UNIVERSITY OF CALIFORNIA
RIVERSIDE

# Algorithms for Constrained Route Planning in Road Networks

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Michael Norris Rice

December 2013

The Dissertation of Michael Norris Rice is approved:

Committee Chairperson

University of California, Riverside

## Acknowledgments

First, I would like to thank my advisor, Vassilis Tsotras, for having the unexpected confidence in me to take me on as a student, in spite of my work schedule away from school. Thank you for always allowing me the freedom to explore my own path, and for your unwavering support and guidance in all matters, both personal and professional. Similarly, I wish to thank my dissertation committee members, Marek Chrobak, Neal Young, and Eamonn Keogh for their kind and helpful advice.

I would like to thank my work colleagues at ESRI for allowing me the opportunity to pursue this goal away from work, and for their continued patience and support during this process. Special thanks to Patrick Stevens for the constant encouragement, support, and great friendship established throughout our many conversational walks over these past few years (without which, my sanity may not have otherwise survived).

I would also like to thank the many incredible scientists who have laid the groundwork in this exciting field of research, some of whom I have had the privilege and honor to meet and work with, and who provided the initial inspiration for me to return to school and pursue this goal: Wee-Liang Heng, Peter Sanders, Andrew Goldberg, Renato Werneck, Dorothea Wagner, Dominik Schultes, Daniel Delling, and Robert Geisberger. I cannot thank you enough for your many wonderful influences and illuminations.

Finally, I would like to thank my family for their constant sacrifice in allowing me the time away from them to complete this journey. Most importantly, I would like to thank my loving wife, Christy, who was been my strongest supporter and greatest friend throughout this entire experience. I could not have done this without you.

To my beautiful wife, Christy.

ABSTRACT OF THE DISSERTATION

Algorithms for Constrained Route Planning in Road Networks

by

Michael Norris Rice

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2013
Dr. Vassilis J. Tsotras, Chairperson

This dissertation examines advanced pre-processing techniques and query algorithms for efficiently solving several practical, real-world route planning problems for personalized navigation in road networks. Unlike most prior research in this domain, where shortest paths are assumed to be static, this research supports dynamically-constrained shortest path queries in which the resulting solution paths can vary depending on each traveler's unique requirements. Specifically, this research is comprised of two parts, each focusing on a high-level class of constrained route planning problems in road networks. In the first part, we consider route planning problems with avoidance constraints, in which the route is required to avoid certain types of roads (e.g., toll roads, unpaved roads, etc.). In the second part, we consider route planning problems with detour constraints, in which the route is required to visit one or more categorical points of interest along the way (e.g., gas stations, coffee shops, etc.). We consider several specific sub-problems for each of these high-level problems and present extensive experimental results of our proposed algorithms on the continental road networks of North America and Europe (with over 50 million edges and 90 million edges, respectively).

# Contents

# List of Figures

# List of Tables

# Part I

# Foundations

# Chapter 1

# Introduction

Route planning in road networks is a common task in everyday life, whereby travelers must effectively navigate their way through local streets in order to reach their destination in the most efficient manner possible. Contemporary algorithmic approaches for route planning typically involve modeling the road network as a graph whose edges represent individual streets connected at nodes representing the street intersections. The weight, or cost, of these edges can represent any measure of interest to be optimized over the course of the intended route, such as the travel time, distance, fuel consumption, or monetary cost to traverse each edge (i.e., street). Given this graph-theoretical model, classical graph search algorithms, such as the well-known Dijkstra's algorithm [39], are capable of solving many common route planning problems to optimality.

However, such standard algorithms are unfortunately still quite impractical in practice when applied to very large-scale, real-world road networks (e.g., the North American road network with over 50 million edges), as they may often require searching nearly the entire graph to find the optimal solution path[1], thus taking too much time to

[1]For example, Dijkstra's algorithm processes nodes in order of increasing cost (e.g., distance or time) from the source until it reaches the destination. All nodes closer to the source than the destination is will therefore be processed during the search.

compute. Given such scalability limits, and due to the increasing presence of personal navigation systems and applications over the last decade, much research focus has recently been given to more efficiently solving route planning problems in such large-scale, real-world road networks.

Recent pioneering efforts from within the algorithm engineering community (e.g., see [5]) have focused on greatly improving the performance and scalability of such standard algorithms via an increasingly-dominant paradigm in algorithm design, based on a common two-phased algorithmic approach. This approach consists of an *offline* pre-processing (or indexing) phase to compute a set of auxiliary data and an *online* query phase intended to utilize this auxiliary data to speed up the required processing. The pre-processing phase is typically only required once, and can thus incur significantly more space and time overhead than the query phase, if necessary. The query phase can occur as often as necessary and is intended to be as fast and efficient as possible.

Given that most road networks are fairly static in nature (i.e., it is infrequent that new roads are added or old roads are removed), such a two-phased approach makes practical sense for enabling more efficient route planning algorithms in road networks. However, most work in this particular domain to-date has focused solely on simple route planning (e.g., find only a *least-cost*, or "shortest"[2], overall path), whereas most travelers tend to have very specific and unique travel requirements, in practice. For example, some travelers may wish to avoid toll roads to avoid the toll fee, even if taking the toll road would result in less travel time or a shorter travel distance to the destination. Likewise, depending on the traveler's vehicle dimensions, some roads may be considered infeasible for travel (e.g., a compact car may pass easily under a low highway overpass, but a large

[2]The phrase "shortest path" is commonly used by convention in the literature (and throughout this dissertation) to refer to a least-cost path for any cost measure of interest (even those which are not based on actual travel distance).

truck pulling a trailer may be too tall to pass without incurring damage to the vehicle or overpass). For such cases, the route planning process involves dynamically restricting certain types of edges (streets) from being allowed in the resulting solution path.

Additional route planning scenarios which are dynamically-constrained by specific traveler requirements include so-called errand-scheduling problems in which the traveler wishes to complete several errands along the way to their destination. For example, a traveler may wish to stop by an ATM and a coffee shop on the way to their destination, taking the minimum possible detour along the way (note that there may be many ATMs and/or coffee shops to choose from). As we shall see, some versions of this problem also turn out to be NP-hard and thus cannot easily be solved using existing route-planning techniques.

In the remainder of this work, we address several of these common constrained route-planning problems. Part II of this dissertation (summarized in Section 1.1) presents our work on designing practical graph indexing and query algorithms for efficiently supporting route planning with avoidance constraints, enforced by dynamic edge restrictions. Part III of this dissertation (summarized in Section 1.2) presents our work on designing efficient algorithms (both exact and approximate) for solving detour-constrained route-planning problems, such as errand scheduling and other related problems.

## 1.1 Route Planning with Avoidance Constraints

In Part II, we explore a new type of constrained shortest-path queries, based on avoidance constraints, in which the query can be dynamically parameterized to restrict the type of edges which may be included in the resulting shortest path. More specifically, each edge has an associated set of Boolean and/or scalar values representing both qualitative and quantitative information about the edge, respectively (e.g., Does the edge represent a toll road (yes/no)?, What is the maximum-allowed vehicle height for the edge (in meters)?). The query then specifies a constraint for each value type (e.g., restrict toll roads and any roads with vehicle height clearance under 3 meters), thus limiting the edges which may be considered valid for a solution path. To the best of our knowledge, this is the first work to address this practical variant of shortest path query. For this research, we further distinguish between two types of edge-restricted shortest path queries, based on two different classes of restrictions: *label restrictions* and *parameterized restrictions.*

Label restrictions (discussed in Chapter 3) represent the set of *qualitative* constraints which may be applied to route planning in the road network. In this context, the graph is defined with a fixed set of possible edge classes (e.g., highways, toll roads, ferries, unpaved roads, etc.). Each edge is then assigned a subset of "labels" to indicate which class(es) it belongs to in the graph. A shortest-path query can then be dynamically adjusted according to which labels (i.e., classes) of edges should be excluded from the resulting shortest path. For example, a traveler may wish to find the shortest path between two locations which avoids both toll roads and ferries, or, alternatively, they might prefer to find the shortest path which avoids only unpaved roads.

Parameterized restrictions (discussed in Chapter 4)[3] represent the set of *quantitative* constraints which may be applied to route planning in the road network. For parameterized restrictions, the graph is defined with a fixed set of parameter types (e.g., vehicle height, vehicle weight), whose values are to be specified at query time. Each edge is then assigned a threshold value defining an absolute limit for each parameter type,

[3]This chapter is based on work done in collaboration with Robert Geisberger and Peter Sanders of the Karlsruhe Institute of Technology (KIT).

beyond which the edge will be restricted for that parameter type. For example, if an edge represents a section of road which travels under an overpass or through a tunnel, then its “vehicle height” threshold value will be equal to that of the height clearance for that overpass or tunnel, so that no vehicle taller than this threshold can be routed on that section of road. A shortest-path query can then be dynamically adjusted by providing different combinations of parameter values for each parameter type (e.g., different vehicle types will have different heights, weights, etc., resulting in potentially different shortest paths between the same pairs of locations).

## 1.2 Route Planning with Detour Constraints

Within the last decade, the growing online presence of geospatial information systems has made possible many novel applications in the fields of transportation and location-based services. Many massive, online location databases are now being made publicly available for mining spatial locations based on categorical points of interest, thus paving the way for highly-advanced navigation solutions.

