Voronoi diagram* of (S, A) is a subdivision of $\mathbb{R}^2$ into $n + 1$ *regions*, which are closed sets in $\mathbb{R}^2$. For each site $s \in S$ there is a corresponding region C_s of area $A(s)$, and there is an extra region, $C_\emptyset$, for the the remaining "unmatched" points. The regions do not overlap except along boundaries (boundary points are included in more than one region). The regions are such that there are no blocking pairs. A blocking pair is a site $s \in S$ and a point $p \in \mathbb{R}^2$ such that (*i*) $p \notin C_s$, (*ii*) $d(p, s) < \max\{d(p', s) \mid p' \in C_s\}$, and (*iii*) $p \in C_\emptyset$ or $d(p, s) < d(p, s')$, where s' is a site such that $p \in C_{s'}$.

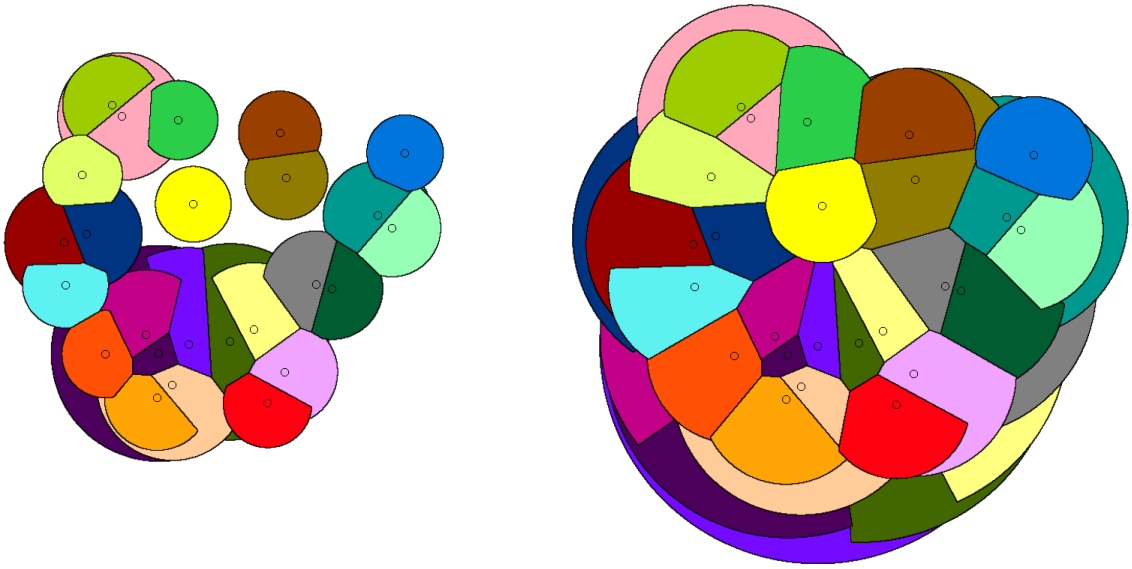


Figure 6.1: Stable-matching Voronoi diagrams for a set of 25 point sites, where each site in the left diagram has an appetite of 1 and each site in the right diagram has an appetite of 2. Each color corresponds to an individual cell, which is not necessarily convex or even connected.

As mentioned above, Hoffman *et al.* [113] show that, for any set of sites S and appetites, the stable-matching Voronoi diagram of S always exists and is unique. Technically, they consider the setting where all the sites have the same appetite, but the result applies to different appetites. They also describe a continuous process that results in the stable-matching Voronoi diagram: Start growing a circle from all the sites at the same time and at the same rate, matching the sites with all the points encountered by the circles that are not matched yet—when a site fulfills its appetite, its circle stops growing. The process ends when all the circles have stopped growing.

Note that this circle-growing method is analogous to a continuous version of the "deferred-acceptance" stable matching algorithm of Gale and Shapley [97]. The sites correspond to the set making proposals, and $\mathbb{R}^2$ to the set accepting and rejecting proposals. The sites propose to the points in order by preference (with the growing circles), as in the deferred acceptance algorithm. The difference is that, in this setting, points receive all the proposals also in order by their own preference, so they always accept the first one and reject the rest.

Clearly, the circle-growing method can be simulated to obtain a numerical approximation of the diagram, but this would not be an effective discrete algorithm, which is one of the interests of the present chapter.

Figure 6.2 shows a side-by-side comparison of the standard and stable-matching Voronoi diagrams. Note that the standard Voronoi diagram is stable in the same sense as the stable-matching Voronoi diagram: by definition, every point is matched to its first choice among the sites, so there can be no blocking pairs. In fact, the standard Voronoi diagram of a set of sites can be seen as the limit of the stable-matching Voronoi diagram as all the appetites grow to infinity, in the following sense: for any point p in $\mathbb{R}^2$, and for sufficiently large appetites for all the sites, p will belong to the region of the same site in the standard and stable-matching Voronoi diagrams.

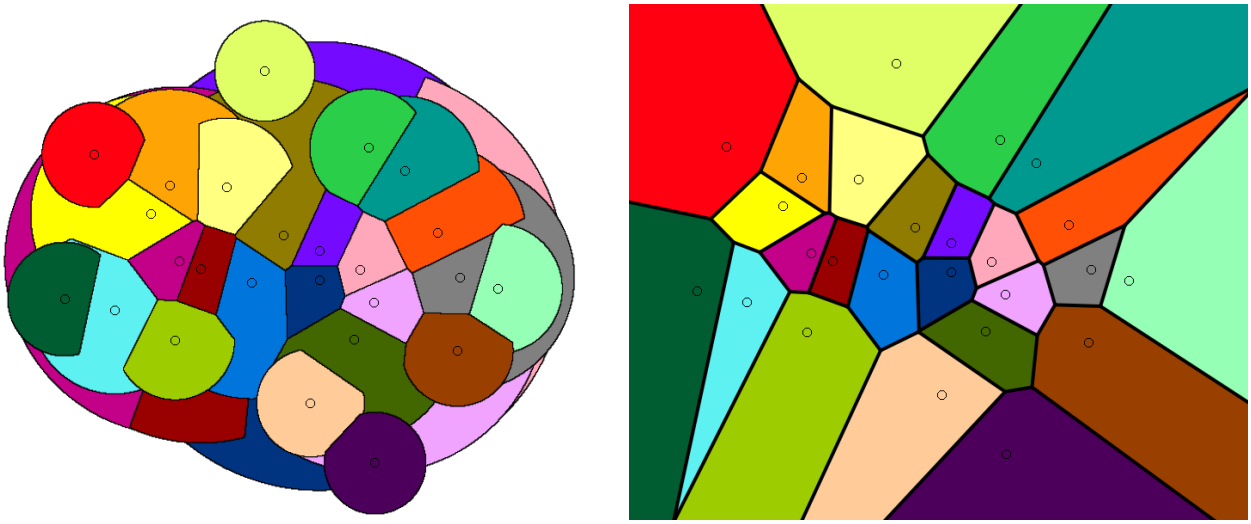


Figure 6.2: A stable-matching Voronoi diagram (left) and a standard Voronoi diagram (clipped to a rectangle) (right) for the same set of 25 sites. Each color represents a region.

A standard Voronoi diagram solves the *post office problem* of assigning points to their closest post office [133]. A stable-matching Voronoi diagram adds the real-world assumption that each post office has a limit on the size of its jurisdiction. Such notions may also be useful for political redistricting, where point sites could represent polling stations, and appetites could represent their capacities. We consider this application in Chapter 7. Nevertheless, depending on the appetites and locations of the sites, the regions of the sites in a stable-matching Voronoi diagram are not

necessarily convex or even connected (e.g., see Figure 6.1). Thus, we are interested in this chapter in characterizing the worst-case combinatorial complexity of such diagrams (i.e., the maximum number of faces, edges, and vertices among all diagrams with n sites), as well as finding an efficient algorithm for constructing them.

Related work. There are large volumes of work on the topics of Voronoi diagrams and stable matchings; hence, we refer the interested reader to surveys or books on the subjects (e.g., see [16, 18, 107, 118]).

A generalization of Voronoi diagram of particular interest are *power diagrams*, where a weight associated to each site indicates how strongly the site draws the points in its neighborhood. Power diagrams have also been considered for political redistricting [60]. Aurenhammer *et al.* [17] show that, given a set of sites in a square and a quota for each site, it is always possible to find weights for the sites such that, in the power diagram induced by those weights, the area of the region of each site within the square is proportional to its prescribed quota. Thus, both stable-matching Voronoi diagrams and power diagrams are Voronoi-like diagrams that allow predetermined region sizes. Power diagrams minimize the total squared distance between the sites and their associated points, while stable-matching Voronoi diagrams result in a stable matching.

In Chapter 4, we studied stable matching for preferences based on proximity. However, the sets there were discrete. Hence, we did not encounter the algorithmic and combinatorial challenges raised by stable-matching Voronoi diagrams in the plane.

Our contributions. In Section 6.2, we give a geometric interpretation of stable-matching Voronoi diagrams as the lower envelope of a set of cones, and discuss some basic properties of stable-matching Voronoi diagrams.

In Section 6.3, we give an $O(n^{2+\varepsilon})$ upper bound, for any $\varepsilon > 0$, and an $\Omega(n^2)$ lower bound for the number of faces and edges of a stable-matching Voronoi diagrams in the worst case, where n is the number of sites. The upper bound applies for arbitrary appetites, while the lower bound applies even in the special case where all the sites have the same appetite.

In Section 6.4, we show that stable-matching Voronoi diagrams cannot be computed exactly in an algebraic model of computation. In light of this, we provide a discrete algorithm for constructing them that runs in $O(n^3 \log n + n^2 f(n))$ time, where $f(n)$ is the runtime of a geometric primitive (which we define) that encapsulates this difficulty. This geometric primitive can be approximated numerically. We also show how to compute the diagram exactly when the distance metric is a polygonal convex distance function (Section 6.4.2) and when the plane is discretized.

We assume Euclidean distance as the distance metric throughout the chapter, except in Section 6.4.2. In particular, the upper and lower bounds on the combinatorial complexity apply to Euclidean distance. We conclude in Section 6.5.

6.2 The geometry of stable-matching Voronoi diagrams

As is now well known, a (2-dimensional) Voronoi diagram can be viewed as a lower envelope of cones in 3 dimensions, as follows [92]. Suppose that the set of sites are embedded in the plane $z = 0$. That is, we map each site $s = (x_s, y_s)$ to the 3-dimensional point $(x_s, y_s, 0)$. Then, we draw one cone for each site, with the site as the vertex, and growing to $+\infty$ all with the same slope. If we then view the cones from below, i.e., from $z = -\infty$ towards $z = +\infty$, the part of the cone of each site that we see corresponds to the Voronoi cell of the site. This is because two such cones intersect at points that are equally distant to both vertices. As a result, the xy-projection of their intersection corresponds to the perpendicular bisector of the vertices, and the boundaries of the Voronoi cells in the Voronoi diagram are determined by the perpendicular bisectors with neighboring sites.

Similarly, a stable-matching Voronoi diagram can also be viewed as the lower envelope of a set of cones. However, in this setting cones do not extending to $+\infty$. Instead, they are cut off at a finite height (which is a potentially different height for each cone, even if the associated sites have the same appetite). This system of cones can be generated by a dynamic process that begins with cones of height zero and then grows them all at the same rate, halting the growth of each cone as soon as its area in the lower envelope reaches its appetite (see Figure 6.3). This process mimics the circle-growing method by Hoffman et al. [113] mentioned before: if the z-axis is interpreted as time, the growing circles become the cones, and their lower envelope shows which circle reaches each point of the xy-plane first.

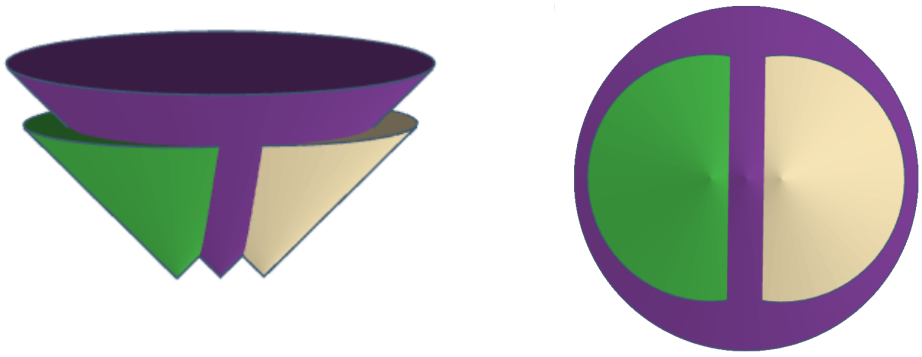


Figure 6.3: View of a stable-matching Voronoi diagram of 3 sites as the lower envelope of a set of cones.

A stable-matching Voronoi diagram consists of three types of elements:

- A *face* is a maximal, closed, connected subset of a stable cell. The stable cells can be disconnected, that is, a cell can have more than one face. There is also one or more *empty faces*, which are maximal connected regions not assigned to any site. One of the empty faces is the *external face*, which is the only face with infinite area.

- An *edge* is a maximal line segment or circular arc on the boundary of two faces. We call the two types of edges *straight* and *curved edges*, respectively. For curved edges, we distinguish between its incident convex face (the one inside the circle along which the edge lies) and its incident concave face.

- A *vertex* is a point shared by more than one edge. Generally, edges end at vertices, but curved edges may have no endpoints when they form a complete circle. This situation arises when the region of a site is isolated from other sites.

We say a set of sites with appetites is *not* in general position if two curved edges of the stable-matching Voronoi diagram are tangent, i.e., touch at a point p that is not an endpoint (e.g., two circles of radius 1 with centers 2 units apart). In this special case, we consider that the curved edges are split at p, and that p is a vertex.

In order to study the topology of the stable-matching Voronoi diagram, let the *bounding disk*, B_s, of a site, s, be the smallest closed disk centered at s that contains the stable cell of s. The bounding disks arise in the topology of the diagram due to the following lemma:

Lemma 6.2. *If part of the boundary between a face of site s and a face of site s' lies in the half-plane closer to s than to s', then that part of the boundary must lie along the boundary of the bounding disk B_s, and the convex face must belong to s.*

Proof. The boundary between the faces of s and s' cannot lie outside of B_s, by definition of the bounding disk. If the boundary is in the half-plane closer to s, then it also cannot be in the interior of B_s, because then there would exist a point p inside B_s and in the half-plane closer to s, but matched to s' (see Figure 6.4). In such a situation, s and p would be a blocking pair: s prefers p to the point(s) matched to it along B_s, and p prefers s to s'. □

Lemma 6.3. *The union of non-empty faces of the diagram is the union of the bounding disks of all the sites.*

Proof. For any site s, all the points inside the bounding disk of s must be matched. Otherwise, there would be a point, say, p, not matched to anyone but closer to s than points actually matched to s (along the boundary of B_s), which would be unstable, as p and s would be a blocking pair.

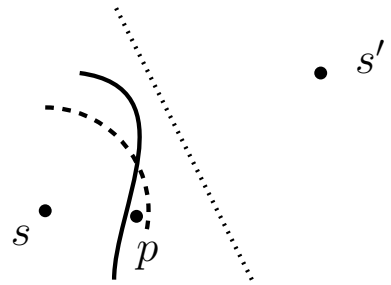


Figure 6.4: Illustration of the setting in the proof of Lemma 6.2. It shows the perpendicular bisector of two sites s and s' (dotted line), the boundary of the bounding disk, B_s, of s (dashed circular arc), and a hypothetical boundary between the faces of sites s and s' (solid curve). In this setting, s and p would be a blocking pair.

Moreover, points outside of all the bounding disks cannot be matched to anyone, by definition of the bounding disks. □

Lemma 6.4 (Characterization of edges)**.**

1. *A straight edge separating faces of sites s and s' can only lie along the perpendicular bisector of s and s'.*

2. *A curved edge whose convex face belongs to site s lies along the boundary of B_s. Moreover, if the concave face belongs to a site s', the edge must be contained in the half-plane closer to s than s'.*

3. *Empty faces can only be concave faces of curved edges.*

Proof. Claims (1) and (2) are consequences of Lemma 6.2, and Claim (3) is a consequence of Lemma 6.3. □

6.3 Combinatorial complexity

6.3.1 Upper bound on the number of faces

As mentioned in Section 6.2, a stable-matching Voronoi diagram can be viewed as the lower envelope of a set of cones. Sharir and Agarwal [5] provide results that characterize the combinatorial complexity of the lower envelope of certain sets of functions, including cones.

Formally, the *lower envelope* (also called *minimization diagram*) of a set of bivariate continuous functions $F = \{f_1(x, y), \ldots, f_n(x, y)\}$ is the function

$$E_F(x, y) = \min_{1 \le i \le n} f_i(x, y),$$

where ties are broken arbitrarily. The lower envelope of F subdivides the plane into maximal connected regions such that E_F is attained by a single function f_i (or by no function at all). The *combinatorial complexity* of the lower envelope E_F, denoted $K(F)$, is the number of maximal connected regions of E_F. To prove our upper bound, we use the following result:

Lemma 6.5 (Sharir and Agarwal [5], page 191)**.** *The combinatorial complexity $K(F)$ of the lower envelope of a collection F of n (partially defined) bivariate functions that satisfy the assumptions below is $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$.*[14]

- *Each $f_i \in F$ is a portion of an algebraic surface of the form $P_i(x, y)$, for some polynomial P_i of constant maximum degree.*
- *The vertical projection of each $f_i \in F$ onto the xy-plane is a planar region bounded by a constant number of algebraic arcs of constant maximum degree.*

Corollary 6.6. *A stable-matching Voronoi diagram for n sites has $O(n^{2+\varepsilon})$ faces, for any $\varepsilon > 0$.*

[14] The theorem, as stated in the book (Theorem 7.7), includes some additional assumptions, but the book then shows that they are not essential.

Proof. It is clear that the finite, "upside-down" cones whose lower envelope forms the stable-matching Voronoi diagram of a set of sites satisfy the above assumptions. In particular, their projection onto the xy-plane are disks. Note that the bound still applies if we include the empty faces, as Lemma 6.5 still holds if we add an extra bivariate function $f_{n+1}(x, y) = z^*$, where z^* is any value higher than the height of any cone (i.e., f_{n+1} is a plane that "hovers" over the cones). Such a function would have a face in the lower envelope for each empty face in the stable-matching Voronoi diagram. □

6.3.2 Upper bound on the number of edges and vertices

Euler's formula relates the number of faces in a planar graph with the number of vertices and edges. By viewing the stable-matching Voronoi diagram as a graph, we can use Euler's formula to prove that the $O(n^{2+\varepsilon})$ upper bound also applies to the number of edges and vertices. In order to do so, we will need to show that the average degree is more than two, which is the purpose of the following lemmas.

In this section, we assume that sites are in general position (as defined in Section 6.2). However, note that non-general-position constructions cannot yield the worst-case complexity. This is because if two curved edges coincide exactly at a point that is not an endpoint, we can perturb slightly the site locations to move them a little closer, which creates a new vertex and edge. For the same reason, we also assume that no vertex has degree four or more, which requires four or more sites to lie on the same circle (as in the standard Voronoi Diagram).

Lemma 6.7. *The following sequences of consecutive edges along the boundary of two faces cannot happen: 1. Straight–straight. 2. Curved–curved. 3. Straight–curved–straight.*

Proof.

1. Straight edges separating two faces of sites s and s' are constrained to lie along the perpendicular bisector of s and s' (Lemma 6.4). Therefore, two consecutive straight edges would not be maximal.

2. Curved edges separating a convex face of a site s are constrained to lie along the boundary of the bounding disk of s (Lemma 6.4). Thus, two consecutive curved edges would not be maximal (under the assumption of general position).

3. In such a case, not both straight edges could lie along the perpendicular bisector. □

Incidentally, *curved–straight–curved* sequences *can* happen, and can be seen in Figure 6.2.

Lemma 6.8. *A vertex with degree two cannot be adjacent to two vertices with degree two.*

Proof. A vertex with degree two connects two edges separating the same two faces. If there were a node with degree two adjacent to two other nodes with degree two, we would either have four consecutive edges separating the same two faces or a triangular face inside another face. However, neither case could avoid the sequences of edges given in Lemma 6.7. □

Lemma 6.9. *The average degree is at least* 2.25.

Proof. Note that all vertices have degree at least 2, as they are the endpoints of edges, and every edge has different faces on each side. Also recall the assumption that there are no nodes with degree more than three, as this cannot yield a worst-case number of vertices nor edges.

Thus, all nodes have degree two or three. Let n be the number of 2-degree vertices, and k the number of 3-degree vertices. The average degree is $(2n + 3k)/(n + k) = 2 + k/(n + k)$. Thus, we need to show that $k/(n + k) \geq 1/4$, or, rearranging, that $k \geq n/3$.

By Lemma 6.8, a vertex with degree two cannot be adjacent to two vertices with degree two. Among the n 2-degree vertices, say m_1 are connected with another 2-degree node, while the remaining m_2 are adjacent only to 3-degree nodes. Then, there are $m_1/2$ edges in between 2-degree

nodes, and $m_1+2m_2 = n+m_2$ edges connecting 2-degree nodes with 3-degree nodes. This means that $k \geq (n + m_2)/3$, completing the proof. □

Lemma 6.10. *Let V, E and F be the number of vertices, edges, and faces of the stable-matching Voronoi diagram of a set of sites S. Then, $V \leq 8F - 16$ and $E \leq 9F - 18$.*

Proof. For this proof, suppose that there are no curved edges that form a full circle. Note that the presence of such edges can only reduce the number of vertices and edges, as for each such edge there is a site with a single edge and no vertices.

Without such edges, the vertices and edges of the stable-matching Voronoi diagram form a planar graph, and V, E, F are the number of vertices, edges, and faces of this graph, respectively. Moreover, let C be the number of connected components. Due to Euler's formula for planar graphs, we have $F = E - V + C + 1$, and thus $F \geq E - V + 2$. Moreover, by Lemma 6.9, the sum of degrees is at least $2.25V$, so $2E \geq 2.25V$. Combining the two relations above, we have $V \leq 8F - 16$ and $E \leq 9F - 18$. □

We conclude by stating the main theorem of this section, which is a combination of Corollary 6.6 and Lemma 6.10:

Theorem 6.11. *A stable-matching Voronoi diagram for n point sites has $O(n^{2+\varepsilon})$ faces, vertices, and edges, for any $\varepsilon > 0$.*

6.3.3 Lower bound

We show a quadratic lower bound on the number of faces in the worst case by constructing an infinite family of instances with $\Omega(n^2)$ faces. To start, we give such a family of instances where sites have arbitrary appetites. This introduces the technique behind our second, more intricate construction, which only uses sites with appetite 1. This shows that the $\Omega(n^2)$ lower bound holds

even in this restricted case where all the sites have the same appetite. The lower bound extends trivially to vertices and edges as well.