As an example, consider a traveler in a new city for the first time. On their way to do some sightseeing at a local attraction, they wish to visit a coffee shop, a gas station, and an ATM (in no particular order). However, there may be many such locations to choose from for each of these categorical location types. As the traveler likely does not care exactly *which* gas station, ATM, or coffee shop they visit (since each provides the same general type of service[4]), a desirable solution is then any path which visits one of each of these location types with the least overall detour on the way to the destination. Such a scenario is a common occurrence for everyday personal navigation needs, and

[4]Note that the locations are user-defined and may be made more specific, as necessary; e.g., only consider gas stations of a certain brand.

also has many additional applications in the logistics industry as well. For example, for long-haul trucking, the route may require multiple days of traveling long distances, necessitating the recurrent need for refueling, lodging, eating, etc. throughout the trip. Selecting the optimal location(s) amongst all of these options to minimize travel time and cost is an important part of the route planning process.

This basic problem is more commonly known as the Generalized Traveling Salesman Path Problem (GTSPP), and it is known to be NP-hard, as the solution must determine not only which location from each location type to visit, but also the optimal order in which to visit them. However, for many scenarios, the order in which each location type is visited during the trip can often be very important (or even required). For example, in the previous personal navigation scenario, the traveler might wish to refuel first if their vehicle is low on gas. Additional scenarios from logistics in which the visit order of each location type is crucial include deliveries which require specialized equipment, such as moving equipment or regulated containers (e.g., for storing temperature-sensitive materials). In such cases, the vehicle must first pick up the equipment along the way at any one of several pickup (or rental) locations, then pick up the package and make the delivery. Only after the delivery can the equipment then be returned at any one of the other pickup (or rental) locations. Problems such as these, for which the visit order of each location type is fixed, are called here Generalized Shortest Path (GSP) queries. Unlike GTSPP, which is NP-hard, GSP problems can be solved efficiently in polynomial time.

In Part III of this dissertation, we explore solutions to both GSP and GTSPP problem types. Chapter 5 presents our initial work on efficiently solving GSP queries, demonstrating that GSP algorithms may also be used to effectively approximate GTSPP queries as well. Chapter 6 presents our later work on designing exact algorithms for solving the more-practical GTSPP queries to optimality in road networks. Chapter 7 concludes our examination of GTSPP by considering effective pre-processing techniques and approximation algorithms which allow us to ultimately solve such practical queries in only a matter of milliseconds on the entire road network of North America.

# Chapter 2

# Background and Related Work

In this chapter, we briefly summarize the formal notation, background information, and related work referenced throughout the remainder of this dissertation. While each of the remaining chapters of this dissertation are intended to be fully self-contained, there are many recurring references and notational conventions which we present here as a general primer for the remainder of this work.

## 2.1 Graphs, Paths, and Cycles

A *graph* $G = (V, E)$ represents a set of entities called *nodes* (a.k.a., junctions or vertices), $V$, related by a set of *edges*, $E \subseteq V \times V$, such that $n = |V|$ and $m = |E|$. The edge set contains pairs of nodes, referenced as $(u, v) \in E$ for some $u, v \in V$, to represent some relation of interest (e.g., connectivity) between the pair of nodes. If the pairs are unordered (i.e., the relation is implicitly mutual), the graph is said to be *undirected.* If the pairs are ordered (i.e., the relation is explicit only in the given direction, from $u$ to $v$), the graph is said to be *directed.* For a directed graph, $G = (V, E)$, a *backward* (a.k.a., *reverse*) *graph*, $\bar{G} = (V, \bar{E})$, is the translation of $G$ in which all edge directions are reversed: $\bar{E} = \{(v, u) \mid (u, v) \in E\}$.

A *path* $P_{s,t} = \langle v_1, v_2, \ldots, v_q \rangle$ in a graph $G$ represents a traversal sequence of related (e.g., connected) nodes starting at a *source node*, $s = v_1 \in V$, and ending at a *target node*, $t = v_q \in V$, such that, for $1 \leq i < q$, $(v_i, v_{i+1}) \in E$. In some contexts, we may alternatively reference a path by its constituent, consecutive edges (instead of its nodes) as $P_{s,t} = \langle e_1, e_2, \ldots, e_q \rangle$, such that, for $1 \leq i \leq q$, $e_i \in E$. A *simple path* is any path which does not traverse any node more than once in the sequence (i.e., each node in the path is distinct). A *non-simple path* traverses one or more nodes more than once.

A *cycle* in a graph is any non-simple path $P_{s,s} = \langle v_1, \ldots, v_q, v_1 \rangle$ which starts and ends at the same node $s = v_1 \in V$. A directed graph is said to be a *directed acyclic graph* (DAG) iff the graph contains no cycles; i.e., $\forall\, s, t \in V$, $\exists\, P_{s,t} \Rightarrow \nexists\, P_{t,s}$. A *topological ordering* of a DAG is any sequential ordering of the nodes, such that, if $(u, v) \in E$, $u$ comes before $v$ in the ordering. A *reverse topological ordering* is simply a topological ordering of the reverse graph of a DAG.

A graph $G = (V, E, w)$ is said to be a *weighted graph* when provided with an additional *weight function*, $w : E \to \mathbb{R}$, mapping each edge to a real value. Unless stated otherwise, we shall assume that all weight functions are positive (i.e., $w : E \to \mathbb{R}_{>0}$).

In a weighted graph, the weight, or cost, of a path $P_{s,t} = \langle e_1, e_2, \ldots, e_q \rangle$ is given as $w(P_{s,t}) = \sum_{1 \leq i \leq q} w(e_i)$. For any given $s, t \in V$, we denote a *minimum-weight* (a.k.a., least-cost or "shortest") path as $P^*_{s,t}$, such that, $\forall\, P_{s,t}$ in $G$, $w(P^*_{s,t}) \leq w(P_{s,t})$. Note that there may be many such minimum-weight paths, and ties may be broken arbitrarily. We formally reference the *shortest-path cost* (a.k.a., *distance*) between any pair of nodes, $s, t \in V$, as $d(s, t) = w(P^*_{s,t})$.

In this work, our graphs of interest are those representing real-world road networks, where the edges represent streets connected at nodes representing the street intersections. Our edge weights represent the travel time (in minutes) for each street.

## 2.2 Graph Search Algorithms

In this section, we summarize many of the classical graph search algorithms referenced throughout this dissertation. The algorithms presented here require no preprocessing. To simplify our discussion, we typically present each search algorithm as if we are only seeking to determine the existence or cost of some path (possibly a shortest path), and not the path itself. However, for all algorithms presented here, it is quite straightforward to extend them to additionally maintain parent-node pointers for reconstructing either the entire search tree or just the desired solution path(s), as necessary.

**Backward Search.** While not necessarily a unique search algorithm in and of itself, this concept remains relevant for all subsequent graph search algorithms discussed below. A *backward search* in this context is simply any of the following search algorithms applied to the reverse graph, $\bar{G}$ (as defined earlier), instead of the original graph, $G$.

**Breadth-First Search (BFS).** Breadth-first search (BFS) is a search algorithm which establishes paths based on the fewest number of edges from a source node to the other nodes in the graph. BFS is typically used to determine basic connectivity (i.e., Does a path even exist?) between nodes, as well as to find shortest paths in unweighted graphs (in which all edges are assumed to have equivalent, unit-weight cost). BFS takes as input a graph $G = (V, E)$, a source node $s \in V$, and (optionally) a set of target nodes $T \subseteq V$. The algorithm maintains for each node, $v \in V$, a *hop count*, $c(v)$, to represent the minimum number of edges needed to reach $v$ via some path from the source node $s$. Initially, $c(v) = \infty$ for all $v \in V$ (i.e., all nodes are *unreached*, since no path has yet been found to them). The algorithm begins by assigning $c(s) = 0$ for the source node $s$, and inserts $s$ into a *queue* [33] data structure. At each iteration, a node

$u$ is removed from the front of the queue. Upon removing $u$, for each $(u, v) \in E$ such that $c(v) = \infty$ (i.e., $v$ is unreached), the algorithm sets $c(v) = c(u) + 1$ and adds $v$ to the back of the queue. The algorithm may terminate once all nodes from the target node set $T$ have been popped from the queue, or whenever the queue is empty, whichever comes first. The time complexity of BFS is $O(m + n)$ [33].

**Depth-First Search (DFS).** Depth-first search (DFS) is another search algorithm which may be used to determine basic connectivity between nodes, but may also be used to detect cycles in graphs or establish topological orderings for directed acyclic graphs. The general concept is to search as "deep" into the graph as possible before backtracking to continue the search. DFS takes as input a graph $G = (V, E)$ and a source node $s \in V$. Initially, all nodes are marked as *unreached*, since no path has yet been found to them. The algorithm begins by marking $s$ as *reached* and inserting $s$ into a *stack* [33] data structure. At each iteration, the node $u$ at the top of the stack is examined (but not yet removed). If $\exists(u, v) \in E$ such that $v$ is unreached, the algorithm marks (one such) $v$ as reached, adds $v$ to the top of the stack, and begins another iteration. Otherwise, if no unreached adjacent nodes remain, $u$ is removed from the top of the stack. The algorithm may terminate once the stack is empty. The time complexity of DFS is $O(m + n)$ [33].