Lemma 6.12. *A stable-matching Voronoi diagram for n point sites has $\Omega(n^2)$ faces, edges, and vertices in the worst case.*

Proof. Consider the setting in Figure 6.5. Assume n is even. We divide the sites into two sets, X and Y, of size $m = n/2$ each. The sites in X are arranged vertically, spaced evenly, and spanning a total height of 2. Note that the *standard* Voronoi diagram of the sites in X alone consists of infinite horizontal strips. The top and bottom sites have strips extending vertically indefinitely, while the rest have thin strips of height $2/(m-1)$.

The sites in Y are aligned vertically with the center of the strips. Half of the sites in Y lie on each side of the sites in X. The sites in Y have appetite π, so their "ideal" stable cell is a disk of radius 1 around them. They are spaced evenly at a distance of at least 2 (e.g., 2.1) of each other and of the first m sites, so that each site in Y is the first choice of all the points within distance 1 of it.

Now, consider the resulting stable-matching Voronoi diagram when the sites of X have large ($\gg m^2$) and equal appetites. To visualize it, consider the circle-growing method from [113] described in Section 6.1, where a circle starts growing from each site at the same time and rate, and any unassigned point reached by a circle is assigned to the corresponding site.

The sites in Y are allowed to grow without interference with any other site until they fulfill their appetite and freeze. Their region is thus a disk with diameter 2, which spans all the thin strips (Figure 6.5). The sites in X start growing their region as a disk, which quickly reach the disks of the sites above and below. Then, the regions are restricted to keep growing along the horizontal strips. The sites in X keep growing and eventually reach a region already assigned to a site in Y (which already fulfilled their appetite and stopped growing by this time). They continue growing along the strips past the regions already assigned to sites in Y. Eventually, they also freeze when they fulfill their appetite. The top and bottom sites are the first to fulfill their appetite, since they

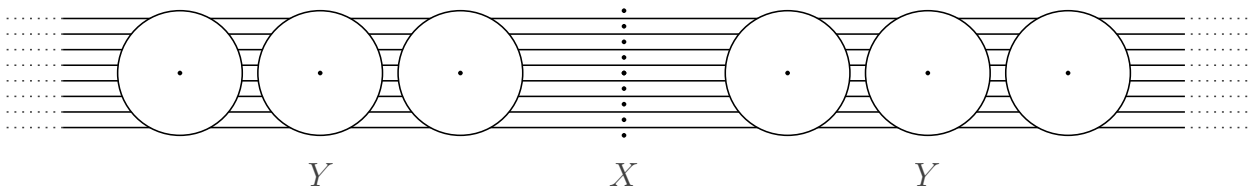


Figure 6.5: Lower bound construction for Lemma 6.12.

are not restricted to grow along thin strips, but we are not interested in the topology of the diagram beyond the thin strips. The only thing we need for our construction is that the appetite of the sites in X is large enough so that their stable cells reach past the stable cells of the furthest sites in Y along the strips.

Informally, the regions of the sites in Y "cut" the thin strips of the sites in X. Each site in Y creates $m-2$ additional faces, (the top and bottom sites in X do not have thin strips), and hence the number of faces is at least $m(m-2) = \Omega(n^2)$. □

In the proof of Lemma 6.12, sites in X and Y have different roles. Sites in X create a linear number of long, thin faces which can all be cut by a single disk. This is repeated a linear number of times, once for each site in Y, yielding quadratic complexity. However, this construction relies on the sites in X having larger appetites than the sites in Y. Next, we consider the case where all the sites have appetite one. The proof will follow the same idea, but now the thin and long strips will be circular strips. Lemma 6.13 is an auxiliary result used in the proof.

Lemma 6.13. *Let A be an annulus of width $\varepsilon > 0$, and D a disk centered outside the outer circle of A, with radius smaller than the inner radius of A, and tangent to the inner circle of A (Figure 6.6). Then,*

$$\lim_{\varepsilon \to 0} \frac{area(A \cap D)}{area(A)} = 0$$

Proof. Consider the smallest circular sector S of A that contains the asymmetric lens $A \cap D$ (the sector determined by angle α in Figure 6.6). Since $A \cap D$ is contained in S, to prove the lemma

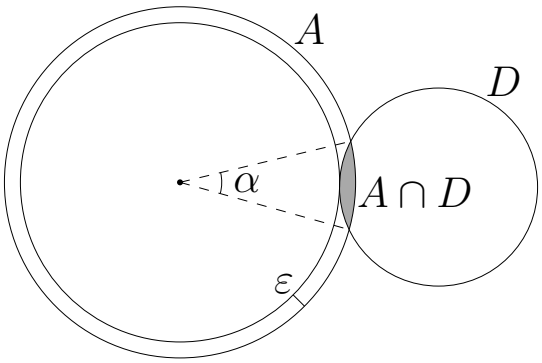


Figure 6.6: Setting in Lemma 6.13.

it suffices to show that $\lim_{\varepsilon \to 0} \frac{area(S)}{area(A)} = 0$. Note that $\frac{area(S)}{area(A)}$ is precisely $\frac{\alpha}{2\pi}$, and it is clear that $\lim_{\varepsilon \to 0} \frac{\alpha}{2\pi} = 0$. □

Theorem 6.14. *A stable-matching Voronoi diagram for n point sites has $\Omega(n^2)$ faces, edges, and vertices in the worst case, even when all the regions are restricted to have the same appetite.*

Proof. Assume n is a multiple of 4. We divide the sites into two sets, X and Y, of size $m = n/2$ each.

Let $\varepsilon_1, \varepsilon_2$ be two parameters with positive values that may depend on m. It will be useful to think of them as very small, since we will argue that the construction works for sufficiently small values of ε_1 and ε_2. Specific values for ε_1 and ε_2 are hard to express analytically but unimportant as long as they are small enough.

The m sites in X, $s_1, \ldots, s_m$, lie, in this order, along a circle of radius ε_1. They are all almost evenly spaced around the circle, except that the angle between s_1 and s_m is slightly larger than the others: the angle between s_1 and s_m is increased by ε_2, and the angles between the rest of pairs of consecutive sites are reduced so that they are all equal.

The *standard* Voronoi diagram of the sites in X consists of infinite angular regions, with those of s_1 and s_m slightly wider than those of the remaining sites. Consider the circle-growing method applied to the sites of X alone. The regions are constrained to grow in the corresponding angular

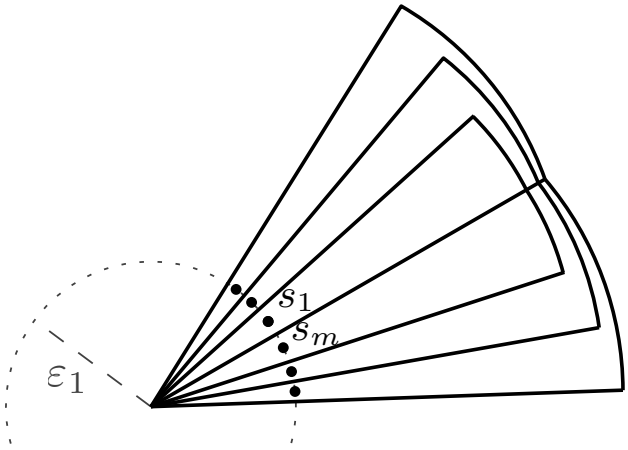


Figure 6.7: Configuration of the sites in X in the proof of Theorem 6.14. The arc between s_1 and s_m is slightly wider than the rest. Sites $s_2, s_3, s_{m-1}, s_{m-2}$ are shown (unlabeled) with their regions. The remaining sites are omitted for clarity. The figure is not to scale, as in the actual construction ε_1 and ε_2 need to be much smaller, but even here we can appreciate the "wrapping around" effect.

region in the standard Voronoi region. Since s_1 and s_m have wider angles, they fill their appetite slightly before the rest, which all grow at the same rate. How much earlier depends on ε_2. Once s_1 and s_m fulfill their appetite and stop growing, their angular regions become "available" to the other sites. The circles of s_2 and s_{m-1} are the closest to the angular regions of s_1 and s_m, respectively, and thus start covering it to fulfill their appetite. In turn, this results in s_2 and s_{m-1} fulfilling their appetite and freezing their circles earlier than the remaining sites. Their respective neighbors, s_3 and s_{m-2}, have the next closest circles to the angular regions of the sites that already stopped growing, and thus they use it to fill their appetite. This creates a cascading effect starting with s_1 and s_m where the region of each site consists of a wedge that ends in a thin circular strip that "wraps around" the regions of the prior sites. This effect is illustrated in Figure 6.7.

As ε_2 approaches zero, the unfulfilled appetite of the sites other than s_1 and s_m at the time s_1 and s_m fill their appetite becomes arbitrarily small. This results in arbitrarily thin circular strips. Note, however, that the circular arcs bounding each strip are not exactly concentric, as each one is centered at a different site. Thus, depending on ε_1, the strips might not wrap around all the way to the regions of s_1 and s_m. However, as ε_1 approaches zero, the sites get closer to each other, and thus their circular arcs become arbitrarily close to being concentric. It follows that if ε_1 is small

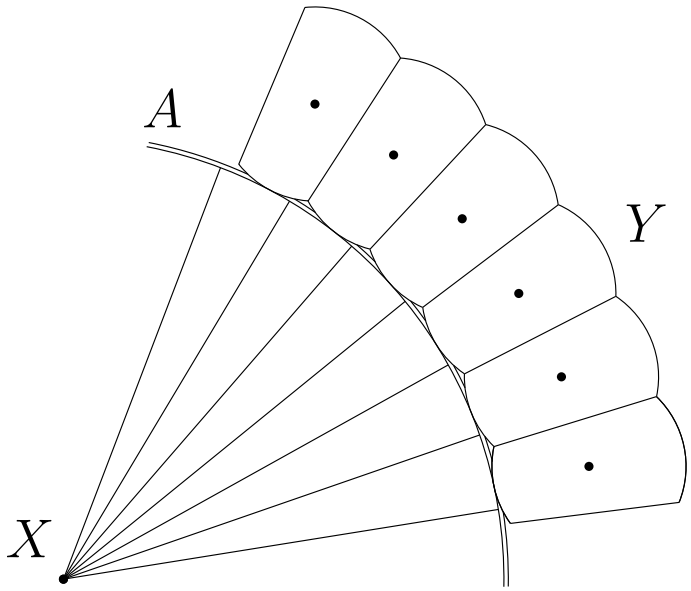


Figure 6.8: Configuration in the proof of Theorem 6.14. For clarity, only the regions of 6 sites in X and 6 sites in Y are shown. The strips inside A are also omitted. The figure is not to scale, as in the actual construction the annulus A needs to be much thinner.

enough (relative to ε_2), the circular strip of each site will wrap around all the way to the angular region of s_1 and s_m. This concludes the first half of the construction, where we have a linear number of arbitrarily thin, long strips.

Let A be the annulus of minimum width centered at the center of the circle of the sites in X and containing all the circular strips. The sites in Y lie evenly spaced along a circle concentric with A. The radius of the circle is such that the regions of the sites in Y are tangent to the inner circle of A, as in Figure 6.8.

Since the wedges of the sites in X are very thin, the sites in Y are closer to A than the sites in X. Thus, the presence of X does not affect the stable cells of the sites in Y. Each stable cell of a site in Y is the intersection of a disk and a wedge of angle $2\pi/m$, with a total area of 1 (Figure 6.8). The important aspect is how the presence of the stable cells of the sites in Y affects the stable cells of the sites in X. Some of the area of A that would be assigned to sites in X is now assigned to sites in Y. Thus, the sites in X need to grow further to make up for the lost appetite. However, recall that A can be arbitrarily thin. Hence, by Lemma 6.13, the fraction of the area of A "eaten" by sites

in Y can be arbitrarily close to zero. As this fraction tends to zero, the distance that the sites in X need to reach further to fulfill the lost appetite also tends to zero. Thus, if A is sufficiently thin, the distance that the regions of s_1 and s_m reach further is so small that the strips of s_2 and s_{m-1} still wrap around the regions of s_1 and s_m, respectively, to fulfill their appetite. Similarly, the strips of s_3 and s_{m-2} still wrap around the regions of the prior sites, and so on. Thus, if A is sufficiently thin, the strips of all the sites in X still wrap around to the regions of s_1 or s_m.

In this setting, half of the strips are at least as long as a quarter of the circle, and each of those gets broken into $\Theta(m)$ faces by the regions of the sites in Y. Therefore, the circular strips are collectively broken into a total of $\Theta(m^2) = \Theta(n^2)$ faces. □

6.4 Algorithms

In general, a stable-matching Voronoi diagram cannot be computed in an algebraic model of computation, as it requires computing transcendental functions such as trigonometric functions.

Observation 6.15. For infinitely-many sets of sites in general position and with algebraic coordinates, the radii of some of the sites' bounding disks cannot be computed exactly in an algebraic model of computation.

Proof. Consider a set with only two sites, s_1 and s_2, with appetite 1 and aligned horizontally at distance $2b$ from each other. By symmetry, the two bounding disks will have the same radius r. Assume that $b < \sqrt{1/\pi}$, so that the stable cells of s_1 and s_2 share a vertical edge. Consider the rectangular triangle with one vertex at s_1, another at the midpoint between s_1 and s_2, and the last at the top of the shared vertical edge (see Figure 6.9). Let α be the angle of the triangle at the vertex at s_1, and a the length of the opposite side. The problem is, then, to determine the value of r which

satisfies

$$\pi r^2 \left(1 - \frac{2\alpha}{2\pi}\right) + 2 \cdot \frac{ab}{2} = 1.$$

Using the equalities $\sin\alpha = a/r$ and $\cos\alpha = b/r$, we obtain

$$r^2(\pi - \cos^{-1}\frac{b}{r}) + br\sin(\cos^{-1}\frac{b}{r}) = 1,$$

that is, r is the solution of the equation

$$r^2(\pi - \cos^{-1}\frac{b}{r}) + b\sqrt{r^2 - b^2} = 1,$$

which cannot be solved in an algebraic model of computation because $\cos^{-1}$ is a transcendental (i.e., non-algebraic) function. Such a construction appears in infinitely-many sets of points, implying the claim. □

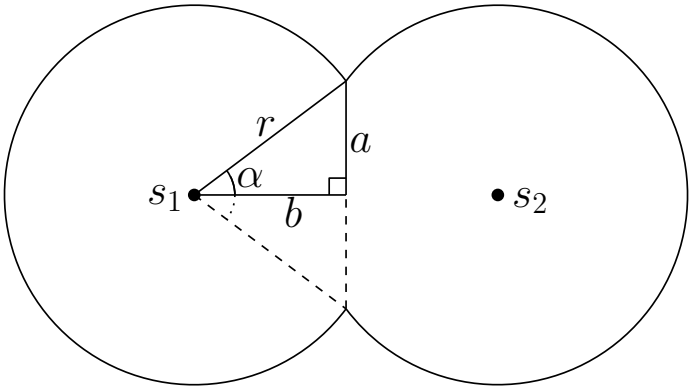


Figure 6.9: Setting in the proof of Observation 6.15.

Thus, we present three different approaches for computing stable-matching Voronoi diagrams, each of which requires a different compromise. First, we give an exact, discrete algorithm which relies on a geometric primitive. This primitive, which we define, encapsulates the problematic computations, and can be approximated numerically to arbitrary precision. Second, we show how to compute the geometric primitive exactly for polygonal convex distance functions. Using this, the

diagram based on Euclidean distance can be approximated by a convex distance function induced by a regular polygon with many sides, which approximates a circle. Finally, one can keep the Euclidean distance untouched but discretize the plane into pixels instead. This allows us to use the first-choice chain algorithm from Chapter 4.

6.4.1 Discrete algorithm

Preliminaries. Let us introduce the notation used in this section. The algorithm deals with multiple diagrams. In this context, a *diagram* is a subdivision of $\mathbb{R}^2$ into *regions*. Each region is a set of one or more faces bounded by straight and circular edges. The regions do not overlap except along boundaries (boundary points are included in more than one region). Each region is assigned to a unique site, but not all sites necessarily have a region. There is also an "unassigned" region consisting of the remaining faces. The *domain* of a diagram is the subset of points of $\mathbb{R}^2$ in any of its assigned regions. If D is a diagram, $D(s)$ denotes the region of site s, which might be empty. If D and D' are diagrams and the domain of D is a subset of the domain of D', we say that D and D' are *coherent* if, for every site s, $D(s) \subseteq D'(s)$. The data structures used to represent diagrams are discussed later.

Recall that we are given a set S of n sites, each with its own appetite $A(s)$. The goal is to compute the (unique) stable-matching diagram of S for those appetites, denoted by D^*. For a site s, let $B^*(s)$ be the bounding disk of $D^*(s)$, and $r^*(s)$ the radius of $B^*(s)$ (the * superscript is used for notation relating to the sought solution). Recall that the union of all the bounding disks $B^*(s)$ equals the domain of D^* (Lemma 6.3), and that the bounding disks may not be disjoint.

We call an ordering $s_1, \ldots, s_n$ of the sites of S *proper* if the sites are sorted by increasing radius of their bounding disks, breaking ties arbitrarily. That is, $i < j$ implies $r^*(s_i) \leq r^*(s_j)$. Such an ordering is initially unknown, but it is discovered in the course of the algorithm. Given a proper ordering, for $i = 1, \ldots, n$, let $B_{1..i} = \{B^*(s_1), \ldots, B^*(s_i)\}$ denote the set of bounding disks

of the first i sites, and $\cup B_{1..i} = B^*(s_1) \cup \cdots \cup B^*(s_i)$ the union of those disks. Let $\hat{B}(s_i) = B^*(s_i) \setminus \cup B_{1..i-1}$ be the part of $B^*(s_i)$ that is not inside a prior bounding disk in the ordering. Let $S_{i..n} = \{s_i, \ldots, s_n\}$, and $V_{i..n}$ be the *standard* Voronoi diagram of $S_{i..n}$. Finally, let $\hat{V}_{i..n}$ be $V_{i..n}$ restricted to the region $\hat{B}(s_i)$. This notation is illustrated in Figure 6.10.

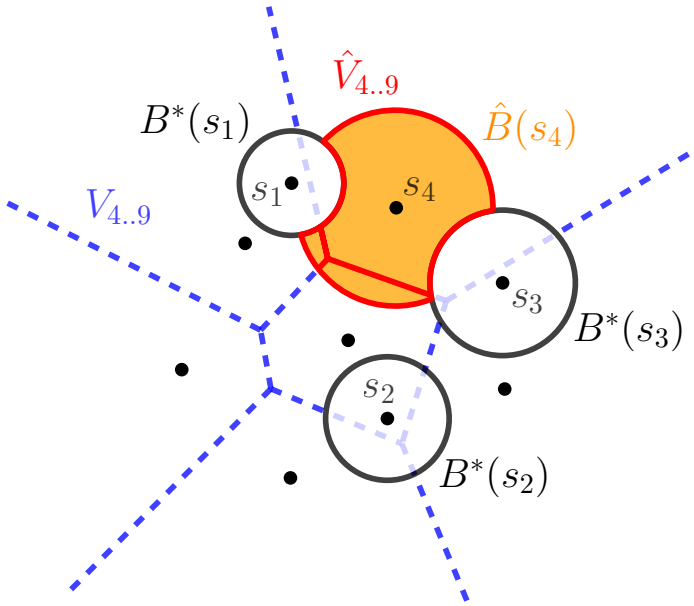


Figure 6.10: Notation used in the algorithm. The disks in $B_{1..3}$ are shown in black, the edges of $V_{4..9}$ are shown dashed in blue, and the interior of $\hat{B}(s_4)$ is shown in orange. The edges of $\hat{V}_{4..9}$ are overlaid on top of everything with red lines. Note that $\hat{V}_{4..9}$ is a diagram with three assigned regions, the largest assigned to s_4 and the others to unlabeled sites.

Incremental construction. The algorithm constructs a sequence of diagrams, $D_0, \ldots, D_n$. The starting diagram, D_0, has an empty domain. We expand it incrementally until $D_n = D^*$. The diagrams are constructed in a greedy fashion: every D_i is coherent with D^*. Thus, once a section of the plane is assigned in D_i to some site, that assignment is definitive and remains part of $D_{i+1}, \ldots, D_n$.

We construct D^* one bounding disk at a time, ordered according to a proper ordering $s_1, \ldots, s_n$ (we address how to find this ordering later): the domain of each D_i is $\cup B_{1..i}$.[15] Thus, D_i can be

[15] An intuitive alternative approach is to construct D^* one stable cell at a time. This is also possible, but the advantage of constructing it by bounding disks is that the topology of the intermediate diagrams D_i is simpler, as

constructed from D_{i-1} by assigning $\hat{B}(s_i)$ (the $\hat{}$ mark is used for notation relating to the region added to D_i at iteration i).

Since the boundaries of the bounding disks do not necessarily align with the edges of D^*, D_i may contain a face of D^* only partially. This can be seen in Figure 6.11, which illustrates the first few steps of the incremental construction.

At iteration i, we assign $\hat{B}(s_i)$ as follows. From $\hat{B}(s_i)$ and the standard Voronoi of the remaining sites, $V_{i..n}$, we compute the 2diagram $\hat{V}_{i..n}$. We then construct D_i as the combination of D_{i-1} and $\hat{V}_{i..n}$. That is, for each site s, $D_i(s) = D_{i-1}(s) \cup \hat{V}_{i..n}(s)$. We first show that this assignment is correct.

Lemma 6.16. *For any i with $0 \leq i \leq n$, D_i is coherent with D^*.*

Proof. We use induction on i. The claim is trivial for $i = 0$, as no site has a region in D_0. We show that if D_{i-1} is coherent with D^* and D_i is constructed as described, D_i is also coherent. In other words, we show that $\hat{V}_{i..n}$ is coherent with D^*.

Let s be an arbitrary site in $S_{i..n}$. We need to show that $\hat{V}_{i..n}(s) \subseteq D^*(s)$. Let p be an arbitrary point in the interior of $\hat{V}_{i..n}(s)$. We show that p is also an interior point of $D^*(s)$. First, note that p does not belong in the stable cell of any of $s_1, \ldots, s_{i-1}$, because the regions of these sites are fully contained in $\cup B_{1..i-1}$, and $\hat{V}_{i..n}$ is disjoint from $\cup B_{1..i-1}$ except perhaps along boundaries.

By virtue of being in the interior of $V_{i..n}(s)$, p has s as first choice among the sites in $S_{i..n}$. We show that s also prefers p over *some* of its assigned points in D^*, and thus they need to be matched or they would be a blocking pair. We consider two cases:

- $s = s_i$. In this case, $\hat{V}_{i..n}$ is a subset of $B^*(s)$, so s prefers p over some of its matched points (those at distance $r^*(s)$).

it can be described as a union of disks, whereas stable cells have complex (and even disjoint) shapes. The simpler topology makes the geometric operations we do on these diagrams easier, in particular the geometric primitive from Definition 6.22.

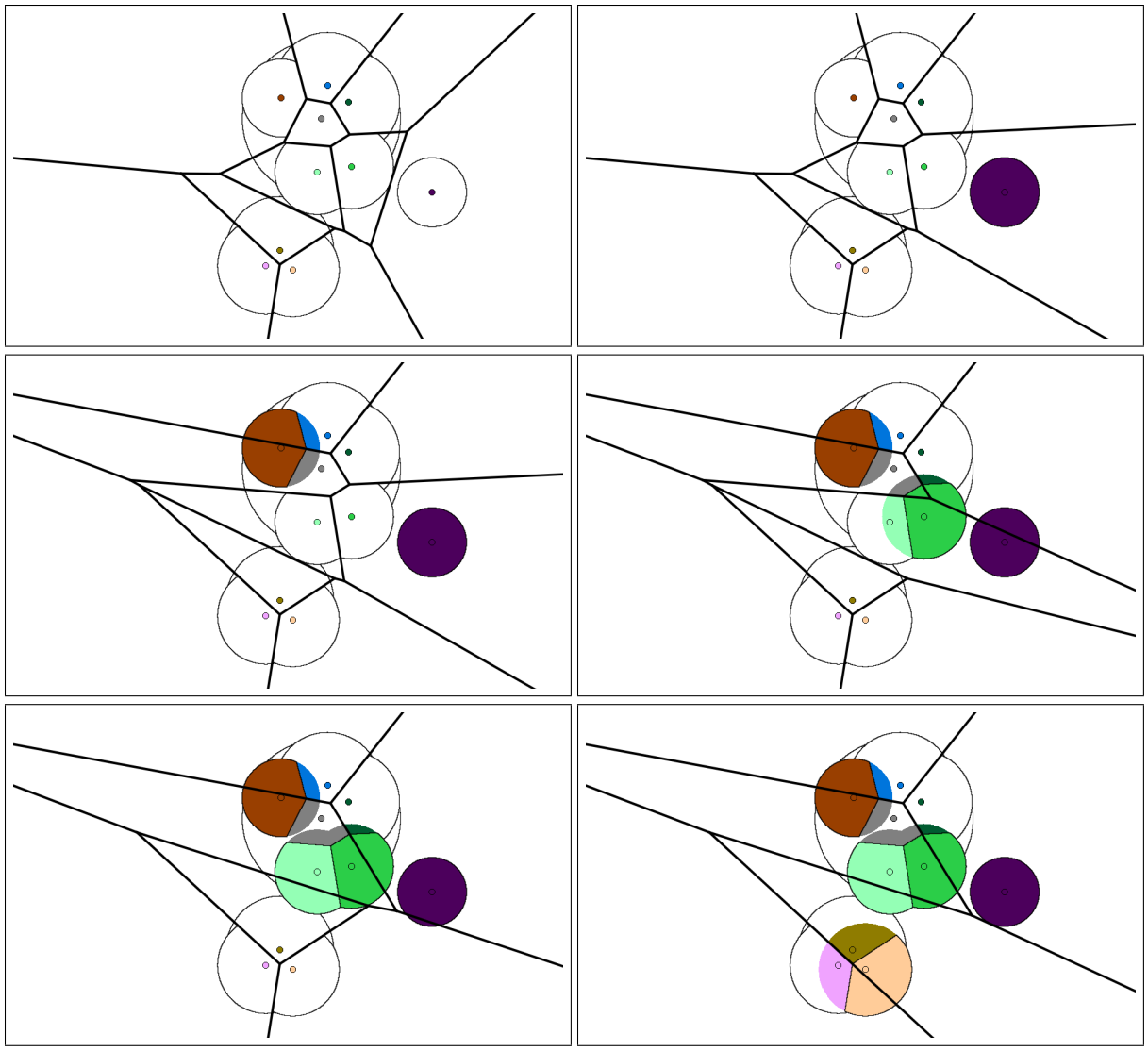


Figure 6.11: Partial diagrams $D_0, \ldots, D_5$ computed in the first five iterations of the algorithm for a set of sites with equal appetites. At each iteration i, the edges of the standard Voronoi diagram $V_{i..n}$ of $S_{i..n}$ are overlaid in thick lines. The edges of the stable-matching Voronoi diagram (unknown to the algorithm) are overlaid in thin lines.

- $s \neq s_i$. In this case, note the following three inequalities: *(i)* $d(p, s) < d(p, s_i)$ because p is in the interior of $V_{i..n}(s)$; *(ii)* $d(p, s_i) < r^*(s_i)$ because p is in the interior of $B^*(s_i)$; *(iii)* $r^*(s_i) \leq r^*(s)$ because s appears after s_i in the proper ordering. Chaining all three, we get that $d(p, s) < r^*(s)$, i.e., p is inside the bounding disk of s. Thus, s prefers p to some of its matched points (those at distance $r^*(s)$).

□

Corollary 6.17. *The diagrams D_n and D^* are the same.*

Proof. The domain of D_n is $\cup B_{1..n}$ by construction. The domain of D^* is also $\cup B_{1..n}$ by Lemma 6.3. By Lemma 6.16, they are coherent, and so it must be that $D_n(s) = D^*(s)$. □

Finding the next bounding disk. The proper ordering $s_1, \ldots, s_n$ cannot be computed in a preprocessing step. Instead, the next site s_i is discovered at each iteration. Consider the point where we have computed D_{i-1} and want to construct D_i ($1 \leq i \leq n$). At this point, we have found the ordering up to s_{i-1}. Hence, we know which sites are in $S_{i..n}$, but we do not know their ordering yet. In this step, we need to find a site s in $S_{i..n}$ minimizing $r^*(s)$, and we need to find the radius $r^*(s)$ itself. The site s can then be the next site in the ordering, i.e., we can "label" s as s_i. If there is a tie for the smallest bounding disk among those sites, then there may be several valid candidates for the next site s_i. The algorithm finds any of them and labels it as s_i.

To find a site s in $S_{i..n}$ minimizing $r(s)$, note the following results.

Lemma 6.18. *If $r^*(s) \leq r^*(s')$, every point p in $D^*(s)$ satisfies $d(p, s) \leq d(p, s')$.*

Proof. Suppose, for a contradiction, that $r^*(s) \leq r^*(s')$ and p is a point in $D^*(s)$, but $d(p, s') < d(p, s)$. Clearly, p prefers s' to s. We show that s' also prefers p over some of its assigned points in D^*, and thus p and s' are a blocking pair.

If we combine the three inequalities $d(p, s') < d(p, s)$, $d(p, s) \leq r^*(s)$ (because p is in $D^*(s)$), and $r^*(s) \leq r^*(s')$, we see that $d(p, s') < r^*(s')$. Thus, s' prefers p to the points matched to s' along the boundary of its bounding disk. □

Corollary 6.19. *For any s_i in a proper ordering, $D^*(s_i) \subseteq V_{i..n}(s_i)$.*

Proof. According to Lemma 6.18, every point p in $D^*(s_i)$ satisfies $d(p, s_i) \leq d(p, s_j)$ for any other site s_j with $r_j \geq r_i$, and this includes every site in $S_{i+1..n}$. □

Based on Corollary 6.19, the idea for finding a site with the next smallest bounding disk is to compute what would be the stable cell of each site s in $S_{i..n}$ if it were constrained to be a subset of $V_{i..n}(s)$. As we will see, among those stable cells, the one with the smallest bounding disk is correct.

More precisely, for each site s in $S_{i..n}$, let $A_i(s) = A(s) - area(D_{i-1}(s))$ be the *remaining appetite* of s at iteration i: the starting appetite $A(s)$ of s minus the area already assigned to s in D_{i-1}. We define an *estimate cell* $D_i^\dagger(s)$ for site s at iteration i as follows: $D_i^\dagger(s)$ is the union of $D_{i-1}(s)$ and the intersection of $V_{i..n}(s) \setminus \cup B_{1..i-1}$ with a disk centered at s such that that intersection has area $A_i(s)$. Note that if $area(V_{i..n}(s) \setminus \cup B_{1..i-1}) < A_i(s)$, no such disk exists. In this case, $D_i^\dagger(s)$ is not well-defined. If it is well-defined, we use $B_i^\dagger(s)$ to refer to its bounding disk (the smallest disk centered at s that contains $D_i^\dagger(s)$), and $r_i^\dagger(s)$ to refer to the radius of $B_i^\dagger(s)$. Otherwise, we define $r_i^\dagger(s)$ as $+\infty$.

Lemma 6.20. *At iteration i, for any site $s \in S_{i..n}$, $r^*(s) \leq r_i^\dagger(s)$. In addition, if $r^*(s)$ is minimum among the radii r^* of the sites in $S_{i..n}$, then, $r^*(s) = r_i^\dagger(s)$ and $D^*(s) = D_i^\dagger(s)$.*

Proof. For the first claim, let s be a site in $S_{i..n}$. Since D_{i-1} is coherent with D^* (Lemma 6.16), the region $D_{i-1}(s)$ is in both $D^*(s)$ and $D_i^\dagger(s)$. The appetite of s that is not accounted for in $D_{i-1}(s)$ is $A_i(s)$, and it must be fulfilled outside the domain of D_{i-1}, $\cup B_{1..i-1}$.

In $D_i^\dagger(s)$, s fulfills the rest of its appetite with the points in $V_{i..n}(s) \setminus \cup B_{1..i-1}$ closest to it. Note that all these points have s as first choice among the sites in $S_{i..n}$. Thus, the remaining sites in $S_{i..n}$ cannot "steal" those points away from s, so s for sure does not need to be matched to points even further than that. In other words, in the worst case for s, in D^*, s fulfills the rest of its appetite, $A_i(s)$, with those points, and thus $r^*(s) = r_i^\dagger(s)$. However, in D^*, s may partly fulfill that appetite

with points outside of $V_{i..n}(s)$ (and outside $\cup B_{1..i-1}$, of course) which are even closer. These points do not have s as first choice, but they may end up not being claimed by a closer site. Hence, it could also be that $r^*(s) < r_i^\dagger(s)$. For instance, see Figure 6.12.

For the second claim, if $r^*(s)$ is minimum, we are in the worst case for s, because, according to Corollary 6.19, s fulfills the rest of its appetite in $V_{i..n}(s)$ and not outside. □

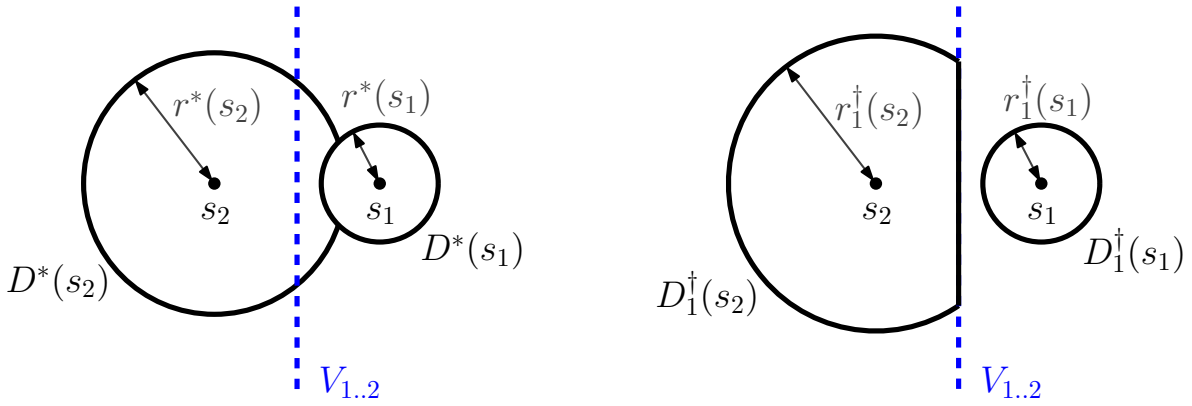


Figure 6.12: An instance of two sites with different appetites. The left side shows the regions of the sought diagram, D^*, and the actual radii r^* of the sites. The right shows the estimate cells and estimate radii of the sites at iteration 1. We can see that $r^*(s_1) = r_1^\dagger(s_1)$ and that $r^*(s_2) < r_1^\dagger(s_2)$.

Corollary 6.21. *At iteration i, if s has a smallest estimate radius $r_i^\dagger(s)$ among all the sites in $S_{i..n}$, then s has a smallest actual radius $r^*(s)$ in D^* among all the sites in $S_{i..n}$.*

Proof. Let x be the value of the smallest actual radius. Then, every site has estimate radius at least x (Lemma 6.20). Further, there are sites with exactly estimate radius x: in particular, those with actual radius x (Lemma 6.20 again). Thus, a site s with smallest estimate radius has estimate radius $r_i^\dagger(s) = x$, and so it must also have actual radius $r^*(s) = x$. □

Corollary 6.21 gives us a way to find the next site s_i in a proper ordering: compute the estimate radii of all the sites, and choose a sites with a smallest estimate radius. To do this, we need to be able to compute the estimate radii $r_i^\dagger(s)$. This is the most challenging step in our algorithm. In fact, Observation 6.15 speaks to its difficulty. To circumvent this problem, we encapsulate

the computation of each $r_i^\dagger(s)$ in a geometric primitive that can be approximated numerically in an algebraic model of computation. For the sake of the algorithm description, we assume the existence of a black-box function that allows us to compute the following geometric primitive.

Definition 6.22 (Geometric primitive)**.** Given a convex polygon P, a point s in P, an appetite A, and a set C of disks, return the radius r (if it exists) such that the area of the intersection of $P \setminus C$ and a disk centered at s with radius r equals A.

In the context of our algorithm, the point s is a site in $S_{i..n}$, the appetite A is the remaining appetite $A_i(s)$ of s, the polygon P is the Voronoi cell $V_{i..n}(s)$, and the set of disks C is $B_{1..i-1}$. Note that such a primitive could be approximated numerically to arbitrary precision with a binary search like the one described later in Section 6.4.2.

Algorithm 12 Stable-matching Voronoi diagram algorithm.

Input: set S of n sites, and the appetite $A(s)$ of each site s.
Initialize $S_{1..n}$ as S, $V_{1..n}$ as a standard Voronoi diagram of S, $B_{1..0}$ as an empty set of disks, $\cup B_{1..0}$ as an empty union of disks, and D_0 as an empty diagram.
For each site $s \in S$, initialize its remaining appetite $A_1(s) = A(s)$.
for $i = 1, \ldots, n$ **do**
  for each site s in $S_{i..n}$ **do**
    Calculate the estimate radius $r_i^\dagger(s)$ and estimate bounding disk $B_i^\dagger(s)$ of s using the primitive from Definition 6.22 with parameters $V_{i..n}(s)$, s, $A_i(s)$, and $B_{1..i-1}$.
  Let s be a site in $S_{i..n}$ whose estimate radius $r_i^\dagger(s)$ is minimum.
  Set $s_i = s$, $r^*(s_i) = r^\dagger(s_i)$, $B^*(s_i) = B^\dagger(s_i)$.
  Compute $\hat{B}(s_i) = B^*(s_i) \setminus \cup B_{1..i-1}$.
  Compute $\hat{V}_{i..n}$ by partitioning $\hat{B}(s_i)$ according to $V_{i..n}$.
  Add $\hat{V}_{i..n}$ to D_{i-1} to obtain D_i.
  for each site s' in $S_{i..n}$ **do**
    Set $A_{i+1}(s') = A_i(s') - area(\hat{V}_{i..n}(s'))$ ($\hat{V}_{i..n}(s')$ might be empty).
  Set $S_{i+1..n} = S_{i..n} \setminus \{s_i\}$ and $B_{1..i} = B_{1..i-1} \cup \{B^*(s_i)\}$.
  Add $B^*(s_i)$ to $\cup B_{1..i-1}$ to obtain $\cup B_{1..i}$.
  Remove s_i from $V_{i..n}$ to obtain $V_{i+1..n}$.
Return D_n.

Implementation and runtime analysis. Given the preceding discussion, Algorithm 12 shows the full pseudocode. It uses the following data structures:

- $V_{i..n}$: the standard Voronoi diagram of n sites has $O(n)$ combinatorial complexity. It can be initially computed in $O(n \log n)$ time (e.g., see [16, 18]). It can be updated after the removal of a site in $O(n)$ time [105].

- $\cup B_{1..i}$: the union of n disks also has $O(n)$ combinatorial complexity [5, 127]. To compute $\cup B_{1..i}$ from $\cup B_{1..i-1}$, a new disk can be added to the union in $O(n \log n)$ time, e.g., with a typical plane sweep algorithm.

- $\hat{V}_{i..n}$: since $\cup B_{1..i-1}$ has $O(n)$ complexity, and the boundary of $B^*(s_i)$ can only intersect each edge of $\cup B_{1..i-1}$ twice, $\hat{B}(s_i)$ also has $O(n)$ complexity. Given that both $\hat{B}(s_i)$ and $V_{i..n}$ have $O(n)$ combinatorial complexity, $\hat{V}_{i..n}$ has $O(n^2)$ combinatorial complexity. The diagram $\hat{V}_{i..n}$ can be computed in $O(n^2 \log n)$ time, e.g., with a typical plane sweep algorithm.

- D_i: we do not maintain the faces of the diagram D_i explicitly as ordered sequences of edges. Instead, for each site s, we simply maintain the region $D_i(s)$ as the (unordered) set of edges $\cup_{1 \le j \le i}\, edges(\hat{V}_{j..n}(s))$. That is, at each iteration i, we add to the edge set of each site s the edges bounding the (possibly empty) region of s in $\hat{V}_{i..n}$. Note that after iteration j, the set of edges of s_j does not change anymore. Since $\hat{V}_{i..n}$ has $O(n^2)$ complexity for any i, the collective size of the these edge sets is $O(n^3)$ throughout the algorithm.

We wait until the end of the algorithm to construct a proper data structure representing the planar subdivision D^*, e.g., a doubly connected edge cell (DCEL) data structure. We construct it from the sets of edges collected during the algorithm. Let $E(s_i) = \cup_{1 \le j \le i}\, edges(\hat{V}_{j..n}(s_i))$ be the set of edges for a site s_i.

Lemma 6.23. *If all the fragments of edges in $E(s_i)$ that overlap with other parts of edges in $E(s_i)$ are removed, then the parts left are precisely the edges of $D^*(s_i)$, perhaps split into multiple parts.*

Proof. Since D^* and D_n are the same, every edge e of $D^*(s_i)$ appears in $E(s_i)$. However, e may not appear as a single edge. Instead, it may be split into multiple edges or fragments of edges of

$E(s_i)$. This may happen when, for some s_j with $j < i$, the boundary of $\hat{B}(s_j)$ intersects e. In this case, in $E(s_i)$, the edge e is split in two at the intersection point. This is because the two parts of the edge are found at different iterations of the algorithm. See, e.g., edge e in Figure 6.13.

However, in $E(s_i)$ there may also be edges or fragments of edges which do not correspond to edges of $D^*(s_i)$. These are edges or fragments of edges that actually lie in the interior of $D^*(s_i)$, but that are added to $E(s_i)$ because they lie along the boundary of $\hat{B}(s_j)$ for some s_j with $j < i$, which makes the region of $D^*(s_i)$ be split along that boundary. Such edges appear exactly twice in $E(s_i)$: one for each face on each side of the split. See, e.g., the edges that lie in the interior of $D^*(s_4)$ in Figure 6.13, and note that they are all colored twice (unlike the actual edges of $D^*(s_i)$). □

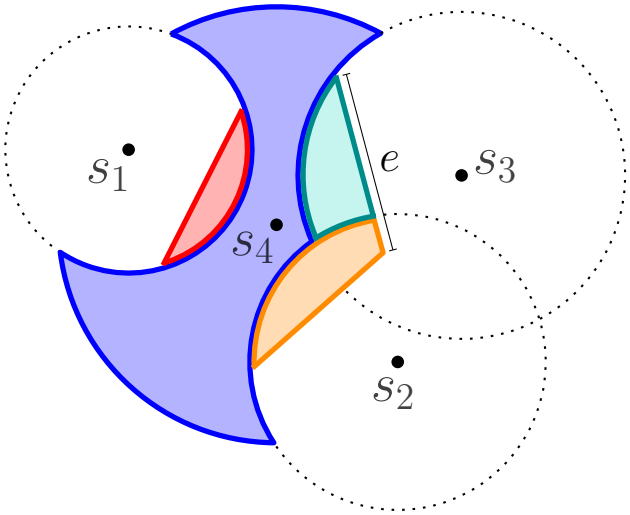


Figure 6.13: The union of colored regions is the stable cell $D^*(s_4)$ of the site s_4. The algorithm finds it divided into four regions, $\hat{V}_{i..4}(s_4)$ for $i = 1, 2, 3, 4$, shown in different colors. The bounding disks of s_1, s_2, and s_3, are hinted in dotted lines. The edge e of $D^*(s_4)$, which lies along the perpendicular bisector between s_3 and s_4, is split between $\hat{V}_{2..4}(s_4)$ and $\hat{V}_{3..4}(s_4)$.

Given Lemma 6.23, we can construct $D^*(s_i)$ from $E(s_i)$ as follows: first, remove all the overlapping fragments of edges in $E(s_i)$. Second, connect the edges with matching endpoints to construct the faces. While doing this, if two straight edges that lie on the same line share an endpoint, merge them into a single edge. Similarly, merge any two curved edges that lie along the same arc and share an endpoint. These are the fragmented edges of $D^*(s_i)$.

Each of these steps can be done with a typical plane sweep algorithm. In more detail, this could be done as follows: sort the endpoints of edges in $E(s_i)$ from left to right. Then, process the edges in the order encountered by a vertical line that sweeps the plane from left to right. Maintain all the edges intersecting the sweep line, ordered by height of the intersection (e.g., in a balanced binary search tree). In this way, overlapping edges (for the first step) or edges with a shared endpoint (for the second step) can be found quickly in $O(\log n)$ time. Construct the faces of $D^*(s_i)$ as they are passed by the sweep line.

Since the sets $E(s_i)$, for $1 \leq i \leq n$, have $O(n^3)$ cumulative combinatorial complexity, sorting all the $E(s_i)$ sets can be done in $O(n^3 \log n)$ time. The plane sweeps for all the s_i have overall $O(n^3)$ events, each of which can be handled in $O(\log n)$ time. Thus, the algorithm takes $O(n^3 \log n)$ time in total.

Theorem 6.24. *The stable-matching Voronoi diagram of a set S of n point sites can be computed in the real-RAM model in $O(n^3 \log n)$ time plus $O(n^2)$ calls to a geometric primitive that has input complexity $O(n)$.*

Proof. For the number of calls to the geometric primitive, note that there are n iterations, and at each iteration we call the geometric primitive $O(n)$ times. Any given cell of the standard Voronoi diagram $V_{i..n}$ has $O(n)$ edges, and there are $O(n)$ already-matched disks, so the input of each call has $O(n)$ size. Therefore, we make $O(n^2)$ calls to the geometric primitive, each of which has combinatorial complexity $O(n)$.

Besides primitive calls, the bottleneck of each iteration i is computing $\hat{V}_{i..n}$. This can be done in $O(n^2 \log n)$ time, for a total of $O(n^3 \log n)$ time over all the iterations. The final step of reconstructing D^* can also can be done in $O(n^3 \log n)$. □

6.4.2 Polygonal convex distance functions

In this section, we show how to implement the geometric primitive *exactly* for convex distance functions induced by convex polygons. The use of this class of metrics for Voronoi Diagrams was introduced in [51], and studied further, e.g., in [142]. Intuitively, the polygonal convex distance function, $d_S(a, b)$, induced by a convex polygon S, is the factor by which we need to scale S, when S is centered at a, to reach b. Solving the primitive exactly for such metrics is interesting for two reasons. First, this class of distance functions includes many commonly used metrics such as the L_1 (Manhattan) and L_∞ (Chebyshev) distances. Second, a convex distance function induced by a regular polygon with a large number of sides can be used to approximate Euclidean distance.