As alluded to earlier, DFS may be used to establish a topological ordering for directed ayclic graphs. To achieve this, we may simply loop over all nodes in the graph, and, at each iteration, if the current node $v \in V$ is still *unreached*, we perform a DFS from $v$. This loop continues until all nodes have been marked as *reached*. Upon completion, the order in which the nodes were removed from the stack defines a reverse topological ordering for the directed acyclic graph. This ordering may simply be reversed to establish a standard (i.e., forward) topological ordering of the explored acyclic graph.

**Dijkstra Search.** Dijkstra's algorithm [39] takes as input a weighted[1] graph $G = (V, E, w)$, a source node $s \in V$, and (optionally) a set of target nodes $T \subseteq V$. The algorithm maintains for each node, $v \in V$, a cost, or "distance", $d(v)$, to represent the total cost of the current-best path found from the source node $s$ to node $v$. Initially, for all $v \in V$, $d(v) = \infty$, and all nodes are marked as *unsettled* (i.e., their costs are not yet known to be correct). The algorithm begins by assigning $d(s) = 0$ and inserting $s$ into a set $F$ (the "fringe" of the search). At each iteration, the algorithm removes from $F$ a node, $u$, with minimum $d$-value, marks $u$ as *settled* (since $d(u) = d(s, u)$ is now provably true [33]), and *expands* $u$ as follows. For all $e = (u, v) \in E$, if $d(v) > d(u) + w(e)$, the algorithm sets $d(v) = d(u) + w(e)$ and $F = F \cup \{v\}$. The algorithm may terminate once all nodes from the target node set $T$ have been settled (i.e., their shortest-path costs are correct), or whenever $F$ is empty, whichever comes first.

A naïve implementation of Dijkstra's algorithm requires a worst-case time complexity of $O(n^2)$, since we have up to $n$ iterations, and a linear scan of the unsettled nodes in $F$ at each iteration (to choose the one with minimum cost) takes $O(n)$ time. However, this can be greatly improved by using a so-called *priority queue* [33] data structure (e.g., binary heaps[33] or Fibonacci heaps[49]) to maintain and access all $F$ nodes in order of increasing cost from the source. Such data structures improve the time complexity to $O((m + n)logn)$ for binary heaps or $O(m + nlogn)$ for Fibonacci heaps.

**Bidirectional Dijkstra Search.** A *bidirectional* version of Dijkstra's algorithm takes as input a weighted graph $G = (V, E, w)$, a source node $s \in V$, and a (single) target node $t \in V$. Bidirectional Dijkstra involves carrying out two Dijkstra searches simultaneously: one forward Dijkstra search from the source node $s$ and one backward Dijkstra search

[1]Edge weights are assumed to be non-negative.

from the target node $t$. Let $d_s$ be the set of path costs maintained by the forward search (from $s$) and $d_t$ be the set of paths costs maintained by the backward search (from $t$). A value, $\gamma$, is used to maintain the cost of the best path seen thus far. Initially $\gamma = \infty$. The search alternates between the forward and backward search direction at each iteration. When a node $u$ is settled by the forward search, for all edges $(u, v) \in E$ such that $v$ has been settled by the backward search, the algorithm sets $\gamma = min\{\gamma,\ d_s(u) + w(u, v) + d_t(v)\}$. A similar update procedure is done for the backward search. The algorithm may terminate once any node has been settled by both directions, at which point $\gamma = d(s, t)$.

**A$^*$ Search.** A$^*$ Search [62] takes as input a weighted graph $G = (V, E, w)$, a source node $s \in V$, and a (single) target node $t \in V$. The general concept is to quickly *direct* the search towards the target node $t$ by using some rough estimate on the remaining cost to $t$. The search maintains for each reached node, $v$, an $f$-value, computed as $f(v) = d(v) + h(v)$, where $d(v)$ represents the current best-known path cost from $s$ to $v$, and $h(v)$, defined as the *heuristic function*[2], represents an estimate on the shortest path cost from $v$ to $t$. The search proceeds in much the same fashion as a Dijkstra search, except that, instead of choosing a node with minimum $d$-value at each iteration, it chooses a node with minimum $f$-value. The algorithm may terminate once $t$ has been removed from $F$ for expansion, or whenever $F$ is empty, whichever comes first.

The correctness and time complexity of A$^*$ search depends primarily on the properties of the chosen heuristic function, $h$. A$^*$ will return the correct shortest-path cost to $t$ if the function $h$ is *admissible*: $\forall v \in V$, $h(v) \leq d(v, t)$ (i.e., $h$ never overestimates the true shortest-path cost). A$^*$ will expand each node at most once in the search if the function $h$ is *consistent*: $\forall (u, v) \in E$, $h(u) \leq w(u, v) + h(v)$. Consistency ensures that

[2]This function is also referred to as the *potential function* in some contexts.

we may still mark a node $v$ as *settled* once it is removed from $F$, as $d(v) = d(s, v)$ will be correctly established by this time. However, inconsistent functions do not maintain this property and thus may require re-inserting nodes back into the set $F$ multiple times.

**Bidirectional A$^*$ Search.** The original A$^*$ concept was later further extended to bidirectional A$^*$ search in [91], where two separate unidirectional A$^*$ searches are carried out simultaneously from both nodes $s$ and $t$. A forward A$^*$ search is carried out from node $s$ (toward goal node $t$) and a backward A$^*$ search is carried out from node $t$ (toward goal node $s$). Each search direction must make use of its own distinct heuristic function, given that each direction is searching towards a different goal node. We denote the heuristic search functions as $h_s$ and $h_t$ for the forward and backward search directions, respectively. In the context of bidirectional A$^*$ search, $h_s(v)$ behaves as before by providing a lower-bound estimate on the cost $d(v, t)$, whereas $h_t(v)$ provides a lower-bound estimate on the cost $d(s, v)$. For forward search, the $f$-values are computed as $f_s(v) = d_s(v) + h_s(v)$. For backward search, $f_t(v) = d_t(v) + h_t(v)$. Similarly, each search direction maintains its own set of "fringe" nodes (described earlier): $F_s$ and $F_t$.

The search proceeds in much the same fashion as a bidirectional Dijkstra search, iteratively swapping between search directions, and keeping track of a tentative best-path cost, $\gamma$. The algorithm may terminate as soon as $max(k_s, k_t) \geq \gamma$, where $k_s = min\{f_s(v) \mid v \in F_s\}$ and $k_t = min\{f_t(v) \mid v \in F_t\}$, ensuring that $\gamma = d(s, t)$ [91]. This termination criterion is correct for any admissible heuristic functions, $h_s$ and $h_t$ (even inconsistent ones, as long as node re-insertion into $F_{s/t}$ is allowed, as necessary).

An alternative bidirectional A$^*$ algorithm is presented in [66]. The authors prove that, by using a set of *balanced*[3] and consistent heuristic functions, $h'_s$ and $h'_t$,

[3]Such functions have also been commonly referred to as *consistent* heuristic functions in [54]. However, this conflicts with the original use of the term consistency in the historical context of A$^*$ literature.

such that, for all nodes $v \in V$, $h'_s(v) + h'_t(v) = c$ (for some constant, $c$), then the search may terminate as soon as the first node has been settled in both directions. The authors further demonstrate a simple approach for deriving such balanced heuristic functions by using $h'_s(v) = (h_s(v) - h_t(v))/2$ and $h'_t(v) = (h_t(v) - h_s(v))/2 = -h'_s(v)$, where $h_s$ and $h_t$ are arbitrarily-defined consistent functions (note the requirement of consistency).

## 2.3 Graph Search Algorithms with Preprocessing

In this section, we examine various graph search algorithms for finding shortest paths which incorporate a separate preprocessing phase to speedup the associated queries. We consider three high-level categories of preprocessing-based search algorithms: goal-directed, hierarchical, and hybrid search (i.e., combinations of the former).

### 2.3.1 Goal-Directed Search with Preprocessing

The first category of preprocessing is based on the concept of *goal-directed search*, in which the algorithm uses its preprocessed data to quickly guide, or "direct", the search towards the target node (a.k.a., the "goal"). We give various examples below.

**A$^*$ Search + Landmarks + Triangle Inequality (ALT).** ALT [54] involves preprocessing which selects a small subset of so-called *landmark* nodes, $L \subseteq V$, typically such that $|L| \ll |V|$. For each landmark, $l \in L$, the preprocessing step computes and stores the shortest-path costs from and to all other nodes, $v$, in the graph: $d(v, l)$ and $d(l, v)$, respectively. After preprocessing, a bidirectional A$^*$ search may be carried out using heuristic functions derived from the preprocessed landmark costs: $h_s(v) = \underset{\forall l \in L'}{max}\{max\{d(v,l) - d(t,l), d(l,t) - d(l,v)\}\}$ and $h_t(v) = \underset{\forall l \in L'}{max}\{max\{d(s,l) -$

Therefore, we have adopted the term *balanced* from [90] to describe this property.

$d(v,l), d(l,v) - d(l,s)\}\}$, for some small landmark subset $L' \subseteq L$, chosen at query time.

**Geometric Containers (GC).** GC [112] preprocessing defines a bounding geometry (e.g., a minimum bounding rectangle [MBR]) for each edge in the graph, such that the bounding geometry contains all nodes that can be reached via some shortest path starting with that edge. This methodology requires some explicit spatial embedding of the nodes of the graph (e.g., geographic coordinates). The set of geometric containers can then be used by any shortest-path search algorithm (e.g., Dijkstra's algorithm) to ignore any edge whose container (e.g., MBR) does not geometrically contain the target node. Pre-processing requires a computation of all-pairs shortest paths.

**Arc Flags.** Arc-Flags [77] preprocessing partitions the nodes of the graph into $k$ disjoint regions and, for each edge in the graph, defines $k$ binary "flags" on the edge to indicate whether or not that edge belongs to some shortest path into each partition, respectively. The set of edge flags can then be used by any shortest-path search algorithm (e.g., Dijkstra's algorithm) to ignore any edge whose flag for the target node's partition is false (i.e., $= 0$), indicating that it cannot belong to any shortest path from $s$ to $t$. More efficient preprocessing approaches for deriving the arc flags are presented in [64].

**Reach.** Reach [60] preprocessing defines the reach of a node, $v$, with respect to some path $P_{s,t} = P_{s,v} \cdot P_{v,t}$ (where $\cdot$ indicates path concatenation) as $min\{w(P_{s,v}), w(P_{v,t})\}$. The (global) reach, $r(v)$, is then defined as the maximum over all shortest paths $P$ containing $v$ of the reach of $v$ with respect to $P$. Let $\underline{d}(u,v)$ be any lower-bound estimate on the shortest-path distance from $u$ to $v$, and let $\overline{r}(v)$ be any upper bound on the reach of $v$. Any shortest-path search algorithm (e.g., Dijkstra's algorithm) may prune a node $v$ from the search if $\overline{r}(v) < \underline{d}(s,v)$ and $\overline{r}(v) < \underline{d}(v,t)$. The calculation of reach values typically involves pre-calculating all-pairs shortest paths within the graph, which can be very expensive. More efficient approaches for reach calculation have also been established by [55, 56].

**Pre-Computed Cluster Distances (PCD).** PCD [82] partitions the nodes of the graph into $k$ disjoint partitions, or "clusters". The shortest connecting path distance between each pair of clusters is then pre-calculated and recorded in a distance table (here shortest path distance between clusters is defined as the minimum-overall shortest-path cost from some boundary node in one cluster to some other boundary node in the other cluster). This inter-cluster cost table may then be used as a lower bound cost estimate within the search. PCD performs a bidirectional Dijkstra search until an upper bound on the cost of the shortest path can be established (via either the searches meeting or based on costs to certain cluster-boundary nodes plus their inter-cluster distances), after which time the search may prune any node whose distance from the source node plus its partition's cost to the target node's partition is greater than the current upper bound.

### 2.3.2 Hierarchical Search with Preprocessing

The second category of preprocessing is based on the concept of *hierarchical search*, in which the nodes and/or edges of the graph are classified into mutually-exclusive *levels* of hierarchy. A bidirectional search (from $s$ and $t$, simultaneously) is typically carried out, such that each search leads toward higher (i.e., more-important) levels in the graph, while progressively ignoring lower (i.e., less-important) levels of the graph. The power of hierarchical search comes from this implicit pruning effect, which often significantly reduces the size of the resulting search space, to achieve much faster query times, (usually) without sacrificing the optimality of the resulting solution path.

**Multi-Level Overlays (MLO).** Given a weighted graph $G = (V, E, w)$ and some subset $S \subseteq V$, MLO [65] defines a *shortest-path overlay graph* $G' = (S, E', w)$ as follows. For each $(u, v) \in S \times S$, a *shortcut edge* $(u, v)$ is added to $E'$ iff no internal vertex on any shortest $u$-$v$ path belongs to $S$. In this context, a shortcut edge is a replacement of a shortest path $P^*_{u,v} = \langle u, \ldots, v \rangle$ by an edge $(u, v)$, such that $w(u, v) = d(u, v)$. By iteratively repeating this overlay graph construction on each newly-produced graph level, the procedure defines an explicit hierarchy of $k$ overlay graphs such that $V = S_0 \supset S_1 \supset S_2 \supset \ldots \supset S_k$. A shortest-path query may then be carried out in a small subgraph composed of only the necessary overlay-graph components, as determined by a so-called tree of connected components data structure.

**Highway Hierarchies (HH).** HH [101, 102] begins with the original graph as the lowest-level hierarchy. It proceeds by iteratively identifying edges that exist outside of the local shortest-path neighborhoods of the nodes in the current hierarchy of the graph and promoting such edges to the next level of the hierarchy. Specifically, an edge $(u, v)$ is considered a *highway edge* for the next level of hierarchy construction if there exists some shortest path $P^*_{s,t} = \langle s, \ldots, u, v, \ldots, t \rangle$ in the current hierarchy level, such that $v$ is not in the local shortest-path neighborhood of $s$ and $u$ is not in the local shortest-path neighborhood of $t$. A bidirectional query procedure can then progressively prune increasing levels of the hierarchy the farther along it gets in its search, without affecting the correctness of the results.

**Transit Node Routing (TNR).** TNR [19, 20] preprocessing establishes a relatively small set of so-called *transit* nodes, $T \subseteq V$, such that, for every pair of nodes which are "sufficiently far apart" (in terms of shortest-path distance) from each other, a shortest path between them contains one or more transit nodes. TNR pre-calculates the shortest-path distances between every pair of transit nodes. Any (non-local) shortest path query can then perform a bidirectional Dijkstra search until the forward shortest-path tree from $s$ is covered by some subset of transit nodes, $T_s \subseteq T$, and the backward shortest-path tree from $t$ is covered by some subset of transit nodes, $T_t \subseteq T$ (note that these subsets may also be preprocessed and stored for every node as well). The shortest path cost may then be computed as $d(s,t) = min\{d(s,u) + d(u,v) + d(v,t) \mid u \in T_s, v \in T_t\}$. Local queries (i.e., for those pairs of nodes which are too close together to be covered by $T$) may be handled by a standard bidirectional Dijkstra search.

**Contraction Hierarchies (CH).** CH [50, 53] preprocessing establishes a total ordering of the nodes in the graph, and then *contracts* (or *shortcuts*) the nodes in this order. To contract a node, $v$, means to bypass it in the graph by adding new *shortcut* edges, as necessary, to preserve shortest paths in the remaining, higher-ranking subgraph. Specifically, for each pair of incoming and outgoing edges, $(u,v)$ and $(v,x)$, leading from and to higher-ranking nodes, respectively, if the path $\langle u, v, x\rangle$ is a unique shortest path, then a new shortcut edge $(u,x)$, with weight $w(u,v) + w(v,x)$, is added to bypass $v$ in the higher-ranking subgraph. A bidirectional Dijkstra shortest-path search may then be carried out such that each search only explores edges leading to higher-ranking nodes. The search in a given direction may be aborted once its minimum unsettled node distance exceeds the cost of the best connecting path seen so far.

**Hub Labeling (HL).** The concepts of *Hub Labeling* have been examined in various forms in [7, 32]. For each node $v \in V$, the preprocessing must establish both a forward and backward *labeling* of the nodes, $L_f(v) \subseteq V$ and $L_b(v) \subseteq V$, respectively, with the following *shortest-path cover* property: $\forall s, t \in V$, $L_f(s) \cap L_b(t)$ contains at least one node along a shortest path from $s$ to $t$. For each $x \in L_f(v)$ (and for each $u \in L_b(v)$), the shortest-path cost $d(v, x)$ (respectively, $d(u, v)$) is also preprocessed and stored along with the labeling. After preprocessing, for any $s, t \in V$, the shortest-path cost from $s$ to $t$ may be computed as $d(s, t) = min\{d(s, v) + d(v, t) \mid v \in L_f(s) \cap L_b(t)\}$.

### 2.3.3 Hybrid Search with Preprocessing

The third, and final, category of preprocessing is based on a hybrid combination of aspects from each of the two previous categories: goal-directed and hierarchical search.

**Reach + ALT (REAL).** REAL [55, 56] integrates the previously-introduced concepts of Reach, shortcut edges, and ALT. By maintaining landmark data for only a relatively small subgraph of those nodes with the highest reach, the memory overhead may be greatly improved without significantly impacting performance.

**Shortcuts + Arc-Flags (SHARC).** SHARC [22] integrates Arc-Flags with the hierarchical concept of shortcut edges (obtained via contraction, to bypass less-important parts of the graph). The result is a very fast *unidirectional* shortest-path search algorithm, intended to help solve those shortest-path problem variants for which bidirectional search is typically impractical (e.g., *time-dependent* shortest path problems).

**Core ALT (CALT).** CALT [23] is a variant of ALT combined with a hierarchical technique, such as CH, in which landmark distances are established only for some relatively small *core* of the "uppermost" (i.e., most important) nodes in the hierarchy. This can be seen to greatly reduce the space overhead for storing landmark distances, as compared to the original ALT. For nodes not within the core, a multi-phase search algorithm utilizes their closest core nodes, known as their *proxy* nodes, to establish valid lower bounds for carrying out the remainder of the search using the preprocessed landmark distances in the core.