Definition 6.25 (Polygonal convex distance functions)**.** Let S be a convex polygon in $\mathbb{R}^2$ that contains the origin. The distance $d_S(a, b)$ from point a to point b is calculated as follows: we translate S by vector a so that a is at the same place inside S as the origin was. Let p be the point at the intersection of S with the ray starting at a in the direction of b. Then, $d_S(a, b) = d(a, b)/d(a, p)$ (where $d(\cdot, \cdot)$ is Euclidean distance).

Convex distance functions satisfy triangle inequality, but they may not be symmetric. Symmetry ($d_S(p, q) = d_S(q, p)$) holds if and only if S is symmetric with respect to the origin [142]. In this section, we assume that S is symmetric with respect to the origin. Another significant difference with Euclidean distance is that the bisector of two points may contain 2-dimensional regions. This happens when the line through the two points is parallel to a side of S [142]. We assume that such degenerates cases do not happen.[16]

The discussion from Section 6.2 applies to diagrams based on polygonal convex distance functions. However, in this setting, all the edges are straight. Recall that, in the Euclidean distance setting, straight edges lie along perpendicular bisectors, while curved edges lie along the boundaries of bounding disks. This is still the case here, but bounding disks are constituted of straight edges. In

[16] Alternatively, we may redefine the bisector to go along the clockwise-most boundary of the two-dimensional region, as in [51].

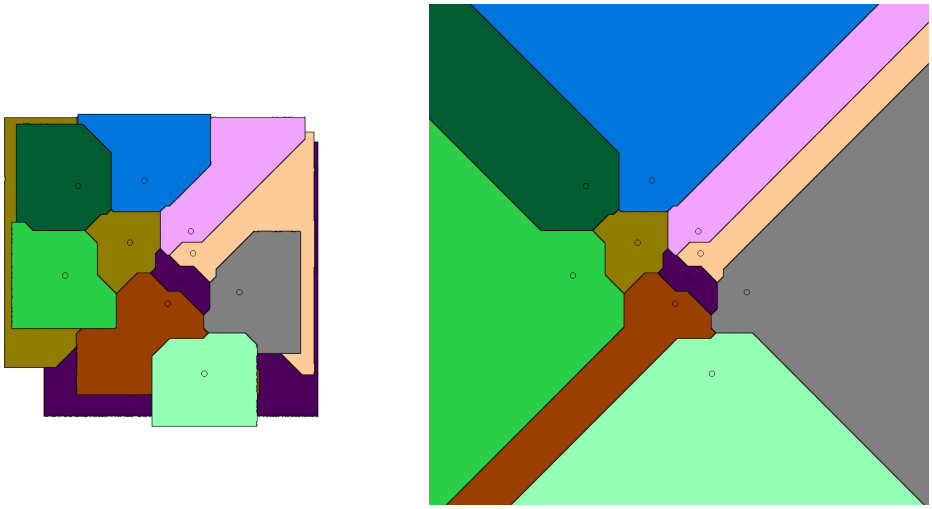


Figure 6.14: Stable-matching Voronoi diagram (left) and standard Voronoi diagram (clipped by a square) (right) for the convex distance function induced by a square centered at the origin, which corresponds to the L_∞ metric.

this context, disks are called balls. A *ball* is a (closed) region bounded by a translated copy of S scaled by some factor. Therefore, straight and curved edges should now be referred to as *bisector edges* and *bounding ball edges*, respectively. With this distinction, the results in that section also apply. Likewise, the algorithm applies as well. However, note that the notion of radius is not well defined for convex distance functions, as they grow at different rates in different directions. Therefore, instead of talking about the radii of the bounding disks, we should talk about the scaling factor of the bounding balls. Most importantly, the fact that there are no curved edges allows us to compute the diagram exactly in an algebraic model of computation. This is the focus of this section.

We need to reformulate the geometric primitive for the case of convex distance functions. Recall that the polygon P in the primitive should correspond to a Voronoi cell, which is the reason why P is assumed to be convex in the primitive. However, Voronoi cells may not be convex for convex distance functions (see Figure 6.14). Instead, Voronoi cells of polygonal convex distance functions are star-shaped, with the site in the kernel [142]. Thus, P will now be a star-shaped polygon. For simplicity, we translate the site s to the origin. Finally, we express the solution as the scaling factor of the wanted ball rather than its radius.

Definition 6.26 (Geometric primitive for polygonal convex distance functions)**.** Given a convex distance function induced by a polygon S symmetric with respect to the origin, a star-shaped polygon P with the origin in the kernel, an appetite A, and a set C of balls, return the scaling factor r (if it exists) such that A equals the area of the intersection of $P \setminus C$ and S scaled by r.

The algorithm.

1. The algorithm begins by computing $P \setminus C$ (which is a polygonal shape that can be concave, have holes, and be disconnected). Then, we triangulate $P \setminus C$ into a triangulation, T_1. For each triangle in T_1 whose interior intersects one of the *spokes* of S (a ray starting at the origin and going through a vertex of S), we divide the triangle along the spoke and re-triangulate each part. After this, the resulting triangulation, T_2, has no triangles intersecting any spoke of S except along the boundaries (see Figure 6.15).

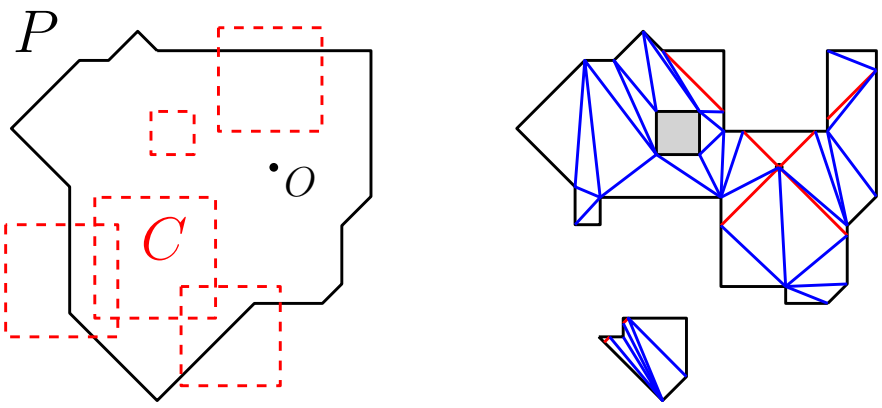


Figure 6.15: Left: an input to the geometric primitive for the convex distance function induced by a square, where the balls in C are shown in dashed red lines. Right: the corresponding triangulation T_2 of $P \setminus C$ where no triangle intersects any spoke of S (shown in red, they are also part of the triangulation).

2. The next step is to narrow down the range of possible values of r. We compute, for each vertex v in T_2, the distance from the origin $d_S(O, v)$, and sort the vertices from shortest to longest distance. If two or more vertices are at the same distance, we discard all but one, so that we have a sorted list L with only one vertex for each distance. Now, we search for two

consecutive vertices v_1 and v_2 in L such that $d_S(O, v_1) \leq r \leq d_S(O, v_2)$ (or conclude that r does not exist). To find v_1 and v_2, we can use binary search on the list L: for a vertex v, we compute the area of the intersection of $P \setminus C$ and a ball centered at the origin passing through v (this can be done by adding the individual contribution of each triangle in T_2). By comparing this area to A, we discern whether v is too close or too far.

3. It remains to pinpoint r between $d_S(O, v_1)$ and $d_S(O, v_2)$. Let B_1 and B_2 denote unit balls centered at the origin scaled by $d_S(O, v_1)$ and $d_S(O, v_2)$, respectively, and B the annulus defined by $B_2 \setminus B_1$. Note that, because v_1 and v_2 are consecutive vertices of L, the interior of B does not contain any vertex of T_2. Conversely, no vertex of B is in the interior of a triangle of T_2, because all the vertices of B lie along the spokes of S, and no triangle in T_2 intersects the spokes of S. As a result, if a triangle in T_2 intersects B, the intersection is either a triangle or a trapezoid (see Figure 6.16). Similarly to Step 1, for each triangle in T_2 whose interior is intersected by B_1 and/or B_2, we divide the triangle along B_1 and/or B_2 and re-triangulate each part. Figure 6.17 illustrates the resulting triangulation, T_3, where the interior of each triangle is either fully contained in B or disjoint from B. Moreover, all the triangles in B have an edge along the boundary of B_1 or B_2, which we call the base, and a vertex in the boundary of the other (the cuspid).

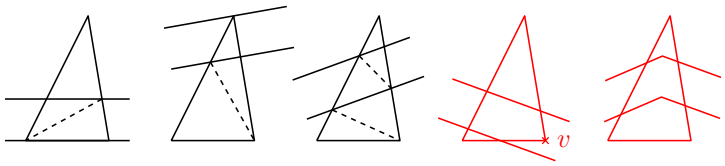


Figure 6.16: In black: three possible intersections of triangles in T_2 and B, and the resulting sub-triangulations. In red: two invalid intersections between a triangle in T_2 and B.

4. Finally, we find r as follows. Since r is between $d_S(O, v_1)$ and $d_S(O, v_2)$, triangles outside B_2 lie outside the ball with radius r. Conversely, all triangles inside B_1 are contained in the ball with radius r. Let A' be the sum of the areas of all the triangles inside B_1. Then, the triangles in B must contribute a total area of $A - A'$. They all have height $h = d_S(O, v_2) - d_S(O, v_1)$. Let R_1 and R_2 be the sets of triangles in B with the base along B_1 and B_2,

respectively. We need to find the height h', with $0 \leq h' \leq h$, such that $A - A'$ equals the sum of *(i)* the areas of the triangles in R_1 from the base to a line parallel to the base at height h', and *(ii)* the areas of the triangles in R_2 from the cuspid to a line parallel to the base at height $h - h'$. Given h', we can output $r = d_S(O, v_1) + h'$.

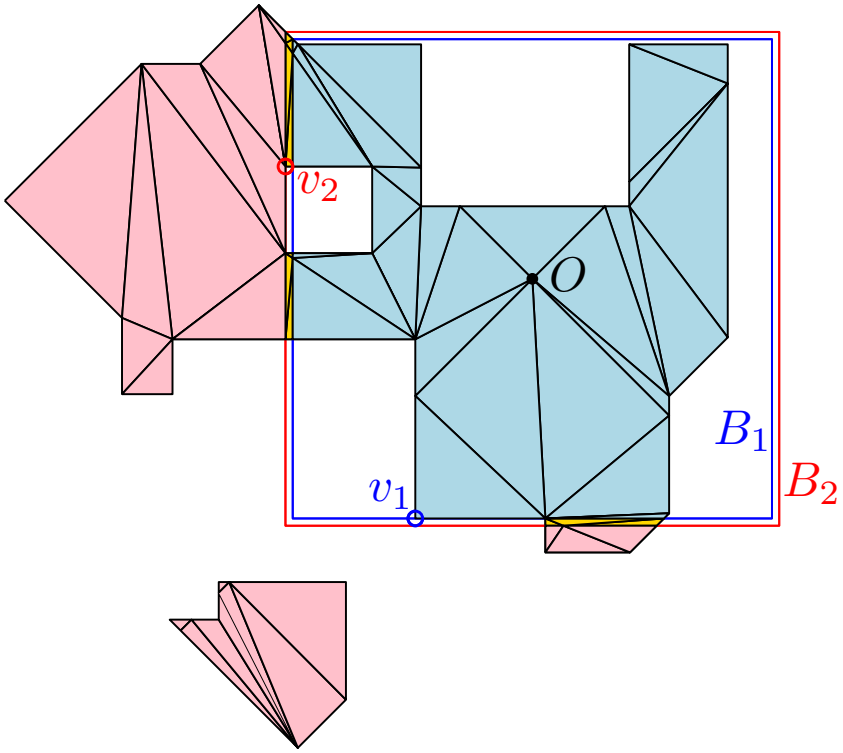


Figure 6.17: Triangulation T_3 of $P \setminus C$ after Step 3 of the algorithm, where no triangle intersects B. The triangles of T_3 can be classified into those inside B_1, inside B, and outside B_2.

In order to find h', we rearrange the triangles to combine them into a trapezoid, as shown in Figure 6.18, Left. We rotate the triangles in R_1 to align their bases, translate them to put their bases adjacent along a line, and shift their cuspids along a line parallel to the bases to coincide at a single point above the leftmost point of the first base. Doing so does not change their area, and guarantees that triangles do not overlap. We do a similar but flipped transformation to triangles in R_2 in order to form the trapezoid. The height h' is the height at which the area of the trapezoid from the base up to that height is $A - A'$, which can be found as the solution to a quadratic equation by using the formula for the area of a trapezoid, as shown in Figure 6.18, Right.

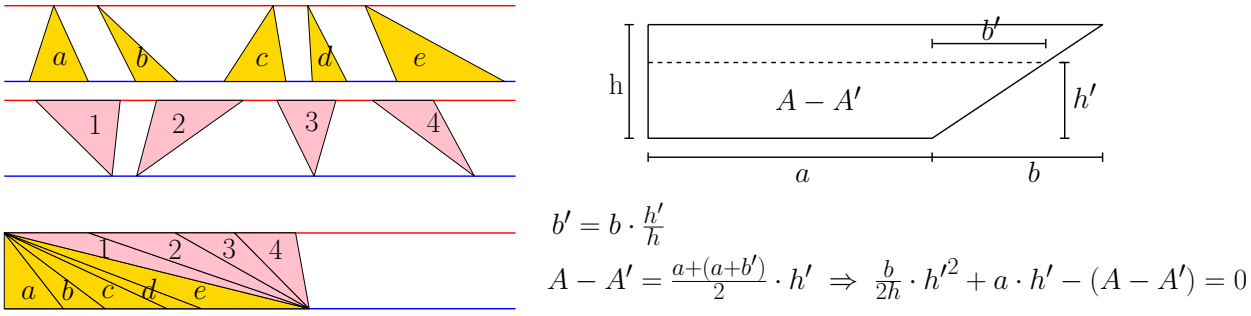


Figure 6.18: Top left: triangles of T_3 inside B, rotated and separated into triangles with the base along B_1 (top) and B_2 (bottom). Bottom left: the triangles rearranged (and transformed) into a trapezoid with the same area. Right: derivation of the quadratic equation for h' from the formula for the area of a trapezoid, for the case where the base is shorter than the top size (i.e., b is positive). Note that a, b, and h are known. The alternative case is similar.

Running time. The correctness of the algorithm follows from the simple argument in Step 4. We now consider its runtime analysis. The size of the input to this primitive is $O(|P|+|C|+|S|)$, where $|P|$ and $|S|$ denote the the number of edges of the polygons P and S, respectively. The polygonal shape $P \setminus C$ has $O(|P|+|C||S|)$ edges, as each ball in C has $|S|$ edges. The corresponding triangulation T_1 has $O(|P|+|C||S|)$ triangles. Each spoke of S may intersect every triangle and divide it in two or three, so T_2 has $|T_2| = O(|P||S|+|C||S|^2)$ triangles (and vertices). Sorting the vertices of T_2 requires $O(|T_2| \log |T_2|)$ time. The binary search has $O(\log |T_2|)$ steps, each of which takes time proportional to the number of triangles, $O(|T_2|)$. These steps are the bottleneck, as T_3 grows only by a constant factor with respect to T_2. Thus, the total runtime of the primitive is $O(|T_2| \log |T_2|) = O((|P||S|+|C||S|^2) \log (|P||S|+|C||S|))$.

In the context of the algorithm, we make calls with to the primitive with $|P| = O(n|S|)$ and $|C| < n$, so we can compute the primitive in $O(n|S|^2 \log(n|S|))$ time. When the polygon S has a constant number of faces, the time is $O(n \log n)$. Thus, the entire stable-matching Voronoi diagram for metrics based on these polygons can be computed in $O(n^3 \log n)$ total time. This includes the metrics L_1 and L_∞.

6.4.3 Discretized plane

Another option is to discretize the plane into a grid of $m \times m$ pixels. Then, we can find a stable matching between the n sites (with appropriate quotas) and the $m^2 \gg n$ pixels (in the version of the college admission problem where some of the pixels may remain unmatched). Larger values of m results in a better approximation to the discrete diagram. This is the geometric stable matching model discussed in Section 4.3. Using a bichromatic closest pair data structure or the first-choice chain algorithm, we can compute the diagram in $O(m^2 \log^4 m)$ time. Alternatively, the Gale–Shapley algorithm requires $O(m^2 n \log m)$ time, where the bottleneck is to sort the distances from each site to every pixel and vice-versa, which is needed to generate the preference lists.

Section 7.3 discusses practical optimizations for this discretized setting.

6.5 Conclusions

We have studied stable-matching Voronoi diagrams, providing characterizations of their combinatorial complexity and a first discrete algorithm for constructing them. Stable-matching Voronoi diagrams are a natural generalization of standard Voronoi diagrams to size-constrained regions. This is because standard Voronoi diagrams also have the defining property of stable-matching Voronoi diagrams: stability for preferences based on proximity. Furthermore, both have similar geometric constructions in terms of the lower envelopes of cones.

However, allowing prescribed region sizes comes at the cost of convexity and connectivity; indeed, we have shown that a stable-matching Voronoi diagram may have $O(n^{2+\varepsilon})$ faces and edges, for any $\varepsilon > 0$. We conjecture that $O(n^2)$ is the right upper bound, matching the lower bound that we have given.

Constructing a stable-matching Voronoi diagram is also more computationally challenging than the construction of a standard Voronoi diagram. In particular, it requires computations that cannot be carried out exactly in an algebraic model of computation. We have given an algorithm which runs in $O(n^3 \log n + n^2 f(n))$-time, where $f(n)$ is the runtime of a geometric primitive that we defined to encapsulate the computations that cannot be carried out analytically. While such primitives cannot be avoided, a step forward from our algorithm would be one that relies only in primitives with constant-sized inputs.

We propose three approaches for computing stable-matching Voronoi diagrams, each of which requires a different compromise: *(a)* use our algorithm and approximate the geometric primitive numerically; *(b)* replace the Euclidean distance by a polygonal convex distance function induced by a regular polygon with many faces (this approximates a circle, which would correspond to Euclidean distance), and compute the primitive exactly as described in Section 6.4.2; *(c)* discretize the plane into a grid, and use a bichromatic closest pair data structure or the first-choice chain algorithm from Section 4.3.

Chapter 7

Stable Redistricting

7.1 Introduction

Location analysis [164] is a classical branch of optimization in geographic information systems concerned both with *facility location*, the placement of facilities such as polling places, fire stations, or post offices in a geographic region, and the *assignment problem*, the problem of partitioning the territory into service regions for these facilities. All points in the territory should be equitably served by nearby facilities and each facility should bear a fair portion of the total service load.

A problems in this category is *political redistricting* (e.g., see [60, 154, 165]). The goal of political redistricting is to partition a territory into districts with roughly the same population size and with "compact" shapes [175]. Highly non-compact districts drawn by cartographers have been the subject of legal cases involving *partisan gerrymandering*, the manipulation of district boundaries for political advantage [143, 175]. In political systems where each district gets a representative determined by majority vote, gerrymandering aims to maximize the number of representatives for a party. It is achieved via two complementary means. *Cracking* consists of forming districts where

the favored party wins by a small margin, and *packing* consists of forming districts where the opposing party wins by a large majority.

Purely geometric "politically-agnostic" algorithms have two potential uses in this context. First, they can be used to analyze the fairness of districts. While population balance is straightforward to assess, there are many competing and conflicting notions of compactness, such as the ratio between the perimeter and the area of the district, or the ratio between the area of the district and the area of its smallest enclosing disk. Second, algorithms can be used to draw districts directly. Ricca et al. [165] adapted the concept of Voronoi diagrams to use them for political districting. Voronoi regions ensure great compactness but not as good population balance, however. Thus, there is motivation for other approaches that guarantee better population balance.

In Chapter 6, we discussed stable-matching Voronoi diagrams, a generalization of Voronoi diagrams which allow us to ascribe region sizes to the sites. This is ideal for ensuring population balance, so we propose the use of these diagrams for districting. We consider that the population wants to have a nearby district center/representative, and, conversely, that district centers want to represent the local population. When the center locations are fixed to begin with, this gives rise to a stable matching problem between the population and the district centers, where preferences are determined by proximity. Thus, stability is our new notion of compactness. We introduced stable matchings in Chapter 4. Each district center has a *quota* indicating its capacity. The use of quotas allows each district to serve a different amount of people, but we focus on the case where all the centers have equal quota.

We consider two settings. In Section 7.2, we model the geographic space of interest as a graph representing a road network. In Section 7.3, we model it as a grid in the plane. In each case, we consider the problem of finding a stable matching. These problems are special cases of the geographic and geometric models that we studied in Chapter 4.

While quotas allow us to enforce that all districts have the same size, the resulting stable districts are not necessarily convex or even connected. Finding a scheme that guarantees both size equality and compactness for fixed centers is an open problem of interest. In Section 7.4, we tackle the problem of *facility location*, that is, how to best position the centers in the first place. Our proposed solution is to integrate a stable matching algorithm into the k-*means* clustering method. This improves the convexity of the regions and nearly achieves connectivity.

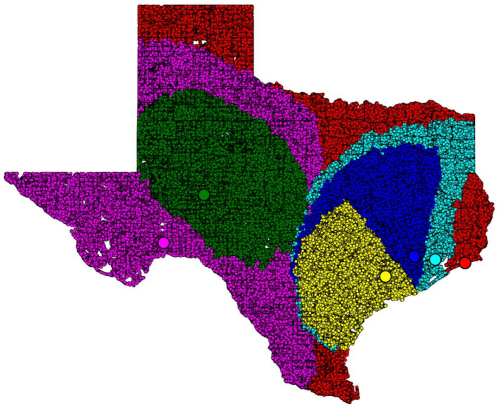
7.2 Geographic setting

A road network can be represented by a graph where each node is an intersection, and each edge is a stretch of road between two intersections. As discussed in Section 5.1.1, we model road networks as embedded graphs with *sparse crossing graphs* (see the definition in Section 5.1.1), which have been shown to be a better model for road networks than, e.g., planar graphs [87].

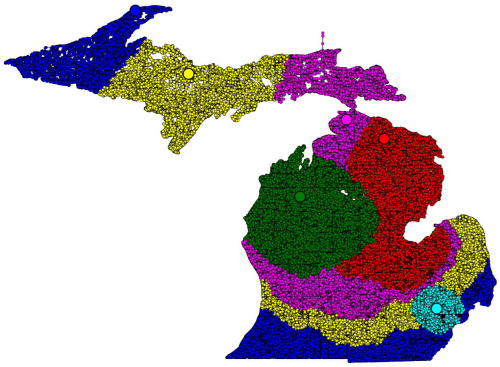
Problem 16 (Geographic stable districting)**.** Given a connected, undirected, n-node graph $G = (V, E)$ with positively-weighted edges, a plane embedding of G with a sparse crossing graph, and a subset P of V of k nodes with equal quotas (up to round-off errors) adding up to n, find a stable matching between V and P, where a node p prefers q over q' if and only if $d(p, q) < d(p, q')$, and d denotes shortest-path distance.