**CH + Arc-Flags (CHASE).** CHASE [23] combines CH and Arc-Flags by establishing a relatively small core of the highest nodes in the hierarchy (similar to CALT). Arc-Flags are preprocessed for the core subgraph, and a two-phase query algorithm is carried out by first performing a bidirectional Dijkstra search into the core and then switching to a bidirectional Arc-Flags search within the core, as necessary.

**Reach + Arc-Flags (ReachFlags).** ReachFlags [23] combines Reach and Arc-Flags by establishing a relatively small subgraph containing only those nodes with reach $\geq r$ (for some sufficiently-large $r \in \mathbb{R}_{>0}$). Arc-Flags are preprocessed for the smaller subgraph, and a two-phase query algorithm (similar to that for CHASE) is carried out.

**TNR + Arc-Flags (TNR+AF).** TNR+AF [23] combines the concepts of TNR and Arc-Flags. The preprocessing partitions the transit nodes of the graph into $k$ disjoint regions (similar to Arc-Flags), and establishes flags on related node/access-node pairs to indicate whether or not they are useful for finding a shortest path into each region, respectively. This helps to reduce the number of unnecessary table lookups performed by the standard TNR technique.

# Part II

# Route Planning with Avoidance Constraints

# Chapter 3

# Shortest Paths with Label Restrictions

## 3.1 Introduction

Due to its ubiquitous usage over the web and in many commercial navigation products, point-to-point shortest path search on graphs has again become a major topic of interest over the last decade, with much research being devoted to designing practical indexing techniques for extremely fast graph searches. Graph indexing techniques have been widely explored for establishing efficient data structures for pruning and/or directing the search of shortest path algorithms, while still guaranteeing the optimality of the resulting paths. Such techniques have resulted in many improvements over the standard Dijkstra's algorithm [39], and may also be used to minimize the overall I/O costs incurred by the graph search for very large, external-memory graph datasets [57, 104]. However, focus thus far has been mostly on static shortest paths with no constraints.

In this chapter, we focus on a variant of shortest path queries in which dynamic constraints may be placed upon the type of edges which may appear on a valid shortest path. For example, the shortest path from Irvine, CA to Riverside, CA travels along State Route 261, which is a local toll road through this area. However, consider the case where the traveler does not wish to pay the toll fee, and would therefore rather find the shortest path from Irvine to Riverside that actually avoids all toll roads. As yet another example, trucks delivering certain hazardous materials may not be allowed to cross over some types of roadways, such as bridges or railroad crossings, due to the public health and safety risks of any potential accidents. Therefore, this query type can be seen to have practical applications in both personalized location-based services, as well as in many logistics and commercial transportation scenarios. Making this query highly efficient on real-world, large-scale graphs, such as the road network of the continental United States, is therefore crucial to effectively supporting such practical applications.

### 3.1.1 Related Work

In recent years, hierarchical graph indexing techniques have been shown to be some of the most time- and space-efficient approaches towards indexing graphs for shortest path computations [23, 53, 65, 69, 102, 104, 105]. Hierarchical techniques generally involve some classification of the nodes/edges within the graph into mutually-exclusive, ordered levels of hierarchy, based on some notion of importance within the graph structure. Shortest path queries carried out on a hierarchical graph index typically prefer searching towards higher (i.e., more-important) levels of the graph hierarchy, while progressively ignoring lower (i.e., less-important) levels of the hierarchy, in order to more effectively reduce the overall search space explored by the query.

Schultes and Sanders [105] have previously explored a variant of their hierarchical indexing techniques designed to support dynamic changes in graph edge weights or cost functions. However, support for this dynamic approach requires either explicit recomputation of the graph index online as the weights (or cost functions) change or the query algorithm must make increasingly-limited use of the information available in the static graph index based on the dynamic changes.

Yet another practical graph indexing approach is the goal-directed approach of the ALT algorithm [54, 57]. The ALT algorithm is based primarily on the concepts of A* search [62], in which the search from a source node is “directed” towards the target node by the use of a potential function to estimate the shortest path cost to the target. The ALT algorithm allows preprocessing in which a set of so-called *landmark* nodes is selected from the graph and the shortest path is computed for each landmark node to/from all other nodes in the graph. Using properties of the triangle inequality derived from the costs to/from all landmark nodes, a highly efficient potential function can be constructed, thus greatly reducing the resulting search space. This technique has been further studied within the context of dynamic graphs in [36], and it can be shown that the potential functions from the original landmark preprocessing remain correct for all shortest paths as long as the edge weights can only *increase* in a dynamic scenario.

In the context of our own constrained shortest path query presented here, the idea of dynamically restricting an edge from being allowed in the search for a particular query can be seen as equivalent to simply increasing the weight of that edge to infinity for the lifetime of the query. Thus, the ALT technique is the only existing indexing technique applicable to our query type without requiring additional or specialized preprocessing.

### 3.1.2 Our Contributions

To the best of our knowledge, this is the first work to address this variant of shortest path query. In particular, our contributions can be summarized as follows. We formalize this problem as a restricted class of language constrained shortest paths, thus tying it to the existing literature and giving this new problem some relative context.

To efficiently support this type of dynamically constrained shortest path query, we detail a practical and efficient approach to extend the hierarchical graph indexing technique known as Contraction Hierarchies [50, 53]. Given implicit knowledge of the range of possible constraints for shortest path queries on a graph, we propose to incorporate this knowledge directly into the graph index construction to avoid the overhead of reconstructing the index for each possible constraint scenario at query time.

Using one of the largest commercial real-world road network datasets, we present experimental results with improvements of over 3 orders of magnitude compared to the naïve adaptation of the standard Dijkstra's algorithm[1] to support this query type. We also show an improvement of over 2 orders of magnitude compared to the dynamic ALT algorithm examined in [36].

The remainder of this chapter is organized as follows. In Section 3.2, we present the concept of constraints on the allowable edges for a given shortest path query as a specific variant of language constrained shortest paths. Section 3.3 presents an overview of Contraction Hierarchies. Section 3.4 extends this technique with the proposed algorithms for constructing and querying the hierarchical graph index to support these constraints for shortest path queries. Section 3.5 presents our experimental analysis of this technique. Section 3.6 concludes the chapter.

## 3.2 Language Constrained Shortest Paths

Language constrained shortest paths [18] are shortest paths whose edge labels must satisfy some formal language constraint over a fixed alphabet $\Sigma$. We define this concept more formally as follows. Let $G = (V, E, w, \Sigma, \ell)$ be a weighted directed graph,

[1]We refer here to the more efficient bidirectional version.

where $V$ is the set of vertices in $G$, $E$ is the set of edges in $G$, $w : E \to \mathbb{R}_{>0}$ is a function mapping edges in $G$ to a positive, real-valued weight, $\Sigma$ is a finite alphabet used for labeling of edges in $G$, and $\ell : E \to \Sigma$ is a function mapping edges in $G$ to a label in $\Sigma$.

Let $P_{s,t} = \langle e_1, e_2, \ldots, e_q \rangle$ be any path in $G$ from some vertex $s \in V$ to some vertex $t \in V$, such that $e_1 = (s, v_1) \in E$, $e_q = (v_{q-1}, t) \in E$, and for $1 < i < q$, $e_i = (v_{i-1}, v_i) \in E$. Let $w(P_{s,t}) = \sum_{1 \leq i \leq q} w(e_i)$ be the total weight of all edges in $P_{s,t}$. Let $\ell(P_{s,t}) = \ell(e_1)\ell(e_2)\ldots\ell(e_q)$ be the concatenation of the labels of all edges in $P_{s,t}$. Given any formal language $L \subseteq \Sigma^*$, a *language constrained shortest path* is a path $P'_{s,t}$ in $G$ such that $\ell(P'_{s,t}) \in L$ and $\forall$ $P_{s,t}$ in $G$ where $\ell(P_{s,t}) \in L$, $w(P'_{s,t}) \leq w(P_{s,t})$.

The *Regular Language Constrained Shortest Paths* (RLCSP) problem is a basic variant of language constrained shortest paths where the constraint language $L$ must be a regular language. In [15, 18], Barrett et al. show that RLCSP is solvable in polynomial time by performing a shortest path search in the product graph of the original graph and the non-deterministic finite automaton (NFA) graph representing the specified regular language. More specifically, a regular language $L$ can be represented by a non-deterministic finite automaton (NFA) $M = (Q, \Sigma, \delta, q_0, F)$, where $Q$ is the set of states, $\Sigma$ is the alphabet, $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is the transition function (where $\mathcal{P}(Q)$ denotes the *power set* of $Q$), $q_0$ is the starting state, and $F$ is the set of accepting states. Let $T$ represent the set of transitions between state pairs $q_i, q_j \in Q$ for some $\alpha \in \Sigma$ such that: $q_j \in \delta(q_i, \alpha) \Leftrightarrow \exists\, t = (q_i, q_j) \in T \wedge \ell(t) = \alpha$. Given any graph $G$ and NFA $M$, a product graph $G \times M = (V_{G,M}, E_{G,M})$ is constructed where $V_{G,M} = \{\langle v, q \rangle \mid v \in V \wedge q \in Q\}$, $E_{G,M} = \{(\langle u, q_i \rangle, \langle v, q_j \rangle) \mid e = (u, v) \in E \wedge t = (q_i, q_j) \in T \wedge \ell(e) = \ell(t)\}$, and $\forall(\langle u, q_i \rangle, \langle v, q_j \rangle) \in E_{G,M}$, $w(\langle u, q_i \rangle, \langle v, q_j \rangle) = w(u, v)$ [15]. For any origin $s \in V$ and destination $t \in V$, a RLCSP problem may be solved by finding the shortest path in the resulting product graph from $\langle s, q_0 \rangle \in V_{G,M}$ to some $\langle t, f \rangle \in V_{G,M}$ for some $f \in F$.