This is a special case of geographic stable matching (Problem 14 in Section 4.4), but in the one-to-many setting. As mentioned there, these preferences are symmetric, so there is a unique stable matching. Figure 7.1 shows the districts obtained in real road networks.

We consider three algorithms for this problem and provide an experimental comparison. First, we consider the classic Gale–Shapley algorithm. We mentioned in Section 4.4 that, in this setting, the bottleneck of the Gale–Shapley algorithm is to compute the preferences, that is, the shortest-

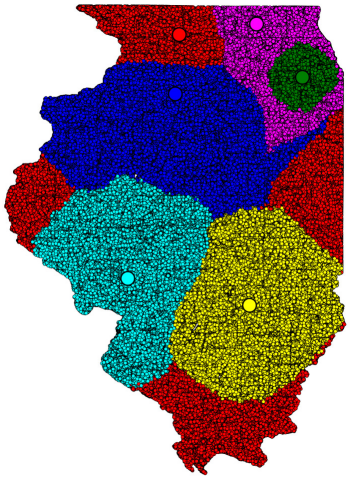


Texas ($n = 2037K, m = 2550K$)



Michigan ($n = 662K, m = 833K$)

Washington, DC ($n = 9522, m = 14850$)



Illinois ($n = 790K, m = 1008K$)

Figure 7.1: The solutions to the geographic stable districting problem for the 2010 road networks of three U.S. states and the District of Columbia, from the DIMACS database [68]. They consists of primary and secondary roads in the biggest connected component of the road networks. In each case, n and m denote the number of nodes and edges, respectively, and there are $k = 6$ randomly-selected centers with equal quota n/k.

path distances between every center and node. This can be done in $O(kn \log n)$ time, e.g., with Dijkstra's algorithm.

Second, we consider the first-choice chain algorithm from Chapter 4 (Theorem 4.7) paired with the reactive nearest-neighbor data structure from Chapter 5 (Theorem 5.4). Since graphs with sparse crossing graphs have $O(n^{0.5})$-size separators that can be computed in linear time [87], this algorithm runs in $O(n^{1.5} \log n)$ time.

Finally, we consider a practical algorithm based on a simulation of the circle-growing method of Hoffman et al. [113] for stable-matching Voronoi diagrams, which we described in Section 6.1.

7.2.1 Circle-growing algorithm

The circle-growing algorithm can be visualized as a process in which we grow circles from each center, all at the same speed, and match each node to the first circle that grows across it.

We start k instances of Dijkstra's algorithm at the same time, one from each center. We explore, at each step, the next closest node to any of the centers, advancing one of the instances of Dijkstra's algorithm by a single step. We match each node to the center whose instance of Dijkstra's algorithm reaches it first. Note that when an instance of Dijkstra's algorithm, starting from center c, reaches a node x that has not already been matched, then c and x must be the global closest pair (omitting already matched pairs). We halt each instance of Dijkstra's algorithm as soon as its center reaches its quota. This stopping condition prevents wasted work in which an instance of Dijkstra's algorithm explores nodes farther than its farthest matched node. In addition, using this method to solve symmetric stable matching problems allows us to avoid running the Gale–Shapley algorithm afterwards.

There are several alternative ways to implementing the parallel instances of Dijkstra's algorithm with a runtime of $O(kn \log n)$. For instance, we can use a priority queue of centers to decide which

instance of Dijkstra's algorithm should advance at each step, or we can merge the priority queues of all the instances of Dijkstra's algorithm into a single larger priority queue.

7.2.2 Experiments

In this section we present an empirical comparison of the Gale–Shapley algorithm, circle-growing algorithm, and first-choice chain algorithm on real road network data. Figure 7.2 and its associated Table 7.1 illustrate the main findings.

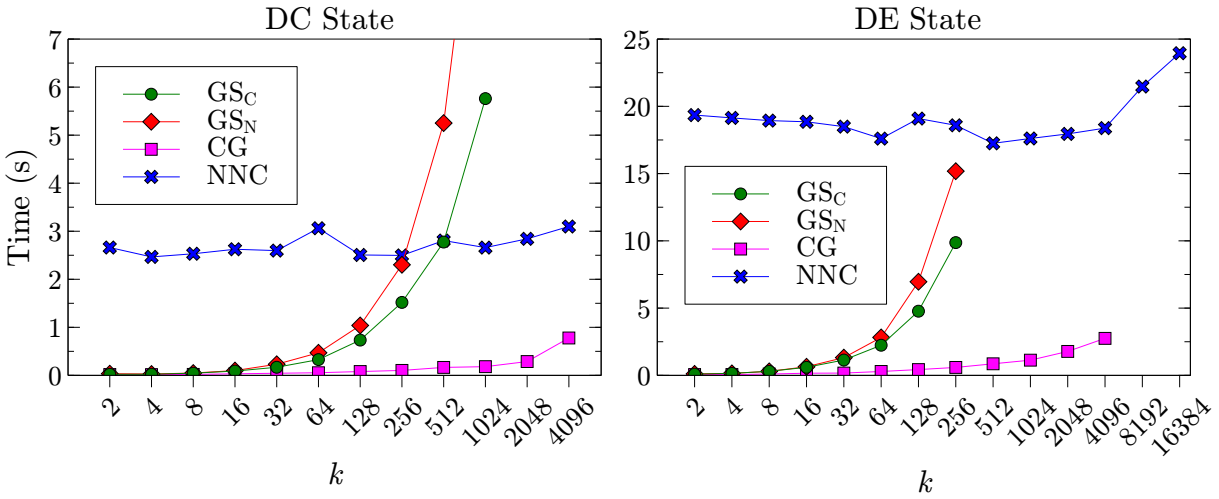


Figure 7.2: Comparison of the running time of the algorithms in the Washington, DC (left, $n = 9522, m = 14850$) and Delaware (right, $n = 48812, m = 60027$) road networks from the DIMACS database [68] for a range of number of centers k (in a logarithmic scale). Each data point is the average of 10 runs with 10 sets of random centers (the same sets for all the algorithms). The first-choice chain algorithm is labeled NNC instead of FCC.

Experiment setup. We implemented the various symmetric stable matching algorithms of our comparison in Java 8.[17] We then executed them and timed them as run on an Intel(R) Core(TM) CPU i7-3537U 2.00GHz with 4GB of RAM, on Windows 10.

In the table and figures presenting our experimental results, we use the label CG for the circle-growing algorithm and FCC for the first-choice chain algorithm. For the Gale–Shapley algorithm,

[17]The source code is available at `https://github.com/nmamano/StableDistricting`.

Table 7.1: Runtime in seconds of the algorithms in the Delaware road network ($n = 48812, m = 60027$). Each data point is the average of 10 runs with 10 sets of random centers (the same sets for all the algorithms). A dash indicates that the algorithm ran out of memory.

k	GS_N	GS_C	CG	FCC
2	0.11	0.09	0.06	19.35
4	0.15	0.15	0.06	19.14
8	0.30	0.30	0.10	18.94
16	0.64	0.60	0.16	18.85
32	1.32	1.14	0.17	18.49
64	2.82	2.24	0.29	17.60
128	6.96	4.77	0.43	19.09
256	15.18	9.87	0.59	18.59
512	—	—	0.86	17.25
1024	—	—	1.13	17.61
2048	—	—	1.78	17.95
4096	—	—	2.75	18.38
8192	—	—	—	21.47
16384	—	—	—	23.95

we consider a variation GS_C where the centers do the proposals (which corresponds to the role of the men in the original algorithm), and the alternative GS_N where the nodes do the proposals. In addition, we used Dijkstra's algorithm to compute the distances. For the first-choice chain algorithm, we implemented and used the reactive nearest-neighbor data structure from Chapter 5.

Results. Figure 7.2 shows a clear picture of the respective algorithms' strengths and weaknesses:

- The Gale–Shapley algorithm, with a runtime of $O(kn \log n)$, scales linearly with k. Moreover, because of the memory requirement of $\Theta(nk)$, we could not run it with large numbers of centers. The version of Gale–Shapley where nodes propose (GS_N) was about 50% slower than the version where centers propose. This is explained by the fact that, when nodes propose, each center needs to keep track of its least preferred already-matched node. This node may need to be rejected if the center receives a preferable proposition from another node. We maintain these least-preferred matched nodes by using a binary heap of nodes for each center; however, the overhead of maintaining this heap adds to the running time of our im-

plementation. In contrast, when centers propose, each node needs to keep track only of a single match, so we do not need to use an additional binary heap for this purpose.

- The circle-growing algorithm was the fastest of our implemented algorithms in practice, over the range of values of k for which we could run it. It is also the only algorithm that could complete a solution for the largest road networks that we tested. For instance, on the Texas road network, which has over 2 million nodes, the algorithm finishes in 3 seconds when given 6 random centers; our other implementations could not solve instances this large. We did not see significant differences in the runtime between different ways to implement the parallel instances of Dijkstra's algorithm.

- Additionally, in contrast to the Gale–Shapley algorithm, the runtime of circle-growing did not appear to be strongly affected by the value of k. The reason for this is that, even though the algorithm runs k instances of Dijkstra's algorithm, the expected number of nodes that each instance explores decreases as k increases. However, this phenomenon may only be valid in expectation with randomly located centers.

- Our first-choice chain algorithm, with a runtime of $O(n\sqrt{n}\log n)$, is the only one with a runtime independent of k. Hence, it has a flat curve in the plots[18]. The Gale–Shapley curve and the first-choice chain curve cross in our experimental data at around $k \approx 4\sqrt{n}$, showing that the constant factors in our implementation of the first-choice chain algorithm are reasonable. Moreover, because of its memory requirement of $O(n\sqrt{n})$, the first-choice chain algorithm is the only algorithm that was able to complete a solution for the entire range of values of k on all networks that were small enough for it to run at all. In comparison, in the Delaware road network, the Gale–Shapley algorithm ran out of memory at $k = 256$, and the circle-growing algorithm ran out of memory at $k = 8192$.

[18]Even though the runtime of FCC seems to start to increase for the largest values of k in the Delaware plot, this seems to be due to an external hardware/software issue, as other simulations of this comparison did not show this.

7.3 Grid Setting

In this section, we consider that the space of interest for the assignment problem is a grid in the plane. A long line of research considers algorithms on objects embedded in $n \times n$ grids, including problems in computational geometry (e.g., see [9, 13, 89, 106, 128, 157, 158]), graph drawing (e.g., see [28,53,65,163]), geographic information systems (e.g., see [64]), and geometric image processing (e.g., see [47,54,66,109]). Continuing this line of work, we consider the problem of matching grid points (which we view as *pixels*) to k *center* points.

Problem 17 (Stable grid districting)**.** Find a stable matching between the points in the integer $\{1, \ldots, n\} \times \{1, \ldots, n\}$ grid and k points with equal quotas (up to round-off errors) adding up to n^2, where a point p prefers q over q' if and only if $d(p, q) < d(p, q')$, and distances are measures under some L_p metric.

This is a special case of geometric stable matching (Problem 13 in Section 4.3), but in the one-to-many setting. As mentioned there, these preferences are symmetric, so there is a unique stable matching. The centers are not required to be grid points, but we assume that they are within the grid, i.e., their coordinates are in the range $[0, n]$. This problem is the same as computing the stable-matching Voronoi diagram from Chapter 6 in a discretized plane (Section 6.4.3). The only difference is that every pixel of the grid must be matched. Thus, the practical algorithms that we describe here can also be used for computing stable-matching Voronoi diagrams.

Figure 7.3 illustrates a solution for a 900×900 grid and 100 random centers. Note that some centers are matched to disconnected regions.

In this setting, the first-choice chain algorithm from Chapter 4 runs in $O(n^2 \log^4 n)$ (for Euclidean distance), and the Gale–Shapley algorithm runs in $O(n^2 k)$ time after sorting the preference lists. We propose two practical algorithms with better performance.

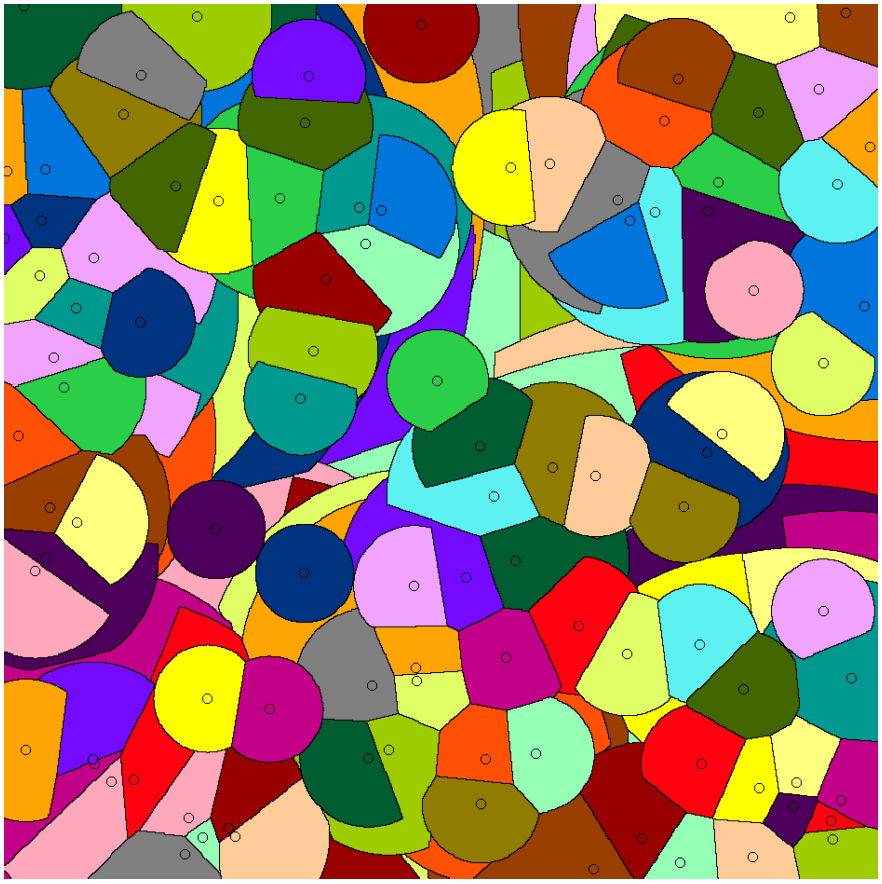


Figure 7.3: An example solution to the *stable grid districting* problem for a 900×900 grid and 100 randomly distributed centers. Each center has a color, and pixels are colored according to their assigned center. Some centers have repeated colors.

The first one is also an adaptation of the circle-growing algorithm. The second one is a new algorithm specific to this setting, the *distance-sorting* algorithm. We provide an experimental analysis of these two algorithms, where we observe that the circle-growing algorithm is faster at matching nearby pairs, while the distance-sorting algorithm is more efficient when pairs are farther apart. Therefore, we show that it is advantageous to switch from one algorithm to the other partway through the matching process. We experiment with the optimal switching point between these two algorithms.

7.3.1 Circle-growing algorithm

All our stable grid districting algorithms start with an empty matching and add center–pixel pairs to it. Given a partial matching, we say a pixel is *available* if it has not been matched yet, and a center is *available* if the size of its region is smaller than its quota. A center–pixel pair is available if both the center and pixel are available, and it is a *closest available pair* if it is available and the distance from the center to the pixel is minimum among all available pairs.

In this section we describe the main practical algorithm, the *circle-growing algorithm*, which mimics the continuous construction from [113]. It implements global greedy: we start with an empty matching and we only add closest available pairs to it until it is complete.

First, we obtain the list of all the lattice points with coordinates ranging from $-n$ to n sorted by distance to the origin. The resulting list P emulates a circle growing from the origin. When initializing P, we can gain a factor of eight savings in space by sorting and storing only the points in the triangle $\triangle(0,0)(0,n)(n,n)$. The remaining points can be obtained by symmetry: if $p = (x, y)$ is a point in the triangle, the eight points with coordinates of the form $(\pm x, \pm y)$ and $(\pm y, \pm x)$ are at the same distance from the origin as p. Moreover, in applications where we find multiple stable matchings, such as in the stable k-means method, we only need to initialize P once. The way we use P depends on the type of centers we consider.

Integer centers (Algorithm 13). In this case we can use the fact that if we relocate the points in P relative to a center, then they are in the order in which a circle growing from that center would reach them. To respect that all the circles grow at the same rate, we iterate through the points in P in order. For each point p, we relocate it relative to each center c to form the pixel $p + c$ (the order of the centers does not matter). We add to the matching any available center–pixel pair $(c, p + c)$. We iterate through P until the matching is complete.

We require $O(n^2)$ space and $O(n^2 \log n)$ time to sort the points in P. For the Euclidean metric instead of using distances to sort P we can use squared distances, which take integer values between 0 and $2n^2$. Then, we can use an integer sorting algorithm such as counting sort to sort in $O(n^2)$ time [61, Chapter 8.2]. Since each point in P results in up to $O(k)$ center–pixel pairs, we need $O(n^2 k)$ time to iterate through P.

Algorithm 13 Circle-growing algorithm for k integer centers on an $n \times n$ grid.

```
Set all pixels as unmatched.
Set the quota of the first n^2 mod k centers to ⌈n^2/k⌉.
Set the quota of the remaining centers to ⌊n^2/k⌋.
Let P = list of points (x, y) such that −n < x, y < n.
Sort P by non-decreasing distance to (0, 0).
for all p ∈ P do until the matching is complete
    for all centers c with quota > 0 do
        s ← p + c
        if 0 ≤ s_x, s_y < n and s is still available then
            Match s and c.
            Reduce the quota of c by 1.
```

Real centers (Algorithm 14). If centers have real coordinates, we cannot translate the points in P relative to the centers, because $p + c$ is not necessarily a lattice point. The workaround is to associate each center c to its closest lattice point p_c. Let δ be the maximum distance $d(c, p_c)$ among all centers. Then, the center–pixel pairs "generated" by each point p in P have the form $(c, p + p_c)$ and their distances can vary between $d(p, O) - \delta$ and $d(p, O) + \delta$ (where O denotes the origin, $(0, 0)$). Consequently, the distances of pairs generated by points p_i, p_j in P with $i < j$ may intertwine, but only if $d(p_j, O) - \delta \leq d(p_i, O) + \delta$. The points in P after p_i whose pairs might intertwine with those of p_i form an annulus centered at O with small radius $d(p_i, O)$ and big radius $d(p_i, O) + 2\delta$ (see Figure 7.4).

Since δ is a constant (for the Euclidean metric, $\delta \leq \sqrt{2}/4$), it can be derived from the Gauss circle problem that such an annulus contains $O(d(p_i, O)) = O(n)$ points.

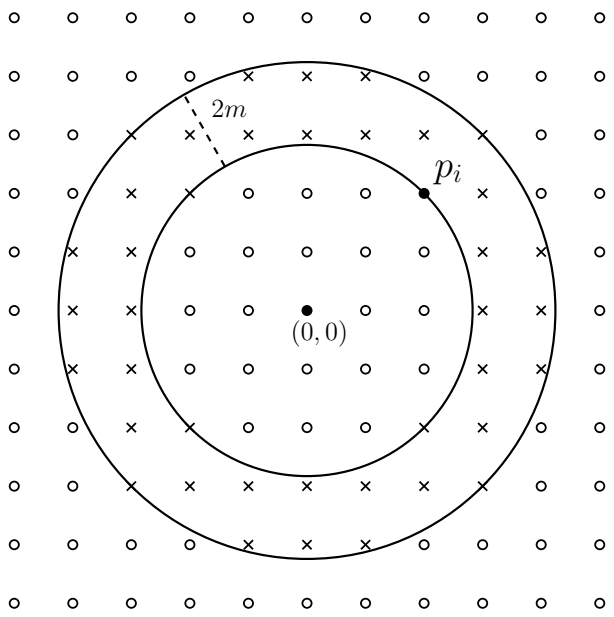


Figure 7.4: The set of lattice points appearing after p_i in P whose pairs might intertwine with those of p_i form an annulus centered at O with small radius $d(p_i, O)$ and big radius $d(p_i, O) + 2\delta$. In the figure, they are marked with an $\times$.

The algorithm processes the points in P in chunks of n at a time, adding available center–pixel pairs generated by points in the chunk (or points after it, as we will see) to the matching in order by distance. The invariant is that after a chunk is processed, its points do not generate any more available pairs, and we can move on to the next one until the matching is complete. To do this, for each chunk we construct the list L of all the pairs generated by its points. Let d be the maximum distance among these pairs. If p_i is the last point in the chunk, the points in P from p_{i+1} up to the last point at distance to the origin at most $d(p_i, O) + 2\delta$ can generate pairs with distance less than d. We add any such pair to L. We have to check $O(n)$ additional points, so L still has size $O(kn)$. We sort all these pairs and consider them in order, adding any available pair to the matching. Since each chunk has size n, there will be $O(n)$ chunks. Each one requires sorting a list of $O(kn)$ pairs, which requires $O(kn \log n)$ time (since $k \leq n^2$) and $O(kn)$ space. In total, we need $O(n^2 k \log n)$ time and $O(n^2 + nk)$ space.

Algorithm 14 Circle-growing algorithm for k real centers on an $n \times n$ grid.

Set all pixels as unmatched.
Set the quota of the first n^2 mod k centers to $\lceil n^2/k \rceil$.
Set the quota of the remaining centers to $\lfloor n^2/k \rfloor$.
Let $P =$ list of points (x, y) such that $-n < x, y < n$.
Sort P by non-decreasing distance to $(0, 0)$.
For each center c, let $p_c = (\text{round}(c_x), \text{round}(c_y))$.
Let $\delta = \max\{\text{dist}(c, p_c)\}$ among all centers.
$j \leftarrow 1$
while the matching is not complete **do**
    $L \leftarrow$ empty list
    $i \leftarrow \min(j + n, |P|)$
    for all $p \in P_j, \ldots, P_i$ **do** ▷ Add to L pairs generated by points in the next chunk
        for all centers c with quota > 0 **do**
            $s \leftarrow p + p_c$
            if $0 \le s_x, s_y < n$ and s is still available **then**
                Add (c, s) to L.
    Let $d = \max\{\text{dist}(c, s)\}$ among all pairs $(c, s) \in L$.
    for all $p \in P_{i+1}, \ldots, P_{|P|}$ **do** ▷ Add to L pairs closer than pairs already in L
        if $\text{dist}(p, O) > \text{dist}(P_i, O) + 2\delta$ **then**
            break
        for all centers c with quota > 0 **do**
            $s \leftarrow p + p_c$
            if $0 \le s_x, s_y < n$ and s is still available and $\text{dist}(c, s) \le d$ **then**
                Add (c, s) to L.
    Sort L by non-decreasing center–pixel distance.
    for all $(c, s) \in L$ **do**
        if c and s are available **then**
            Match s and c.
            Reduce the quota of c by 1.
    $j \leftarrow i + 1$

7.3.2 Distance-sorting algorithms

Unless the centers are clustered together, the circle-growing algorithm finds many available pairs in the early iterations. However, it reaches a point in which most circles overlap. Even if the centers are randomly distributed, in the typical case a large fraction of centers have "far outliers", pixels which belong to their region but are arbitrarily far because all the area in between is claimed by other centers. Consequently, many centers have to scan a large fraction of the square. At some point, thus, it is convenient to switch to a different algorithm that can find the closest available pairs quickly. In this section, let m and $k \leq m$ denote, respectively, the number of available pixels and centers after a matching has been partially completed.

Pair Sort (Algorithm 15). This algorithm simply sorts all the center–pixel pairs by distance and considers them in order, adding any available pair to the matching until it is complete. This algorithm is convenient when we can use integer sorting techniques, as in the case of the Euclidean metric and integer centers. Then, it requires $O(mk)$ time and space.

While the pair sort algorithm has a big memory requirement to be used starting with an empty matching, used after the circle-growing algorithm has matched a large fraction of pixels results in improved performance.

Algorithm 15 Pair Sort algorithm for k centers and m pixels.

```
L ← empty list
for all centers c do
    for all pixels s do
        Add (c, s) to L.
Sort L by non-decreasing center–pixel distance.
for all (c, s) ∈ L do
    if c and s are available then
        Match s and c.
        Reduce the quota of c by 1.
```

Pair Heap (Algorithm 16). When centers have real coordinates, sorting all the pairs takes $O(mk \log m)$ time, but we can do better. We find for each pixel s its closest center c_s, and build a min-heap with all the center–pixel pairs of the form (c_s, s) using $d(c_s, s)$ as key. Clearly, the top of the heap is a closest available pair. We can iteratively extract and match the top of the heap until one of the centers becomes unavailable. When a center c becomes unavailable, all the pairs in the heap containing c become unavailable. At this point, there are two possibilities:

Eager update We find the new closest available center of all the pixels that had c as closest center and rebuild the heap from scratch so that it again contains one pair for each available pixel and its closest available center.

Lazy update We proceed as usual until we actually extract a pair (c_s, s) with an unavailable center. Then, we find the new closest available center only for s, and reinsert the new pair in the heap.

In both cases, we repeat the process until the matching is complete.

We have not addressed yet how to find the closest center to a pixel. For this, we can use a nearest neighbor (NN) data structure that supports deletions. Such a data structure maintains a set of points and is able to answer *nearest neighbor queries*, which provide a query point q and ask for the point in the set closest to q. For the pair heap algorithm, we initialize the NN data structure with the set of centers and delete them as they become unavailable.

Since we need deletions we can use a *dynamic* NN data structure, i.e., with support for insertions as well as deletions. The simplest NN algorithm is a linear search, and a dynamic data structure based on it has $O(k)$ time per query and $O(1)$ time per update. The best known complexity of a dynamic NN data structure is $O(\log^4 k)$ amortized time per operation [46].

Given that we know all the query points for our NN data structure ahead of time (the pixels), we can build for each pixel s an array A_s with all the centers sorted by distance to s. Then, the closest

Algorithm 16 Pair Heap algorithm with lazy updates for k centers and m pixels.

```
Let C = nearest neighbor data structure with all the centers.
Let H = empty min-heap of center–pixel pairs using distance as key.
for all pixels s do
    Add (C.nearest(s), s) to H.
while H is not empty do
    (c, s) ← H.removeMin()
    if c has quota > 0 then
        Match s and c.
        Reduce the quota of c by 1.
    else
        Remove c from C.
        Add (C.nearest(s), s) to H.
```

center to a pixel s is $A_s[i_s]$, where i_s is the index of the first available center in A_s. When a center is deleted we simply mark it. When we get a query for the closest center to a pixel s, we search A_s until we find an unmarked center. We can start the search from the index of the center returned in the last query for s. This data structure requires $O(mk)$ space and has a $O(mk \log k)$ initialization cost to sort all the arrays. The interesting property is that if we do $O(k)$ queries for a given pixel s, we require $O(k)$ time for all of them, as in total we traverse A_s only once. We call this data structure *presort*, although it is not strictly a NN data structure because it knows the query points ahead of time.

In the pair heap algorithm, we can combine eager and lazy updates with any NN data structure. In any case, the running time is influenced by α, the sum among all centers c of the number of pixels that had c as closest center when c became unavailable. In the worst case $\alpha = O(km)$, but assuming that each center is equally likely to be the closest center to each pixel, the expected value of α is $O(m)$. In the experiments, we test the value of α empirically.

With eager updates in total we have to initialize the NN data structure, perform m *extract-min* operations, $O(m + \alpha)$ NN queries, k NN deletions, and rebuild the heap k times. Thus, the running time is $O(P(k, m) + m \log m + (m + \alpha)Q(k) + kD(k) + km)$, where $P(k, m)$ is the cost of initializing the NN data structure of choice with k points (and m query points, in the case of

the presort data structure), and $Q(k)$ and $D(k)$ are the costs of queries and deletions, respectively. With lazy updates, instead of rebuilding the heap we have $O(\alpha)$ extra *insert* and *extract-min* heap operations, which requires $O(\alpha \log m)$ time.

For real centers, the best worst-case bound is with eager deletions and the presort NN data structure. In that case, we have that the NN queries take $O(km)$ for any α, so the total running time is $O(mk \log k + m \log m)$. If we assume that $\alpha = O(m)$, then the best time is with lazy deletions and the NN data structure from [46]. The running time with this heuristic assumption is $O(m \log^4 k + m \log m)$.

7.3.3 Experiments

Data set. Table 7.2 summarizes the parameters used in the different experiments. We use the following labels for the algorithms: CG the circle-growing algorithm alone, and PS and PH for the combination of CG and the pair sort and pair heap algorithms, respectively. Moreover, for the pair heap algorithm we consider the following variations: eager/presort ($PH_{E,P}$), eager/linear search ($PH_{E,L}$), lazy/presort ($PH_{L,P}$), and lazy/linear search ($PH_{L,L}$).

We focus on the Euclidean, Manhattan, and Chebyshev metrics. The parameter n is the length of the side of the square grid, and k is the number of centers. In all the experiments, the centers are chosen uniformly and independently at random. Moreover, every data point is the average of 10 runs, each starting with different centers.

The cutoff is the parameter used to determine when to switch from the circle-growing algorithm to a different one. We define it as a ratio between the number of available pairs and the number of pairs already considered by the circle-growing algorithm.

The algorithms were implemented in C++ (gcc version 4.8.2) and the interface in Qt[19]. The experiments were executed on an Intel(R) Core(TM) CPU i7-3537U 2.00GHz with 4GB of RAM, on Windows 10.

Algorithm comparison. Figure 7.5 contains a comparison of all the algorithms. Pair heap is generally better than pair sort, even for integer distances where it has a higher theoretical complexity. Among pair heap variations, lazy/linear is the best for both types of centers. In general lazy updates perform better, but eager/presort is also a strong combination because of their synergy: eager updates require more NN queries in exchange for fewer extract-min heap operations, and the presort data structure has fast NN queries.

Optimal cutoff. When combining the circle-growing algorithm with another algorithm, the efficiency of the combination depends on the cutoff used to switch between both. If we switch too soon, we don't exploit the good behavior of the circle-growing algorithm when circles are still mostly disjoint. If we switch too late, the circle-growing algorithm slows down as it grows the circles in every direction just to reach some outlying region.

Figure 7.6 illustrates the role of the cutoff. The figures show that the metric used does not play a major role in the optimal cutoff nor the execution time. It shows that most of the execution time of the circle-growing algorithm is spent with the very few last available pairs, so even a really small cutoff prompts a substantial improvement. After that, the additional time spent in the pair heap

Table 7.2: Summary of parameters used in the experiments section.

Experiment	**Algorithms**	**Metric**	n	k	**Cutoff**
Exec. time (Fig. 7.5)	All	L_2	varies	$10n$	0.15
Cutoff (Fig. 7.6)	$CG, PH_{L,L}$	L_2	1000	varies	varies
Cutoff (Fig. 7.7)	$CG, PH_{L,L}$	L_2	1000	10000	varies
Value of α	PH with lazy deletions	L_2	1000	varies	—

[19]The code is available at `https://github.com/nmamano/StableMatchingVoronoiDiagram`.

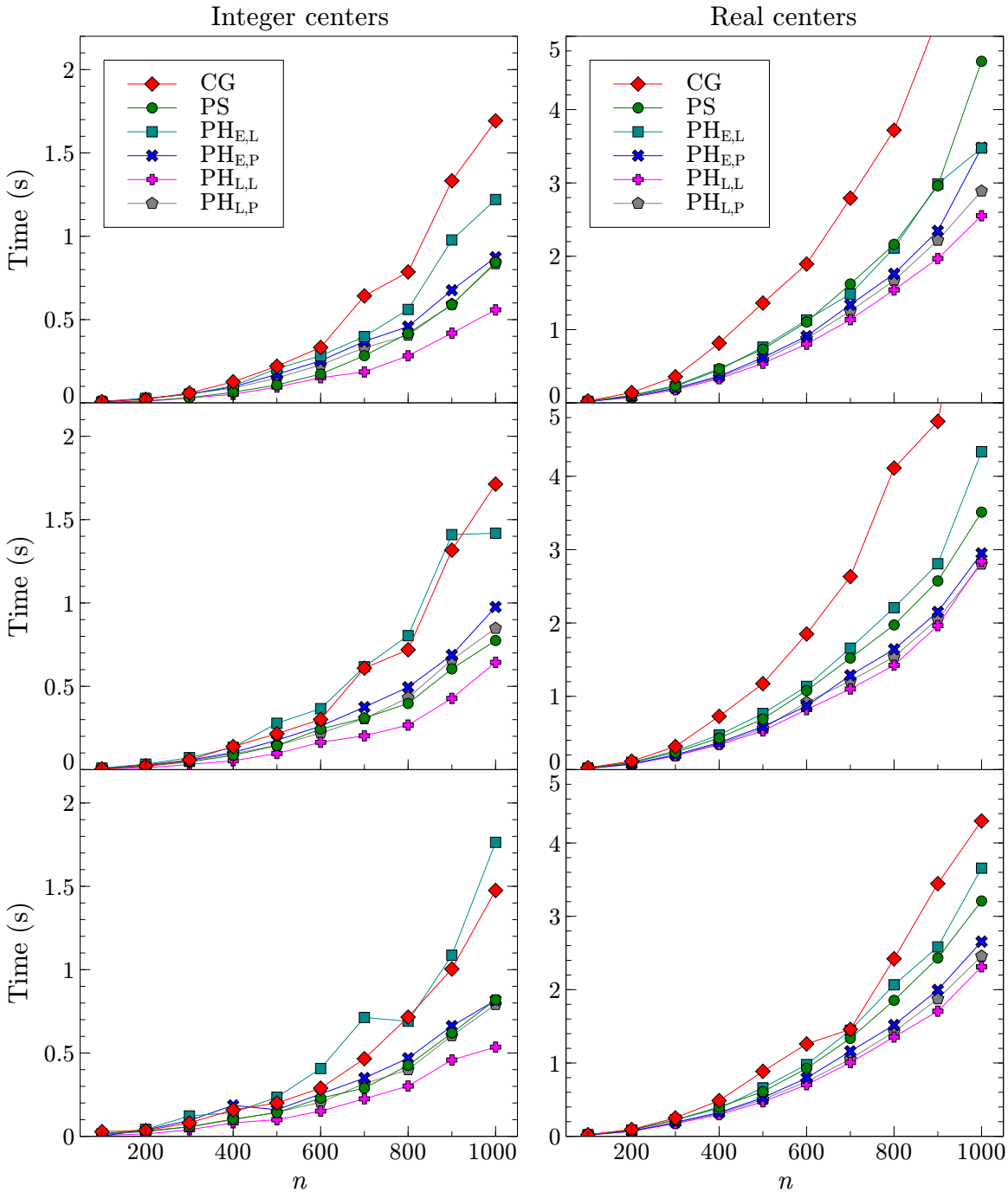


Figure 7.5: Execution time of the various algorithms. We consider integer (left) and real (right) centers, and Euclidean (top), Manhattan (middle), and Chebyshev (bottom) metrics. For all the methods but *CG*, the cutoff is 0.15. Each data point is the average of 10 runs with $10n$ randomly distributed centers.

algorithm slightly beats the savings in the circle-growing algorithm, resulting in a steady increase of the total running time. In addition, Figure 7.7 shows in more detail how this execution time is divided among the circle-growing algorithm and the pair heap algorithm.

Value of α. The running time of the pair heap algorithm depends on α, the sum among all centers c of the number of pixels that had c as closest center when c became unavailable. There is a gap between the worst case $\alpha = O(km)$ and the expected case $\alpha = O(m)$ when pixels and centers are distributed randomly. Even with randomly located centers, the distribution of remaining pixels and centers after the circle-growing algorithm is not random, so here we are interested in the actual value of α in such cases. More precisely, we are interested in β, the total number of extra *extract-min* operations (i.e., operations returning an unavailable pair) when using the pair heap algorithm with lazy updates. Note that $\beta \leq \alpha$, because with lazy updates when a center becomes unavailable some of the pixels that would have it as closest center still have a previously unavailable center instead.

First, we observe the value of β/m when using the pair heap algorithm on its own in an $n = 100$ ($m = 10000$) grid, using randomly distributed integer centers and the L_2 metric. The maximum values of β/m among 10 runs for each k were 0.64 for $k = 10$, 0.80 for $k = 100$, and 0.82 for $k = 1000$. We obtained similar values for the L_1 and L_∞ metrics; in every case, $\beta < m$.

Second, we observed the values of β/m when using the pair heap algorithm after the circle-growing algorithm, again using randomly distributed integer centers and the L_2 metric. We switched between algorithms when there were $m = 10000$ available pairs. The maximum values of β/m among 10 runs for each k were 1.20 for $k = 100$ (with 4 remaining centers), 4.42 for $k = 1000$ (with 31 remaining centers), and 7.86 for $k = 10000$ (with 275 remaining centers). We also obtained similar values for the L_1 and L_∞ metrics.

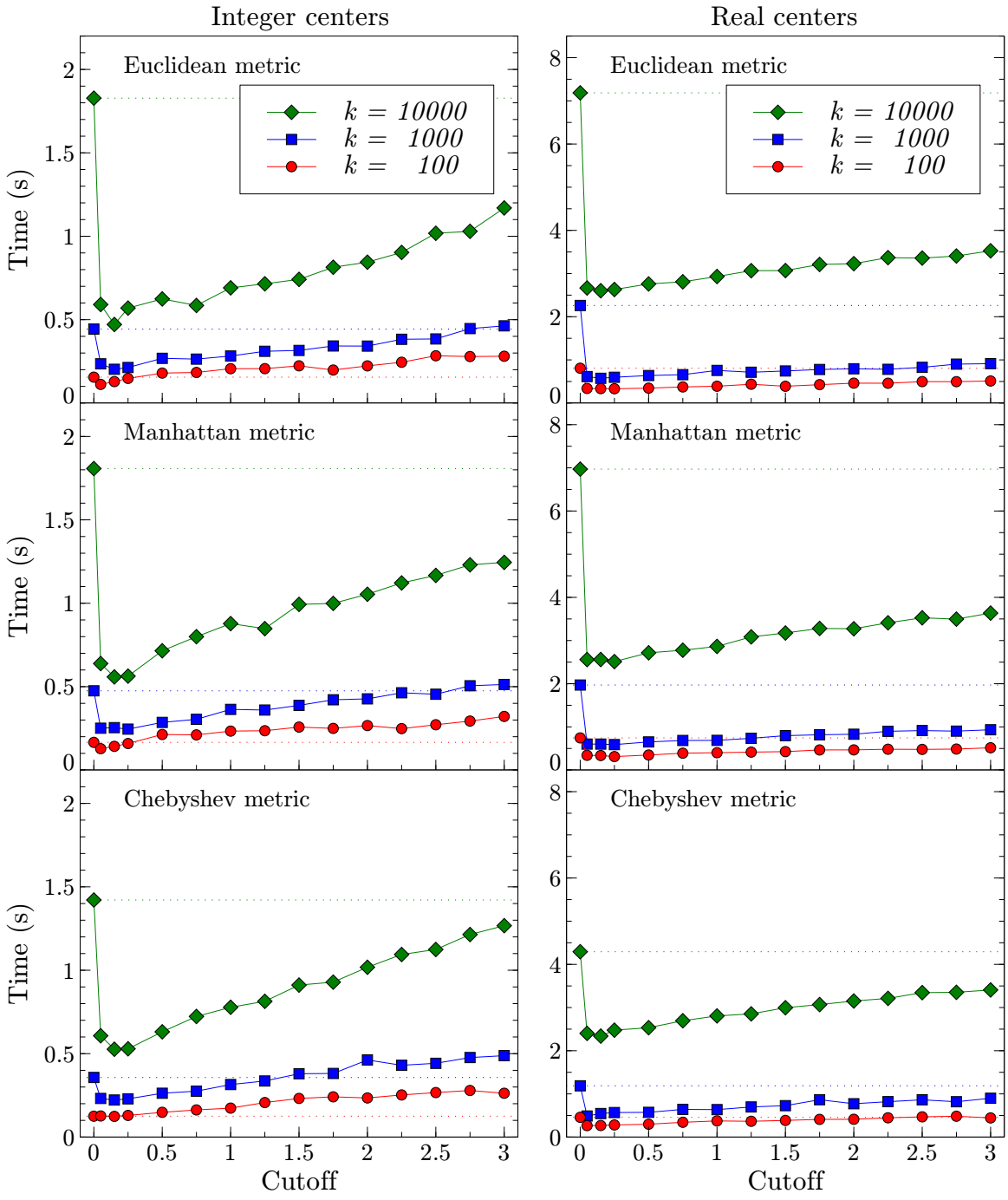


Figure 7.6: Execution time of the circle-growing algorithm combined with the pair heap algorithm with lazy updates and a linear search NN data structure. We consider integer (left) and real (right) centers, and Euclidean (top), Manhattan (middle), and Chebyshev (bottom) metrics. The dotted lines denote the running time of the circle-growing algorithm alone, i.e., with cutoff 0. Each data point is the average of 10 runs with randomly distributed centers, $n = 1000$, and the L_2 metric.

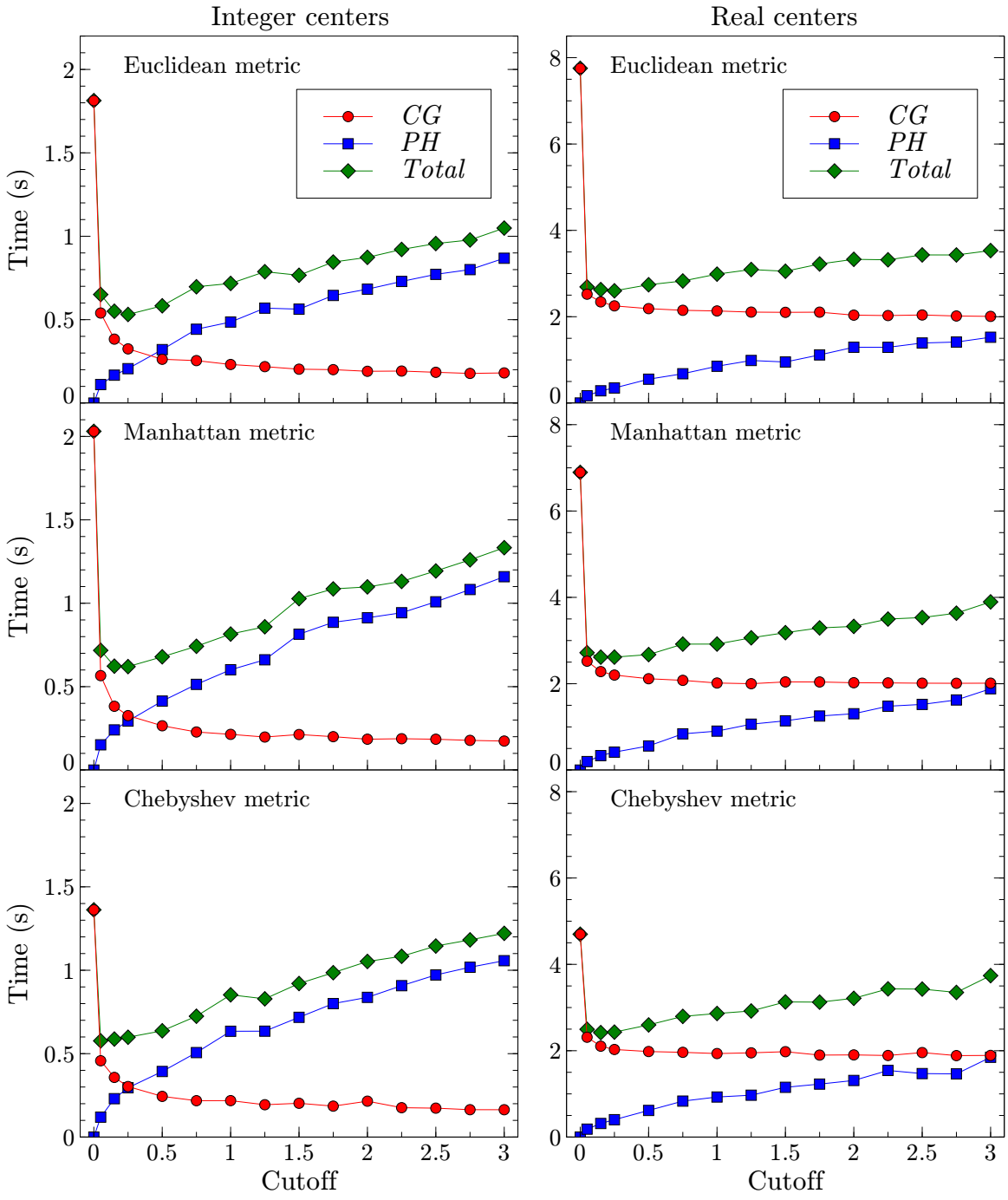


Figure 7.7: Execution time of the circle-growing algorithm combined with the pair heap algorithm with lazy updates and a linear search NN data structure. We consider integer (left) and real (right) centers, and Euclidean (top), Manhattan (middle), and Chebyshev (bottom) metrics. In addition to the total execution time, we show the execution time spent in each algorithm. Each data point is the average of 10 runs with $n = 1000$ and 10000 randomly distributed centers.

The reason for the worse-than-random behavior when matching the last remaining pixels is that outlying zones tend to cluster together, and then all the pixels in those zones are likely to have the same center as closest center. Overall, the experiments show that $\alpha = O(m)$ is a reasonable assumption.

7.4 Center location

One question that we have not addressed so far is finding a good location for the centers. The location of the centers affects the shape of the regions, so it is important, e.g., for districting, where disconnected regions are highly undesirable. Hence, we study the effect of integrating a stable matching algorithm with a *k-means* clustering method. In the traditional k-means clustering problem, the goal is to partition a set of points in space into k clusters such that each point is close to the mean of the cluster. It has been studied extensively in discrete contexts (e.g., see [119, 122]).

The usual method for k-means clustering is a simple iterative refinement algorithm, called the *k-means algorithm* or *Lloyd's algorithm* [122]: we begin by choosing k points, called cluster centers, randomly in the space. Then, we iteratively repeat the following two phases: 1) *assignment* step: each point is assigned to its closest center, and 2) *update* step: each center is moved to the centroid of the points assigned to it.