The *Linear Regular Expression* (LRE) constrained shortest paths problem [16] is a variation of RLCSP in which the regular expressions representing the constraint-language $L$ must be of a specific form related to a restricted subclass of regular languages. In particular, linear regular expressions must be of the form $x_1^+ x_2^+ \dots x_k^+$, where for $1 \leq i \leq k$, $x_i \in \Sigma$, and $x_i^+ = x_i x_i^*$.

LRE is presented primarily as a means of expressing modal constraints on real-world transportation networks, where a traveler knows the exact modes of travel (i.e., labels) they wish to consider and the exact order in which they wish to travel through these modes. As an example (taken from [15]), consider a multimodal network consisting of labels $\Sigma = \{c, b, p\}$, which represent travel modes for ($c$)ars, ($b$)uses, and ($p$)edestrians, respectively. A traveler may wish to walk to the bus station and take no more than one bus line (i.e., no transfers) before walking to their final destination. This could be specified using the linear regular expression $p^+ b^+ p^+$.

One drawback to this approach is that such information may not always be known by the traveler in advance. For example, the traveler may not always know the best order of modes to take in their trip; however, they are still likely to know exactly which modes they are ultimately *willing* to take (as well as those modes which they are *unwilling* to take). Therefore, we present a new variant of language constrained shortest paths (below) designed specifically to support this more flexible scenario.

### 3.2.1 Kleene Language Constrained Shortest Paths

We present the *Kleene Language Constrained Shortest Paths* (KLCSP) problem as a variant of language-constrained shortest paths based on another (simpler) subclass of regular languages which we shall call here the *Kleene languages* (note that this is not a formally-recognized language class, but is used here for convenience of discussion).

A *Kleene language* may be defined in this context as the Kleene closure of any subset of $\Sigma$. More formally, $\forall A \subseteq \Sigma$, $L(A^*)$ defines a Kleene language over alphabet $A$. Note that the subset alphabet $A$ merely defines the set of *allowable* labels that can appear on a valid shortest path for a KLCSP problem. However, unlike LRE, the labels in $A$ are not required to appear on a shortest path for a KLCSP problem and the sequence of the labels of such a path is irrelevant. Additionally, for any Kleene language over $A \subseteq \Sigma$, there is an implicitly defined subset of *restricted* labels $R = \Sigma \setminus A$, such that no labels in $R$ may appear on any valid KLCSP solution. A Kleene language over $A \subseteq \Sigma$ may therefore be equivalently defined simply by specifying the set of such *restricted* labels, $R$, where $A = \Sigma \setminus R$. Given this definition, the KLCSP problem is designed to support the specification of language constraints on the allowed (restricted) set of labels which may (not) appear over a given shortest path, in any permutation. It is considered more common in practice to specify this constraint as the set of restricted labels, $R$, so we will adopt this approach for the remainder of this chapter.

As a simple example, consider a transportation network consisting of labels $\Sigma = \{l, h, i, t, f\}$, which represent $(l)$ocal roads, $(h)$ighways, $(i)$nterstates, $(t)$oll roads, and $(f)$erries, respectively. A traveler may wish to find the shortest path between two locations in the network that avoids both toll roads and ferries. A Kleene language supporting this constraint could be defined as $L((\Sigma \setminus \{t, f\})^*)$.

The practical applications of KLCSP are also not restricted merely to modal constraints on a shortest path query. A label in $\Sigma$ can correspond to any arbitrary predicate condition associated with the edges of the graph. In later sections dealing with the graph index construction, we must extend the notion of edge labels to include support for multiple labels per edge. This also proves highly useful in scenarios where a given edge can support multiple such predicate conditions simultaneously.

In order to support this, we redefine the function $\ell$ to support multiple labels per edge as follows[2]: $\ell : E \to \mathcal{P}(\Sigma)$ is the labeling function mapping edges to a set of labels in $\Sigma$ (where $\mathcal{P}(\Sigma)$ denotes the *power set* of $\Sigma$). Since this new function can now map a given edge to multiple potential labels, we must also redefine what it means for a path $P_{s,t}$ to be valid for a given Kleene language constraint. For any restricted subset of labels $R \subseteq \Sigma$, we say that an *R-restricted path* is any path $P_{s,t} = \langle e_1, e_2, \ldots, e_q \rangle$, such that, for $1 \leq i \leq q$, $\ell(e_i) \cap R = \emptyset$ (i.e., the path avoids all restricted labels in $R$). We denote the least-cost, or "shortest", $R$-restricted path from $s \in V$ to $t \in V$ as $P^R_{s,t}$.

Unlike the algorithms for RLCSP and LRE, which require a search through a product graph, this simple subclass of regular languages allows for a much more efficient optimization of the constrained shortest path search. In particular, we need now only verify that a given edge's labels do not belong to the restricted subset of labels, as indicated by $R$, before relaxing the edge in the search. We present the pseudocode for solving the KLCSP problem using a straightforward adaptation of Dijkstra's algorithm in Algorithm 1. Note that a similar bidirectional search can also be performed instead of the unidirectional search presented in this pseudocode. We present the unidirectional variant here merely for simplicity and greater ease of understanding.

## 3.3 Contraction Hierarchies (CH)

CH [50, 53] have been proposed as an efficient graph indexing technique for supporting static point-to-point shortest path queries. The primary idea of CH is to establish some total order of the vertices in the graph (i.e., the ordering defines a bi-

[2]Note that supporting multiple labels per edge no longer fits properly into the original, formal language-constrained shortest paths model. To retrofit such scenarios back into this formal model, one can establish a new meta-alphabet $\Sigma'$ such that each distinct subset of labels (from $\Sigma$) supported by one or more edges in the graph is represented by a single, distinct meta-label in $\Sigma'$. Formal language constraints can then be properly re-defined according to the new alphabet $\Sigma'$.

**Algorithm 1** KLCSP-Dijkstra($G, s, t, R$)

**Input:** *Graph* $G = (V, E, w, \Sigma, \ell)$, $s, t \in V$, *restricted alphabet* $R \subseteq \Sigma$

**Output:** *Cost of shortest path* $P^R_{s,t}$

```
1:  $PQ \leftarrow \emptyset$
2:  for all $v \in V$ do
3:      $d[v] \leftarrow \infty$
4:  end for
5:  $d[s] \leftarrow 0$
6:  $PQ.Insert(s, d[s])$
7:  while $\neg PQ.Empty()$ do
8:      $u \leftarrow PQ.ExtractMin()$
9:      if $u = t$ then
10:         return $d[t]$
11:     end if
12:     for all $e = (u, v) \in E$ do
13:         if $\ell(e) \cap R = \emptyset \wedge d[u] + w(e) < d[v]$ then
14:             $d[v] \leftarrow d[u] + w(e)$
15:             if $v \notin PQ$ then
16:                 $PQ.Insert(v, d[v])$
17:             else
18:                 $PQ.DecreaseKey(v, d[v])$
19:             end if
20:         end if
21:     end for
22: end while
23: return $\infty$
```

jective function $\phi : V \rightarrow \{1, ..., |V|\}$) with respect to some notion of general, relative importance. Given such an ordering, preprocessing proceeds by "contracting" one vertex at a time, in increasing order of importance. When a vertex, $v$, is contracted, it is (temporarily) removed from the current graph "in such a way that shortest paths in the remaining...[sub]graph are preserved" [53]. In particular, for any pair of remaining vertices, $u$ and $w$, adjacent to $v$ in the original graph whose only shortest $u$-$w$ path is $\langle u, v, w\rangle$, a so-called *shortcut* edge $(u, w)$ must be added with the weight of the original shortest path cost through $v$ (see Figure 3.1 for an example). However, if there is an equivalent- or lesser-cost path from $u$ to $w$ other than $\langle u, v, w\rangle$, then no such shortcut edge is needed. Such a path is called a *witness* path. In order to detect witness paths, a local search from all nodes $u$, such that $(u, v) \in E$ and $\phi(u) > \phi(v)$, to all nodes $w$, such that $(v, w) \in E$ and $\phi(v) < \phi(w)$, is carried out to determine if a $(u, w)$ shortcut edge is necessary.

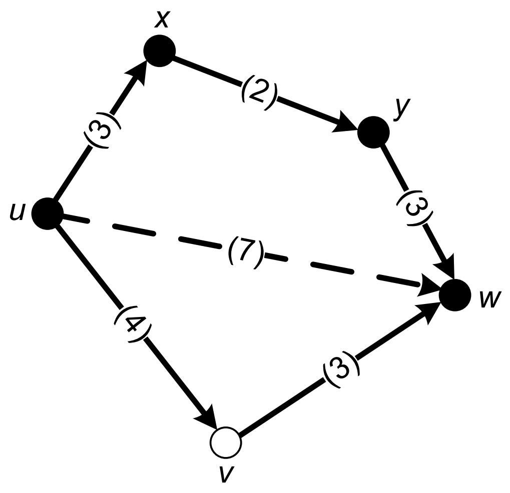


Figure 3.1: Contracting node $v$. Edges are labeled with their weights. The shortcut edge is represented with a dashed line.

Note that the number of shortcut edges added when contracting a graph is heavily dependent on the given ordering. Therefore, establishing a good ordering is one of the most crucial aspects of this methodology. In [53], Geisberger et al. establish several metrics to be associated with a given node that can help in determining the overall priority of that node in the ordering. In this context, vertex ordering is directly integrated into the contraction phase by first simulating the contraction of a given node to determine its resulting priority terms, and ordering the nodes in a priority queue based on a weighted linear combination of these terms. Some of these metrics include: the difference between the number of shortcut edges added and the number of adjacent edges removed when contracting a node (*edge difference*), the number of neighbors of a node that have already been contracted (*contracted neighbors*), and the number of original edges represented by any new shortcuts added when contracting a node (*original edges*). The interested reader is referred to [50, 53] for a more exhaustive list and greater details on each priority term considered. At each iteration, the node with minimum priority value is removed from the priority queue, contracted, and the priority values of all of its neighboring vertices are updated for the next iteration.

Once the set of shortcut edges, $E'$, has been established for a given ordering, shortest path queries may then be carried out using a bidirectional Dijkstra search variant which performs a simultaneous forward search in the *upward* graph $G_{\uparrow} = (V, E_{\uparrow})$, where $E_{\uparrow} = \{(v, w) \in E \cup E' \mid \phi(v) < \phi(w)\}$, and backward search[3] in the *downward* graph $G_{\downarrow} = (V, E_{\downarrow})$, where $E_{\downarrow} = \{(u, v) \in E \cup E' \mid \phi(u) > \phi(v)\}$. A tentative shortest path cost is maintained and is updated only when the two search frontiers meet to form a shorter path. The search in a given direction may be aborted once the minimum key

[3]Backward search in a graph $G = (V, E)$ is the equivalent of performing a standard (i.e., forward) search in the graph $\bar{G} = (V, \bar{E})$, where $\bar{E} = \{(v, u) \mid (u, v) \in E\}$.

for the priority queue in that direction exceeds the cost of the best tentative path seen so far. Once both search directions are finished, the best path seen thus far represents the shortest path cost.

As with any graph search algorithm, the efficiency of the search process is directly proportional to the number of nodes and edges explored during the search. The effectiveness of the CH search technique therefore comes from the use of the newly-added shortcut edges, which allow the Dijkstra search to effectively *bypass* irrelevant nodes during the search, without invalidating correctness, thus resulting in a greatly-reduced search space (and therefore, better runtime), as compared to the standard Dijkstra search on the original graph.

## 3.4 Contraction Hierarchies with Label Restrictions (CHLR)

Despite the naïve adaptation of Dijkstra's algorithm to support the Kleene language constrained shortest paths, as presented in Algorithm 1, this variation is still prohibitively slow on large graph datasets, as will be demonstrated later in our experimental results section. We therefore present the first enhancements to the hierarchical graph indexing concepts of Contraction Hierarchies to support KLCSP problems as follows. We start with a brief overview of the existing limitations of Contraction Hierarchies for solving this particular problem below.

### 3.4.1 Limitations of CH

In order to showcase the limitations of CH for Kleene language constrained shortest paths, let us consider a simple example graph with label alphabet $\Sigma = \{r, g, b\}$, representing the colors *red*, *green*, and *blue*, respectively. This example graph is illus-

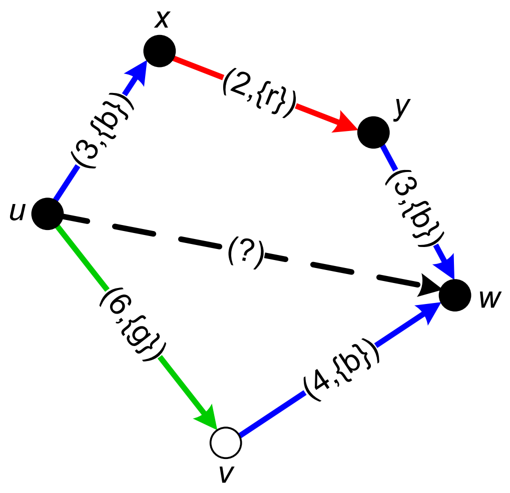


Figure 3.2: Contracting a labeled graph. Each edge, $e$, is labeled as $(w(e), \ell(e))$.

trated in Figure 3.2, where the edges have been colored according to their respective labels. In this scenario, when node $v$ is contracted, a local search will be performed to find a potential witness path from node $u$ to node $w$ in the graph induced by the set of nodes "higher" in the hierarchy than node $v$ (e.g., nodes $u$, $w$, $x$, and $y$). This local search will find a witness path, $\langle u, x, y, w \rangle$, with cost equal to 8, which happens to be less than the cost of the path $\langle u, v, w \rangle$, which is 10. In this case, no shortcut will be added between nodes $u$ and $w$ during the pre-processing. However, if we later wish to perform a Kleene language constrained shortest path query from $u$ to $w$, in which we restrict the color *red* from our shortest path (i.e., our language constraint is $L((\Sigma \setminus \{r\})^*)$), then the bidirectional search will be unable to find any such path between $u$ and $w$ (since there are no valid shortcuts between $u$ and $w$ and the edge $(x, y)$ will be invalid based on its *red* label), even though there exists a valid shortest path that avoids the color *red* in this graph: the path $\langle u, v, w \rangle$ with cost 10.

One naïve solution to this problem would be to establish a separate graph index for all possible subsets of the label alphabet $\Sigma$, and then use the appropriate index based on the incoming query constraints $R$. However, this is prohibitive, and would require the construction and maintenance of $2^{|\Sigma|}$ separate index datasets. Therefore, in the following sections, we propose methods to extend the concepts of Contraction Hierarchies to properly support any Kleene language constraints, and we prove the correctness of this approach, as well as providing experimental evidence in favor of this approach over other existing techniques (e.g., ALT).

### 3.4.2 CHLR Index Construction

The revised contraction algorithm for graph index construction (shown in Algorithm 2) works as follows. The algorithm processes each node $v \in V$ in the order defined by $\phi$ (which, for simplicity, we may assume is pre-defined). For each such node $v$, the algorithm considers all possible pairs of incoming edges $e_{\downarrow} = (u, v)$ and outgoing edges $e_{\uparrow} = (v, w)$, such that both $u$ and $w$ occur after $v$ in the ordering defined by $\phi$ (i.e., they occur "higher" in the hierarchy). For each such pair of edges, the algorithm performs a KLCSP-Dijkstra search in the subgraph defined by $G'_v$ (the subgraph of $G'$ induced by nodes with "higher" hierarchy than $v$), using the set of restricted labels, $R$, defined to be the set of labels "avoided" (or not supported) by both $e_{\downarrow}$ and $e_{\uparrow}$. If the KLCSP-Dijkstra search is able to find an equivalent- or lesser-cost path than the path $\langle u, v, w \rangle$, which also avoids the same set of restricted labels avoided by both $e_{\downarrow}$ and $e_{\uparrow}$, then no shortcut edge is necessary (since there can be no possible constraint scenario for which the path $\langle u, v, w \rangle$ is required). Edges are processed in order of increasing weight (see Lines 4 and 5) to ensure that the total number of shortcut edges constructed by this process is minimal for the given ordering $\phi$. See Section 3.4.5 for a formal proof of both correctness and minimality.

**Algorithm 2** KLCSP-Contraction$(G, \phi)$

**Input:** *Graph* $G = (V, E, w, \Sigma, \ell)$ *and bijective node order function* $\phi : V \to \{1, ..., |V|\}$

**Output:** *Augmented graph* $G' = (V, E \cup E', w, \Sigma, \ell)$, *where* $E'$ *represents newly-added shortcut edges*

```
 1: $G' \leftarrow G$
 2: $E' \leftarrow \emptyset$
 3: for all $v \in V$ ordered by $\phi$ do
 4:   for all $e_\downarrow = (u, v) \in E \cup E'$ ordered by $w(e_\downarrow) : \phi(u) > \phi(v)$ do
 5:     for all $e_\uparrow = (v, w) \in E \cup E'$ ordered by $w(e_\uparrow) : \phi(v) < \phi(w) \wedge w \neq u$ do
 6:       $G'_v \leftarrow G'[\{z \in V \mid \phi(v) < \phi(z)\}]$
 7:       $R \leftarrow \Sigma \setminus \{\ell(e_\downarrow) \cup \ell(e_\uparrow)\}$
 8:       $shortcutCost \leftarrow w(e_\downarrow) + w(e_\uparrow)$
 9:       $witnessCost \leftarrow$ KLCSP-Dijkstra$(G'_v, u, w, R)$
10:       if $shortcutCost < witnessCost$ then
11:         $e' \leftarrow (u, w)$
12:         $w(e') \leftarrow shortcutCost$
13:         $\ell(e') \leftarrow \{\ell(e_\downarrow) \cup \ell(e_\uparrow)\}$
14:         $E' \leftarrow E' \cup \{e'\}$
15:         $G' \leftarrow (V, E \cup E', w, \Sigma, \ell)$
16:       end if
17:     end for
18:   end for
19: end for
20: return $G'$
```

### 3.4.3 Multi-Edge Support

One important aspect of the enhancements to the graph contraction algorithm shown above is that our graph index must now support multi-edges (i.e., parallel edges) due to the potential for multiple possible paths between a given pair of nodes in the graph, depending upon the set of restricted labels chosen for the query. For example, in the graph illustrated in Figure 3.3, if the nodes are contracted in order from bottom to top, we must now insert two separate shortcut edges between nodes $u$ and $w$: edge $e$ is necessary when contracting node $v$ and edge $e'$ is necessary when contracting node $v'$. Note that, in this particular case, we cannot simply replace one shortcut edge with the other when added, since they might both be necessary for ensuring correctness of the resulting shortest paths, depending upon the set of restricted labels. In particular, if the restricted label set is $R = \{r, b\}$, then the shortest path between $u$ and $w$ will make use of the shortcut edge $e$, giving a cost of 10 and a final (expanded) path of $\langle u, v, w\rangle$. However, if the restricted label set is $R = \{r, g\}$, then the shortest path between $u$ and $w$ will make use of the shortcut edge $e'$, giving a cost of 12 and a final (expanded) path of $\langle u, v', w\rangle$.