Lloyd's algorithm converges to a locally-optimal partition that minimizes the sum of the squared distances from each point to its assigned center [122]. In this section, we propose a variation, which we call *stable k-means*, where the assignment step is replaced by a stable matching between points and centers, so as to achieve the additional property that the regions all have equal area (to within round-off errors).

We consider this in the grid setting, although it could also be considered in the graph setting. If the graph is not embedded in the plane, instead of using centroids we would use some notion of graph centrality [32].

We have found through experimentation that, although the stable k-means method succeeds in improving compactness, centers can sometimes stop moving while we are executing Lloyd's algorithm before their regions became completely connected. See Figure 7.8. Thus, we introduce an additional heuristic: we use weighted centroids, which are more sensitive to the outlying parts of their region.

7.4.1 Stable *k*-means with weighted centroids

The usual centroid of a set of points S is defined as $(\sum_{q\in S} q)/|S|$, where the points are regarded as two-dimensional vectors so that the sum makes sense. Instead, we can compute a weighted centroid as $(\sum_{q\in S} w_q q)/(\sum_{q\in S} w_q)$. A natural choice to use for the weight w_q of a point q assigned to the region of the center c is the distance from q to c raised to some exponent p that we can choose, $d(q,c)^p$. The larger p is, the more sensitive the weighted centroids are to outliers. When $p = 0$, we get the usual centroid. When p approaches $+\infty$, we get the circumcenter of the region.

Figure 7.9 shows how the exponent p of the weighted centroid affect the result of the stable k-means method. As evaluation measure, we use the average distance among pixels and their assigned centers. The best results are with $-0.8 < p < 0.4$ (for different metrics and grid sizes, we obtain similar results). Figure 7.10 shows the result of the stable k-means method for values of p between -2 and 0.5. Figure 7.11 shows the transient behavior of stable k-means with $p \geq 1$.

On a related note, if we set $p = -1$ and repeatedly move a center to its weighted centroid (keeping its region unchanged), we get Weiszfeld's algorithm [184], a known iterative method for finding the geometric median (the point minimizing the sum of distances to its region). In our case, we

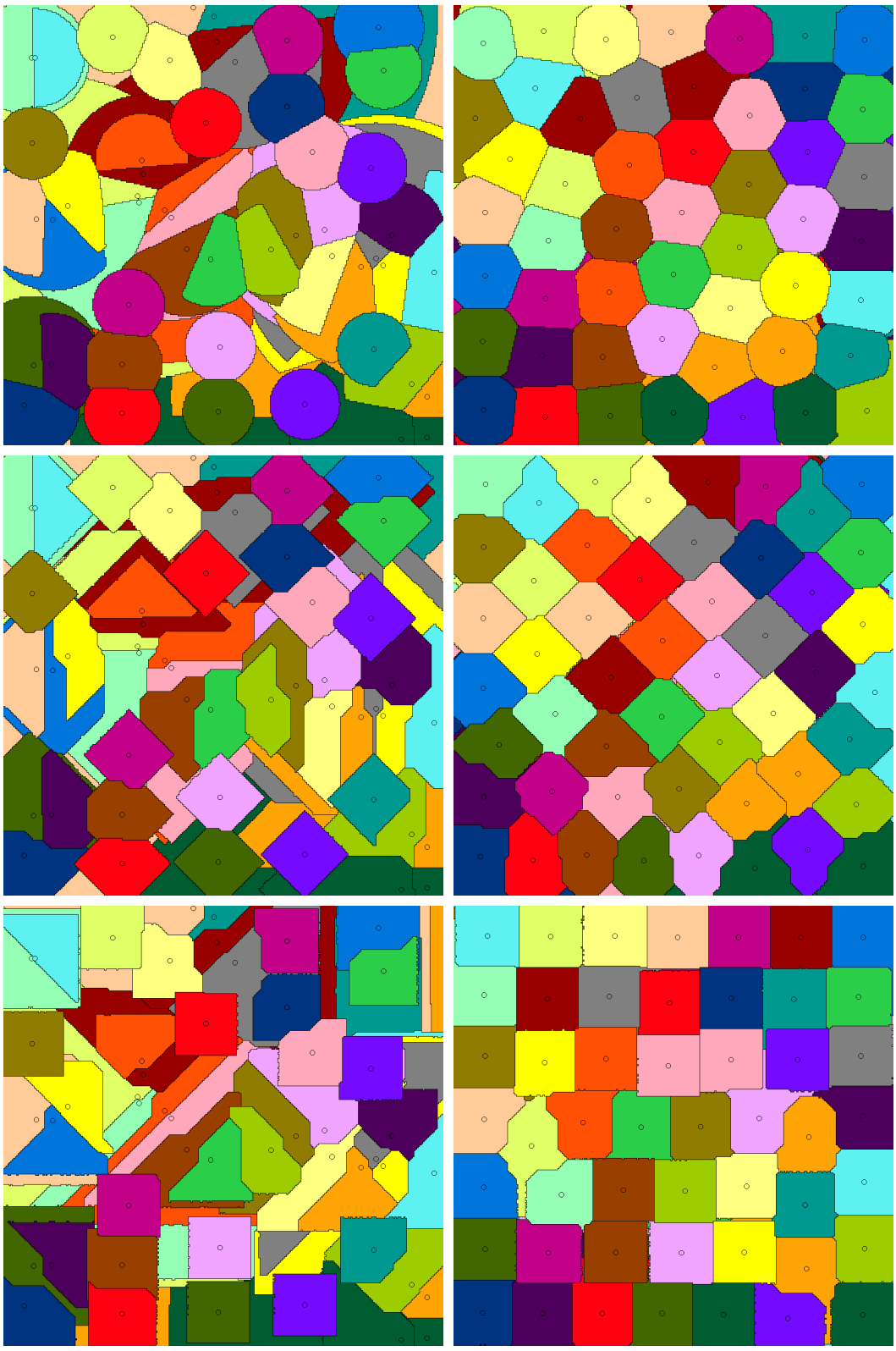


Figure 7.8: Left: stable matching in a 300×300 grid with the same 50 random centers for the Euclidean (top), Manhattan (center), and Chebyshev (bottom) metrics. Right: result of the stable k-means algorithm with unweighted centroids for each metric.

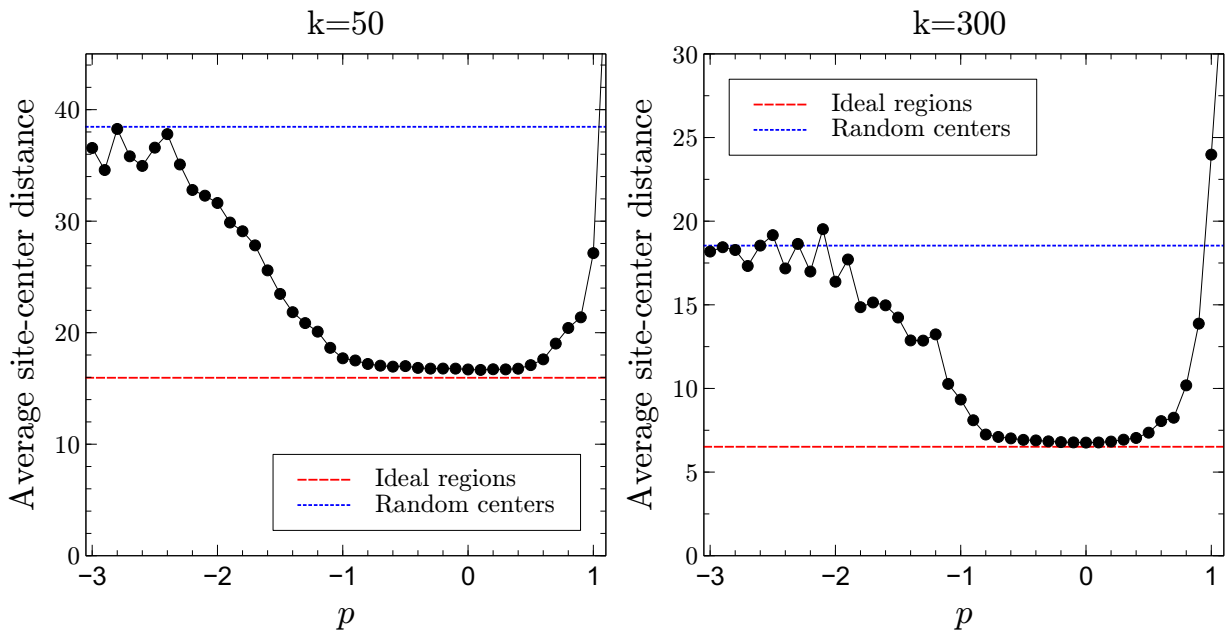


Figure 7.9: Average distance among pixels and their matched center after 100 iterations of stable k-means for different exponents p of the weighted centroid. We consider integer centers, the Euclidean metric, and grid size $n = 300$. The blue dotted line denotes the average distance with random centers, and the red dashed line denotes the average distance in an ideal region (i.e., a disk). Each data point is the average of 10 runs starting with randomly distributed centers.

are not doing the same thing, because we are also recomputing the regions after each update of the centers, but it is still the case that for $p = -1$ the algorithm converges to a state where each center is the geometric median of its region.

7.5 Conclusions

Algorithmic redistricting is an area of research with the potential to mitigate partisan gerrymandering. The goal is to design algorithms that draw fair, politically-neutral districts. We proposed the use of stability—in the stable matching sense—for the redistricting problem. Our algorithms can be applied to road networks with tens of thousands of nodes, or to grids with millions of points.

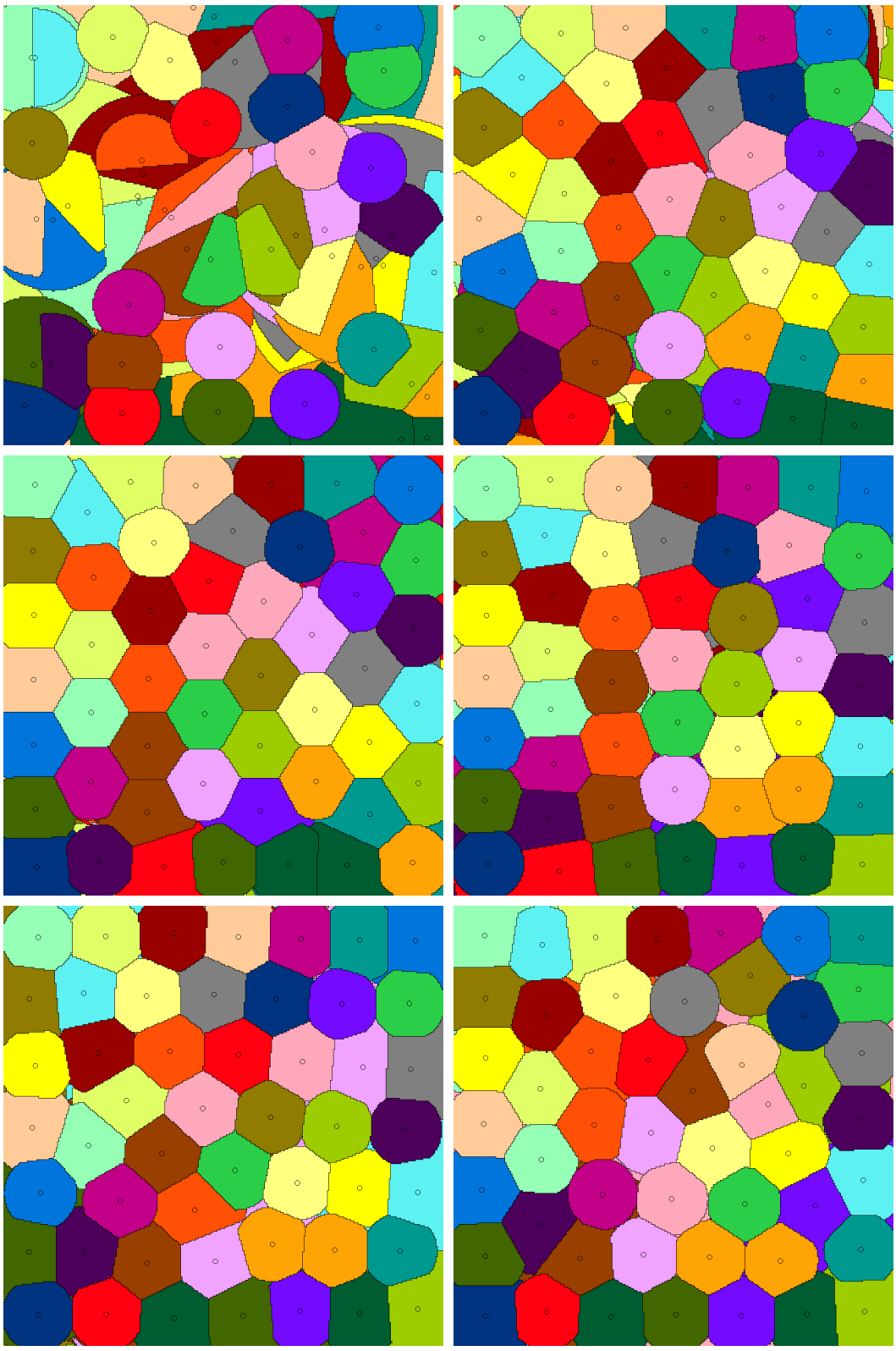


Figure 7.10: Top left: Same matching from Figure 7.8 with $n = 300$ and 50 random centers. Other figures, from top to bottom and left to right: Result of the stable k-means method with weighted centroids for $p = -2, -1, 0.1, 0.2$, and 0.5.

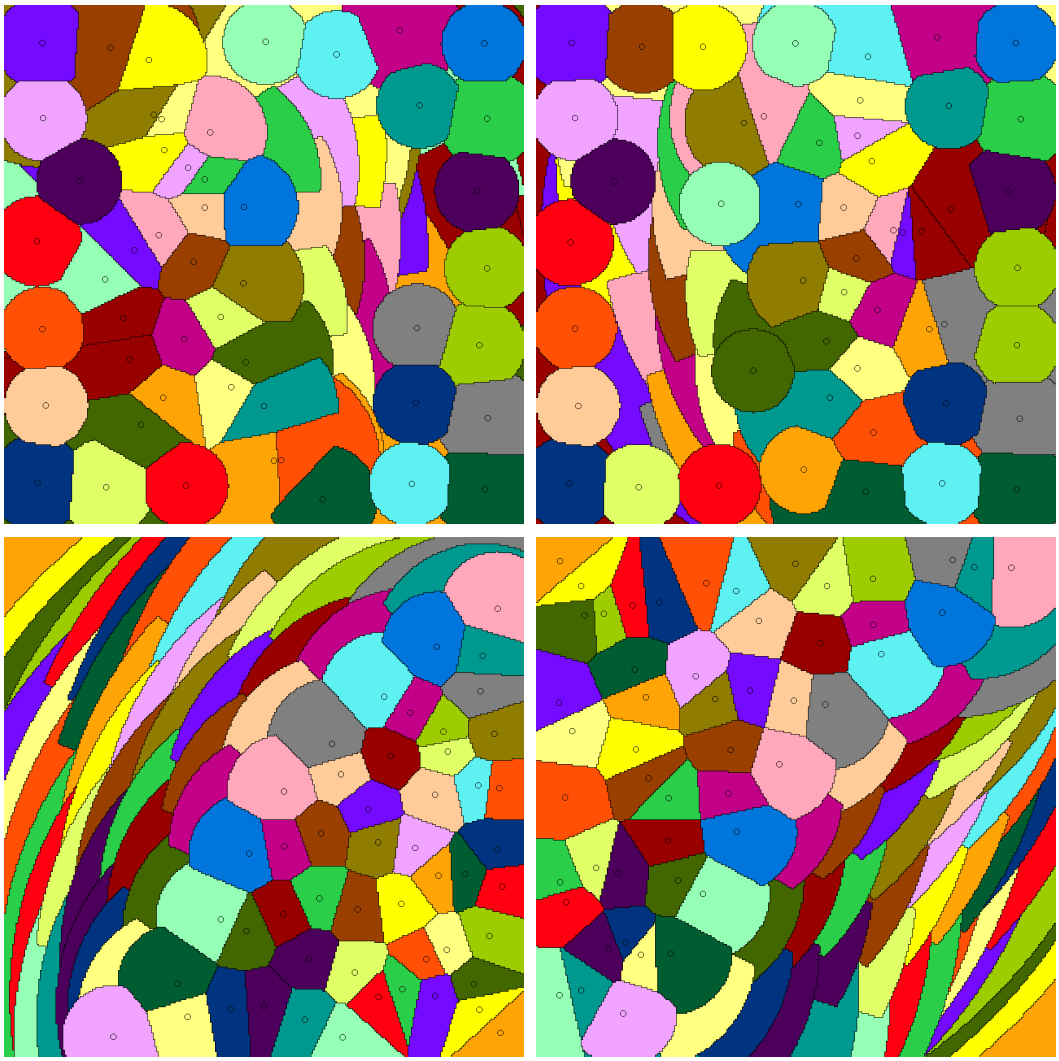


Figure 7.11: Top: Two consecutive iterations of the stable k-means method with weighted centroids for $p = 1$, for the same matching from Figure 7.8 with $n = 300$ and 50 random centers. Bottom: same for $p = 10$.

The main disadvantage of using stability for redistricting is that districts are generally not connected. Achieving connectivity would require a relaxation of the stability property and/or a better choice of the center locations. If the district centers can be moved, the issue of disconnected regions can be severely reduced using *stable k-means*, a variant of k-means that converges to stable and equally-sized clusters. However, non-connectivity is not eliminated completely (e.g., see Figure 7.8). It seems that in order to draw successful districts with this approach, some post-processing would still be required to handle the remaining disconnected points.

A challenge for algorithmic redistricting is that there is no definitive answer to what algorithm to use or what compactness measure to optimize, so meddling parties might fight over the algorithm that benefits them the most. Algorithms themselves have configuration and implementation choices that can affect the outcome. In this way, even politically-agnostic algorithms are susceptible to manipulation, as some choices will favor a party more than others. A nice feature of stability when preferences are based on proximity is that there is a unique stable matching. This is not true of stable matching with arbitrary preferences, where one has to choose among possibly many stable matchings. That said, the output of our algorithm is still heavily affected by the location of the district centers.

Bibliography

[1] P. K. Agarwal, D. Eppstein, and J. Matoušek. Dynamic half-space reporting, geometric optimization, and minimum spanning trees. In *33rd Symp. Foundations of Computer Science (FOCS)*, pages 80–89, 1992.

[2] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In *Advances in discrete and computational geometry (South Hadley, MA, 1996)*, volume 223 of *Contemp. Math.*, pages 1–56. Amer. Math. Soc., Providence, RI, 1999.

[3] P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. *SIAM Journal on Computing*, 22(4):794–806, 1993.

[4] P. K. Agarwal and J. Pan. Near-linear algorithms for geometric hitting sets and set covers. In *Proceedings of the thirtieth annual symposium on Computational geometry*, page 271. ACM, 2014.

[5] P. K. Agarwal and M. Sharir. Davenport–Schinzel Sequences and Their Geometric Applications. Technical Report DUKE–TR–1995–21, Duke University, Durham, NC, USA, 1995.

[6] G. Aggarwal, S. Muthukrishnan, D. Pál, and M. Pál. General auction mechanism for search advertising. In *18th Int. Conf. on the World Wide Web (WWW)*, pages 241–250. ACM, 2009.

[7] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[8] O. Aichholzer, F. Aurenhammer, D. Alberts, and B. Gärtner. A novel type of skeleton for polygons. In *J. UCS The Journal of Universal Computer Science*, pages 752–761. Springer, 1996.

[9] V. Akman, W. R. Franklin, M. Kankanhalli, and C. Narayanaswami. Geometric computing and uniform grid technique. *Computer-Aided Design*, 21(7):410–420, 1989.

[10] J. Alanko and T. Norri. Greedy shortest common superstring approximation in compact space. In *International Symposium on String Processing and Information Retrieval*, pages 1–13. Springer, 2017.

[11] H. Alt, E. M. Arkin, H. Brönnimann, J. Erickson, S. P. Fekete, C. Knauer, J. Lenchner, J. S. B. Mitchell, and K. Whittlesey. Minimum-cost coverage of point sets by disks. In *Proceedings of the twenty-second annual symposium on Computational geometry*, pages 449–458. ACM, 2006.

[12] E. M. Arkin, S. W. Bae, A. Efrat, K. Okamoto, J. S. Mitchell, and V. Polishchuk. Geometric stable roommates. *Information Processing Letters*, 109(4):219–224, 2009.

[13] E. M. Arkin, S. P. Fekete, and J. S. B. Mitchell. Approximation algorithms for lawn mowing and milling. *Computational Geometry*, 17(1):25–50, 2000.

[14] S. Arora. Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. *Journal of the ACM (JACM)*, 45(5):753–782, 1998.

[15] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.

[16] F. Aurenhammer. Voronoi diagrams—A survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1991.

[17] F. Aurenhammer, F. Hoffmann, and B. Aronov. Minkowski-type theorems and least-squares clustering. *Algorithmica*, 20(1):61–76, 1998.

[18] F. Aurenhammer, R. Klein, and D. Lee. *Voronoi Diagrams and Delaunay Triangulations*. World Scientific, 2013.

[19] D. Avis. A survey of heuristics for the weighted matching problem. *Networks*, 13(4):475–493, 1983.

[20] G. Barequet, M. T. Goodrich, A. Levi-Steiner, and D. Steiner. Straight-skeleton based contour interpolation. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 119–127, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[21] J. J. Bentley. Fast algorithms for geometric traveling salesman problems. *ORSA Journal on Computing*, 4(4):387–411, 1992.

[22] J. L. Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23(4):214–229, Apr. 1980.

[23] J. L. Bentley. Experiments on traveling salesman heuristics. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 91–99, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[24] J.-P. Benzécri. Construction d'une classification ascendante hiérarchique par la recherche en chaîne des voisins réciproques. *Les cahiers de l'analyse des données*, 7(2):209–218, 1982.

[25] S. N. Bespamyatnikh. An optimal algorithm for closest-pair maintenance. *Discrete & Computational Geometry*, 19(2):175–195, Feb 1998.

[26] N. Bhatnagar, S. Greenberg, and D. Randall. Sampling stable marriages: why spouse-swapping won't work. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1223–1232. Society for Industrial and Applied Mathematics, 2008.

[27] P. Bhattacharya and M. L. Gavrilova. Roadmap-based path planning – Using the Voronoi diagram for a clearance-based shortest path. *IEEE Robotics Automation Magazine*, 15(2):58–66, 2008.

[28] T. Biedl, T. Bläsius, B. Niedermann, M. Nöllenburg, R. Prutkin, and I. Rutter. Using ILP/SAT to determine pathwidth, visibility representations, and other grid-based graph drawings. In S. Wismath and A. Wolff, editors, *21st Int. Symp. on Graph Drawing (GD)*, pages 460–471, 2013.

[29] V. Bilò, I. Caragiannis, C. Kaklamanis, and P. Kanellopoulos. Geometric clustering to minimize the sum of cluster sizes. In *Algorithms - ESA 2005, 13th Annual European Symposium, Palma de Mallorca, Spain, October 3-6, 2005, Proceedings*, pages 460–471, 2005.

[30] A. Biniaz, P. Bose, P. Carmi, A. Maheshwari, I. Munro, and M. Smid. Faster Algorithms for some Optimization Problems on Collinear Points. In B. Speckmann and C. D. Tóth, editors, *34th International Symposium on Computational Geometry (SoCG 2018)*, volume 99 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:14, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[31] H. Blum. A Transformation for Extracting New Descriptors of Shape. In W. Wathen-Dunn, editor, *Models for the Perception of Speech and Visual Form*, pages 362–380. MIT Press, Cambridge, 1967.

[32] P. Bonacich. Power and centrality: A family of measures. *American Journal of Sociology*, 92(5):1170–1182, 1987.

[33] J. C. Bowers. Faster Reductions for Straight Skeletons to Motorcycle Graphs. *arXiv e-prints*, page arXiv:1405.6260, May 2014.

[34] J. W. Brandt and V. R. Algazi. Continuous skeleton computation by Voronoi diagram. *CVGIP: Image Understanding*, 55(3):329–338, 1992.

[35] J. Brecklinghaus and S. Hougardy. The approximation ratio of the greedy algorithm for the metric traveling salesman problem. *Operations Research Letters*, 43(3):259 – 261, 2015.

[36] G. S. Brodal. Worst-case efficient priority queues. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '96, pages 52–58, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

[37] G. S. Brodal, G. Lagogiannis, and R. E. Tarjan. Strict fibonacci heaps. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 1177–1184, New York, NY, USA, 2012. ACM.

[38] H. Brönnimann and M. T. Goodrich. Almost optimal set covers in finite VC-dimension. *Discrete & Computational Geometry*, 14(4):463–479, 1995.

[39] M. Bruynooghe. New methods in automatic classification of numerous taxonomic data. *Statistics and data analysis*, 2(3):24–42, 1977.

[40] M. Bruynooghe. Classification ascendante hiérarchique des grands ensembles de données: un algorithme rapide fondé sur la construction des voisinages réductibles. *Les cahiers de l'analyse de données*, 3:7–33, 1978.

[41] L. S. Buriol, M. G. C. Resende, and M. Thorup. Speeding up dynamic shortest-path algorithms. *INFORMS J. Comput.*, 20(2):191–204, 2008.

[42] F. Cacciola. A CGAL implementation of the straight skeleton of a simple 2d polygon with holes. *2nd CGAL User Workshop*, 01 2004.

[43] C. Calabro, R. Impagliazzo, and R. Paturi. The complexity of satisfiability of small depth circuits. In J. Chen and F. V. Fomin, editors, *Parameterized and Exact Computation*, pages 75–85, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[44] E. P. F. Chan and Y. Yang. Shortest path tree computation in dynamic graphs. *IEEE Trans. Comput.*, 58(4):541–557, 2009.

[45] T. M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. *J. ACM*, 57(3):16:1–16:15, 2010.

[46] T. M. Chan. Dynamic geometric data structures via shallow cuttings. In *35th International Symposium on Computational Geometry, SoCG 2019, June 18-21, 2019, Portland, Oregon, USA.*, pages 24:1–24:13, 2019.

[47] S. Chandran, S. K. Kim, and D. M. Mount. Parallel computational geometry of rectangles. *Algorithmica*, 7(1):25–49, 1992.

[48] J. Chen and U. P. Finnendahl. On the number of single-peaked narcissistic or single-crossing narcissistic preference profiles. *Discrete Mathematics*, 341(5):1225–1236, 2018.

[49] S.-W. Cheng, L. Mencel, and A. Vigneron. A faster algorithm for computing straight skeletons. *ACM Trans. Algorithms*, 12(3):44:1–44:21, Apr. 2016.

[50] S.-W. Cheng and A. Vigneron. Motorcycle graphs and straight skeletons. *Algorithmica*, 47(2):159–182, Feb 2007.

[51] L. P. Chew and R. L. S. Dyrsdale, III. Voronoi diagrams based on convex distance functions. In *Proceedings of the First Annual Symposium on Computational Geometry*, SCG '85, pages 235–244, New York, NY, USA, 1985. ACM.

[52] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon University Pittsburgh Pa Management Sciences Research Group, 1976.

[53] M. Chrobak and S. Nakano. Minimum-width grid drawings of plane graphs. *Computational Geometry*, 11(1):29–54, 1998.

[54] J. Chun, M. Korman, M. Nöllenburg, and T. Tokuyama. Consistent digital rays. *Discrete & Computational Geometry*, 42(3):359–378, 2009.

[55] F. R. K. Chung. Separator theorems and their applications. In *Paths, flows, and VLSI-layout (Bonn, 1988)*, volume 9 of *Algorithms Combin.*, pages 17–34. Springer, Berlin, 1990.

[56] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[57] K. L. Clarkson. Fast algorithms for the all nearest neighbors problem. In *24th Annual Symposium on Foundations of Computer Science*, pages 226–232, Nov 1983.

[58] K. L. Clarkson. Nearest-neighbor searching and metric space dimensions. In G. Shakhnarovich, T. Darrell, and P. Indyk, editors, *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice*, chapter 2, pages 15–59. MIT Press, 2006.

[59] F. Cloppet, J. M. Oliva, and G. Stamon. Angular bisector network, a simplified generalized Voronoi diagram: application to processing complex intersections in biomedical images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):120–128, Jan 2000.

[60] V. Cohen-Addad, P. N. Klein, and N. E. Young. Balanced centroidal power diagrams for redistricting. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '18, pages 389–396, New York, NY, USA, 2018. ACM.

[61] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill, 2nd edition, 2001.

[62] G. Cornuéjols, J. Fonlupt, and D. Naddef. The traveling salesman problem on a graph and some related integer polyhedra. *Mathematical Programming*, 33(1):1–27, Sep 1985.

[63] M. De Berg, O. Cheong, M. Van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 2008.

[64] L. De Floriani, E. Puppo, and P. Magillo. Applications of computational geometry to geographic information systems. *Handbook of Computational Geometry*, pages 333–388, 1999.

[65] H. De Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.

[66] F. Dehne, Q. T. Pham, and I. Stojmenović. Optimal visibility algorithms for binary images on the hypercube. *International Journal of Parallel Programming*, 19(3):213–224, 1990.

[67] E. D. Demaine, M. L. Demaine, and J. S. B. Mitchell. Folding flat silhouettes and wrapping polyhedral packages: New results in computational origami. *Computational Geometry*, 16(1):3–21, 2000.

[68] C. Demetrescu, A. V. Goldberg, and D. S. Johnson. 9th DIMACS Implementation Challenge: Shortest Paths, 2006.

[69] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.

[70] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, 1959.

[71] H. N. Djidjev, G. E. Pantziou, and C. D. Zaroliagis. On-line and dynamic algorithms for shortest path problems. In *12th Symp. on Theoretical Aspects of Computer Science (STACS)*, volume 900 of *LNCS*, pages 193–204. Springer, 1995.

[72] D. E. Drake and S. Hougardy. A simple approximation algorithm for the weighted matching problem. *Information Processing Letters*, 85(4):211–213, 2003.

[73] V. Dujmović, D. Eppstein, and D. R. Wood. Structure of graphs with locally restricted crossings. *SIAM J. Discrete Mathematics*, 31(2):805–824, 2017.

[74] Z. Dvořák and S. Norin. Strongly sublinear separators and polynomial expansion. *SIAM Journal on Discrete Mathematics*, 30(2):1095–1101, 2016.

[75] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, Apr. 1972.

[76] J. Eeckhout. On the uniqueness of stable marriage matchings. *Economics Letters*, 69(1):1–8, 2000.

[77] M. El Krari, B. Ahiod, and B. El Benani. An empirical study of the multi-fragment tour construction algorithm for the travelling salesman problem. In A. Abraham, A. Haqiq, A. M. Alimi, G. Mezzour, N. Rokbani, and A. K. Muda, editors, *Proceedings of the 16th International Conference on Hybrid Intelligent Systems (HIS 2016)*, pages 278–287, Cham, 2017. Springer International Publishing.

[78] D. Eppstein. Dynamic Euclidean minimum spanning trees and extrema of binary functions. *Discrete & Computational Geometry*, 13(1):111–122, 1995.

[79] D. Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. *J. Exp. Algorithmics*, 5, 2000.

[80] D. Eppstein. All maximal independent sets and dynamic dominance for sparse graphs. *ACM Transactions on Algorithms*, 5(4):Art. 38, 2009.

[81] D. Eppstein. Treetopes and their graphs. In *27th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 969–984, 2016.

[82] D. Eppstein and J. Erickson. Raising roofs, crashing cycles, and playing pool: Applications of a data structure for finding pairwise interactions. *Discrete & Computational Geometry*, 22(4):569–592, Dec 1999.

[83] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, pages 9.1–9.28. CRC Press, 2nd edition, 2010.

[84] D. Eppstein and M. T. Goodrich. Studying (non-planar) road networks through an algorithmic lens. In *16th ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems*, pages 16:1–16:10. ACM, 2008.

[85] D. Eppstein, M. T. Goodrich, and J. Z. Sun. Skip quadtrees: Dynamic data structures for multidimensional point sets. *Int. J. Comp. Geom. & Appl.*, 18(1-2):131–160, 2008.

[86] D. Eppstein, M. T. Goodrich, and L. Trott. Going off-road: Transversal complexity in road networks. In *17th ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems*, pages 23–32, 2009.

[87] D. Eppstein and S. Gupta. Crossing patterns in nonplanar road networks. In *25th ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems*, 09 2017.

[88] M. Erwig. The graph Voronoi diagram with applications. *Networks*, 36(3):156–163, 2000.

[89] T.-P. Fang and L. A. Piegl. Delaunay triangulation using a uniform grid. *IEEE Computer Graphics and Applications*, 13(3):36–47, 1993.

[90] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000.

[91] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[92] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2(2):153–174, 1987.

[93] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.

[94] A. Frieze and W. Szpankowski. Greedy algorithms for the shortest common superstring that are asymptotically optimal. *Algorithmica*, 21(1):21–36, 1998.

[95] A. M. Frieze, G. L. Miller, and S.-H. Teng. Separator based parallel divide and conquer in computational geometry. In *4th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 420–429, 1992.

[96] D. Frigioni and G. F. Italiano. Dynamically switching vertices in planar graphs. *Algorithmica*, 28(1):76–103, 2000.

[97] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[98] P. Gawrychowski, S. Mozes, O. Weimann, and C. Wulff-Nilsen. Better tradeoffs for exact distance oracles in planar graphs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '18, pages 515–529, Philadelphia, PA, USA, 2018. Society for Industrial and Applied Mathematics.

[99] M. Gibson, G. Kanade, E. Krohn, I. A. Pirwani, and K. Varadarajan. On clustering to minimize the sum of radii. *SIAM J. Comput.*, 41(1):47–60, Jan. 2012.

[100] J. R. Gilbert, J. P. Hutchinson, and R. E. Tarjan. A separator theorem for graphs of bounded genus. *Journal of Algorithms*, 5(3):391–407, 1984.

[101] Y. A. Gonczarowski, N. Nisan, R. Ostrovsky, and W. Rosenbaum. A stable marriage requires communication. In *Proceedings of the Twenty-sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '15, pages 1003–1017, Philadelphia, PA, USA, 2015. Society for Industrial and Applied Mathematics.

[102] M. T. Goodrich. Planar separators and parallel polygon triangulation. *Journal of Computer and System Sciences*, 51(3):374–389, 1995.

[103] M. T. Goodrich and R. Tamassia. Dynamic ray shooting and shortest paths via balanced geodesic triangulations. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, SCG '93, pages 318–327, New York, NY, USA, 1993. ACM.

[104] M. T. Goodrich and R. Tamassia. *Algorithm Design and Applications*. Wiley, Hoboken, NJ, 1st edition, 2014.

[105] I. G. Gowda, D. G. Kirkpatrick, D. T. Lee, and A. Naamad. Dynamic Voronoi diagrams. *IEEE Transactions on Information Theory*, 29(5):724–731, September 1983.

[106] D. H. Greene and F. F. Yao. Finite-resolution computational geometry. In *27th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 143–152, 1986.

[107] D. Gusfield and R. W. Irving. *The Stable Marriage Problem: Structure and Algorithms*. MIT Press, Cambridge, MA, USA, 1989.

[108] C. H. Papadimitriou. The Euclidean traveling salesman problem is NP-complete. *Theoretical Computer Science*, 4:237–244, 06 1977.

[109] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2003.

[110] M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997.

[111] J. Hershberger and S. Suri. Applications of a semi-dynamic convex hull algorithm. *BIT*, 32(2):249–267, May 1992.

[112] J.-H. Hoepman. Simple distributed weighted matchings. *CoRR*, cs.DC/0410047, 2004.

[113] C. Hoffman, A. E. Holroyd, and Y. Peres. A stable marriage of Poisson and Lebesgue. *Annals of Probability*, 34(4):1241–1272, 2006.

[114] S. Hoory, N. Linial, and A. Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, 2006.

[115] S. Huber and M. Held. Theoretical and practical results on straight skeletons of planar straight-line graphs. In *Proceedings of the Twenty-seventh Annual Symposium on Computational Geometry*, SoCG '11, pages 171–178, New York, NY, USA, 2011. ACM.

[116] S. Huber and M. Held. A fast straight-skeleton algorithm based on generalized motorcycle graphs. *International Journal of Computational Geometry & Applications*, 22(05):471–498, 2012.

[117] R. W. Irving. An efficient algorithm for the "stable roommates" problem. *Journal of Algorithms*, 6(4):577 – 595, 1985.

[118] K. Iwama and S. Miyazaki. A survey of the stable marriage problem and its variants. In *IEEE Int. Conf. on Informatics Education and Research for Knowledge-Circulating Society (ICKS)*, pages 131–136, 2008.

[119] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[120] D. S. Johnson and L. A. McGeoch. The traveling salesman problem: A case study in local optimization. In E. H. L. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley and Sons, Chichester, United Kingdom, 1997.

[121] J. Juan. Programme de classification hiérarchique par l'algorithme de la recherche en chaıne des voisins réciproques. *Les Cahiers de l'Analyse des Données*, 7(2):219–225, 1982.

[122] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, 2002.

[123] H. Kaplan, W. Mulzer, L. Roditty, P. Seiferth, and M. Sharir. Dynamic planar Voronoi diagrams for general distance functions and their algorithmic applications. In *28th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 2495–2504, 2017.

[124] H. Kaplan and N. Shafrir. The greedy algorithm for shortest superstrings. *Information Processing Letters*, 93(1):13–17, 2005.

[125] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[126] K. Kawarabayashi and B. Reed. A separator theorem in minor-closed classes. In *51st IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 153–162, 2010.

[127] K. Kedem, R. Livné, J. Pach, and M. Sharir. On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles. *Discrete Comput. Geom.*, 1(1):59–71, 1986.

[128] J. M. Keil. *Computational Geometry on an Integer Grid*. PhD thesis, University of British Columbia, 1980.

[129] A. Khan and A. Pothen. A new $3/2$-approximation algorithm for the b-edge cover problem. In *2016 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing*, pages 52–61. SIAM, 2016.

[130] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *40th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 81–89, 1999.

[131] K. Kise, A. Sato, and M. Iwata. Segmentation of page images using the area Voronoi diagram. *Computer Vision and Image Understanding*, 70(3):370–382, 1998.

[132] M. Künnemann, D. Moeller, R. Paturi, and S. Schneider. Subquadratic algorithms for succinct stable matching. *Algorithmica*, 81:2991–3024, 2019.

[133] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, MA, 2nd edition, 1998.

[134] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 1956.

[135] H. W. Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[136] E. Lawler. *The Travelling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley-Interscience series in discrete mathematics and optimization. John Wiley & Sons, 1985.

[137] N. Lev-Tov and D. Peleg. Polynomial time approximation schemes for base station coverage with minimum total radii. *Comput. Netw.*, 47(4):489–501, Mar. 2005.

[138] B. Liang and Z. J. Haas. Virtual backbone generation and maintenance in ad hoc network mobility management. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1293–1302, March 2000.

[139] D. R. Lick and A. T. White. k-degenerate graphs. *Canadian Journal of Mathematics*, 22(5):1082–1096, 1970.

[140] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[141] B. Ma. Why greed works for shortest common superstring problem. *Theoretical Computer Science*, 410(51):5374–5381, 2009.

[142] L. Ma. *Bisectors and Voronoi Diagrams for Convex Distance Functions*. PhD thesis, FernUniversität Hagen, 2000.

[143] J. Mackenzie. Gerrymandering and Legislator Efficiency. 2009.

[144] J. Matoušek and O. Schwarzkopf. Linear optimization queries. In *Proceedings of the Eighth Annual Symposium on Computational Geometry*, SCG '92, pages 16–25, New York, NY, USA, 1992. ACM.

[145] S. Meguerdichian, F. Koushanfar, G. Qu, and M. Potkonjak. Exposure in wireless ad-hoc sensor networks. In *7th Int. Conf. on Mobile Computing and Networking (MobiCom)*, pages 139–150. ACM, 2001.

[146] S. Micali and V. V. Vazirani. An $O(\sqrt{|v|}\cdot|E|)$ Algorithm for Finding Maximum Matching in General Graphs. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*, SFCS '80, pages 17–27, Washington, DC, USA, 1980. IEEE Computer Society.

[147] A. Misevicius and A. Blazinskas. Combining 2-opt, 3-opt and 4-opt with k-swap-kick perturbations for the traveling salesman problem. *17th International Conference on Information and Software Technologies*, 2011.

[148] H. Mittelmann and F. Vallentin. High-accuracy semidefinite programming bounds for kissing numbers. *Experimental Mathematics*, 19(2):175–179, 2010.

[149] P. Moscato and M. G. Norman. On the performance of heuristics on finite and infinite fractal instances of the euclidean traveling salesman problem. *INFORMS Journal on Computing*, 10(2):121–132, 1998.

[150] D. Müllner. Modern hierarchical, agglomerative clustering algorithms. *arXiv e-prints*, page arXiv:1109.2378, Sep 2011.

[151] F. Murtagh. A survey of recent advances in hierarchical clustering algorithms. *The Computer Journal*, 26(4):354–359, 1983.

[152] F. Murtagh and P. Contreras. Algorithms for hierarchical clustering: an overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(1):86–97, 2012.

[153] National Resident Matching Program, 2017.

[154] R. G. Niemi and J. Deegan. A Theory of Political Districting. *American Political Science Review*, 72(4):1304ÔÇô–1323, 1978.

[155] J. M. Oliva, M. Perrin, and S. Coquillart. 3d reconstruction of complex polyhedral shapes from contours using a simplified generalized Voronoi diagram. *Computer Graphics Forum*, 15(3):397–408, 1996.

[156] H. L. Ong and J. B. Moore. Worst-case analysis of two travelling salesman heuristics. *Operations Research Letters*, 2(6):273 – 277, 1984.

[157] M. H. Overmars. Computational geometry on a grid: an overview. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, volume 40 of *NATO ASI Series*, pages 167–184. Springer, 1988.

[158] M. H. Overmars. Efficient data structures for range searching on a grid. *Journal of Algorithms*, 9(2):254–275, 1988.

[159] K. Paluch. Better Approximation Algorithms for Maximum Asymmetric Traveling Salesman and Shortest Superstring. *arXiv e-prints*, page arXiv:1401.3670, Jan 2014.

[160] L. Pedersen and H. Wang. On the coverage of points in the plane by disks centered at a line. In *Proceedings of the 30th Canadian Conference on Computational Geometry, CCCG 2018, August 8-10, 2018, University of Manitoba, Winnipeg, Manitoba, Canada*, pages 158–164, 2018.

[161] M. Petřek, P. Košinová, J. Koča, and M. Otyepka. MOLE: A Voronoi diagram-based explorer of molecular channels, pores, and tunnels. *Structure*, 15(11):1357–1363, 2007.

[162] R. Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In C. Meinel and S. Tison, editors, *STACS 99*, pages 259–269, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[163] M. S. Rahman, S. Nakano, and T. Nishizeki. Rectangular grid drawings of plane graphs. *Computational Geometry*, 10(3):203–220, 1998.

[164] C. S. ReVelle and H. A. Eiselt. Location analysis: A synthesis and survey. *European Journal of Operational Research*, 165(1):1–19, 2005.

[165] F. Ricca, A. Scozzari, and B. Simeone. Weighted Voronoi region algorithms for political districting. *Mathematical and Computer Modelling*, 48(9-10):1468–1477, 2008.

[166] K. Räihä and E. Ukkonen. The shortest common supersequence problem over binary alphabet is np-complete. *Theoretical Computer Science*, 16(2):187 – 198, 1981.

[167] N. Robertson and P. Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49 – 64, 1984.

[168] L. Roditty and U. Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011.

[169] A. E. Roth and M. Sotomayor. The college admissions problem revisited. *Econometrica*, 57(3):559–570, 1989.

[170] S. Sakai, M. Togasaki, and K. Yamazaki. A note on greedy algorithms for the maximum weighted independent set problem. *Discrete Applied Mathematics*, 126(2-3):313–322, 2003.

[171] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, MA, 1990.

[172] M. I. Shamos. Geometric complexity. In *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*, STOC '75, pages 224–233, New York, NY, USA, 1975. ACM.

[173] M. I. Shamos and D. Hoey. Closest-point problems. In *16th Annual Symposium on Foundations of Computer Science*, pages 151–162, Oct 1975.

[174] L. Shapley and H. Scarf. On cores and indivisibility. *Journal of mathematical economics*, 1(1):23–37, 1974.

[175] M. Solbrig. Mathematical Aspects of Gerrymandering. Master's thesis, University of Washington, 2013.

[176] I. Stojmenović, A. P. Ruhil, and D. K. Lobiyal. Voronoi diagram and convex hull based geocasting and routing in wireless networks. *Wireless Communications and Mobile Computing*, 6(2):247–258, 2006.

[177] J. Tarhio and E. Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical computer science*, 57(1):131–145, 1988.

[178] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[179] J. S. Turner. Approximation algorithms for the shortest common superstring problem. *Information and computation*, 83(1):1–20, 1989.

[180] E. Ukkonen. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5(1-4):313–323, 1990.

[181] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *16th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 75–84, 1975.

[182] V. V. Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.

[183] A. Vigneron and L. Yan. A faster algorithm for computing motorcycle graphs. *Discrete Comput. Geom.*, 52(3):492–514, Oct. 2014.

[184] E. Weiszfeld and F. Plastria. On the point for which the sum of the distances to n given points is minimum. *Annals of Operations Research*, 167(1):7–41, 2009. Translation with annotations of E. Weiszfeld, Sur le point pour lequel la somme des distances de n points donnés est minimum, *Tôhoku Mathematical Journal* (first series), 43 (1937) pp. 355–386.

[185] E. Yang and Z. Zhang. The shortest common superstring problem: Average case analysis for both exact and approximate matching. *IEEE Transactions on Information Theory*, 45(6):1867–1886, 1999.