### 3.4.4 CHLR Index Queries

Once the CHLR hierarchy has been established with the shortcut edge set, $E'$, shortest path queries for any given restricted label set, $R \subseteq \Sigma$, may then be carried out as follows. The search algorithm employed is (mostly) the same bidirectional Dijkstra search variant as is used for the static CH query algorithm (described in Section 3.3). However, we must now further augment the resulting upward and downward search graphs explored for a given query, respective of $R$. We redefine the upward search graph

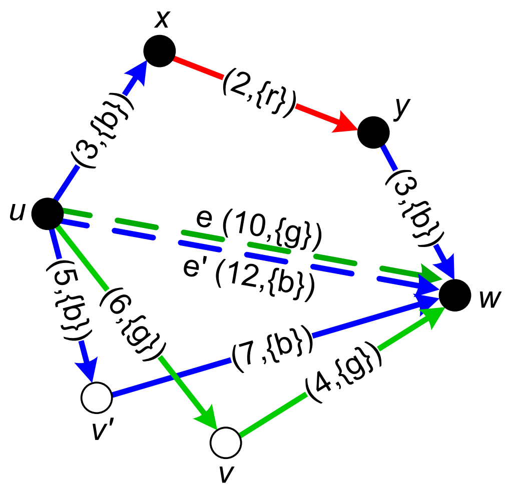


Figure 3.3: Multi-edge example

as $G_\uparrow = (V, E_\uparrow)$, where $E_\uparrow = \{e = (v, w) \in E \cup E' \mid \phi(v) < \phi(w) \wedge \ell(e) \cap R = \emptyset\}$, and the downward graph as $G_\downarrow = (V, E_\downarrow)$, where $E_\downarrow = \{e = (u, v) \in E \cup E' \mid \phi(u) > \phi(v) \wedge \ell(e) \cap R = \emptyset\}$. The CHLR query will now explore only those edges whose label sets are valid for the given query constraints.

### 3.4.5 Correctness and Minimality

**Lemma 1** *Let $P_{s,t}^{R'}$ define an $R'$-restricted shortest path from $s \in V$ to $t \in V$ for some $R' \subseteq \Sigma$. For any $R \subseteq R'$, $w(P_{s,t}^{R}) \leq w(P_{s,t}^{R'})$.*

**Proof.** Suppose there exists an $R$-restricted shortest path $P_{s,t}^{R}$ such that $w(P_{s,t}^{R}) > w(P_{s,t}^{R'})$. The path $P_{s,t}^{R'}$ is clearly a valid path for the restricted label set $R$ too, since $R \subseteq R'$ and, by definition, $P_{s,t}^{R'}$ must therefore avoid all restricted labels in $R$ as well. However, this contradicts the optimality of $P_{s,t}^{R}$. ■

**Theorem 2** *Given a graph $G' = (V, E \cup E', w, \Sigma, \ell)$ constructed by the KLCSP-Contraction algorithm, the query algorithm is **correct** for any $s \in V$, $t \in V$, and $R \subseteq \Sigma$.*

**Proof.** For consistency, we extend the original proof of correctness presented in [50] for static Contraction Hierarchies to support our new language constrained variant. For a given path $P_{s,t} = \langle s = v_1, \ldots, v_i, \ldots, v_q = t \rangle$, let $M_{P_{s,t}} = \{v_i \in P_{s,t} \mid 1 < i < q, \phi(v_{i-1}) > \phi(v_i) < \phi(v_{i+1})\}$ (i.e., the set of all local minima in $P_{s,t}$ with respect to $\phi$). We can classify all paths, $P_{s,t}$, in a given graph into one of two basic forms: (1) those with $M_{P_{s,t}} = \emptyset$ and (2) those with $M_{P_{s,t}} \neq \emptyset$.

Since the search algorithm only searches forward in the *upward* graph $G_\uparrow$ and *backward* in the downward graph $G_\downarrow$ (i.e., $\phi$ is strictly increasing in each search direction), then it will explore only paths of the form (1) during the search. For any origin node $s \in V$, destination node $t \in V$, and restricted label set $R \subseteq \Sigma$, suppose there exists a shortest path $P^R_{s,t}$ of the form (2) above in the original graph. We must now prove the claim that there must also exist an alternate (and equivalent) shortest path of the form (1) above after the KLCSP-Contraction algorithm has been run on the graph.

Since $M_{P^R_{s,t}} \neq \emptyset$, let $m(P^R_{s,t}) = min\{\phi(v) \mid v \in M_{P^R_{s,t}}\}$. Let $v_i$ be the node in path $P^R_{s,t}$ such that $\phi(v_i) = m(P^R_{s,t})$ (i.e., $v_i$ is the lowest order node in $M_{P^R_{s,t}}$). For edges $e_i = (v_{i-1}, v_i)$ and $e_{i+1} = (v_i, v_{i+1})$ in a shortest path $P^R_{s,t}$ of the form (2) above, let $R' = \Sigma \setminus \{\ell(e_i) \cup \ell(e_{i+1})\}$. We first demonstrate that $R \subseteq R'$.

Suppose for the sake of contradiction that $R \nsubseteq R'$. This implies that $\exists \alpha \in R$, such that either $\alpha \in \ell(e_i)$ or $\alpha \in \ell(e_{i+1})$. In either case, the subpath $\langle e_i, e_{i+1} \rangle$ would then be invalid for any $R$-restricted shortest path, contradicting the validity of $P^R_{s,t}$. Therefore, in this context, $R \subseteq R'$ must be true.

Next, let us consider the hypothetical scenario where we perform a call to the KLCSP-Dijkstra search algorithm to find an $R$-restricted shortest path $P^R_{v_{i-1},v_{i+1}}$ in the subgraph $G'_{v_i} = G'[\{z \in V \mid \phi(v_i) < \phi(z)\}]$. If $w(P^R_{v_{i-1},v_{i+1}}) < w(e_i) + w(e_{i+1})$, then there exists a *shorter* $R$-restricted path between $v_{i-1}$ and $v_{i+1}$ in $G'_{v_i}$ (that does not include $e_i$ or $e_{i+1}$), contradicting the optimality of $P^R_{s,t}$. Therefore, $w(P^R_{v_{i-1},v_{i+1}}) \geq w(e_i) + w(e_{i+1})$ must hold true. Given that $R \subseteq R'$, then by Lemma 1, we know that $w(P^{R'}_{v_{i-1},v_{i+1}}) \geq w(P^R_{v_{i-1},v_{i+1}})$ must also hold true. This gives us $w(P^{R'}_{v_{i-1},v_{i+1}}) \geq w(e_i) + w(e_{i+1})$. Note that $R'$ is exactly equal to the restricted label set used in the search for a restricted witness path during the graph index construction of the KLCSP-Contraction algorithm when processing node $v_i$, where $e_{\downarrow} = e_i$ and $e_{\uparrow} = e_{i+1}$. In the case where $w(P^{R'}_{v_{i-1},v_{i+1}}) > w(e_i) + w(e_{i+1})$, then the KLCSP-Contraction algorithm will have added a shortcut edge from $v_{i-1}$ to $v_{i+1}$ with weight $w(e_i) + w(e_{i+1})$. In the case where $w(P^{R'}_{v_{i-1},v_{i+1}}) = w(e_i) + w(e_{i+1})$, then this means that there already exists an alternate and equivalent-cost path in the subgraph $G'_{v_i}$, defined above. Either way, we can construct a new path $\bar{P}^R_{s,t}$ which bypasses $v_i$ altogether (using either the shortcut or the path between $v_{i-1}$ and $v_{i+1}$ in $G'_{v_i}$; since $R \subseteq R'$, either is valid for $R$), such that $w(\bar{P}^R_{s,t}) = w(P^R_{s,t})$. If $\bar{P}^R_{s,t}$ is of the form (1), then the proof is complete. If $\bar{P}^R_{s,t}$ is itself of the form (2), then, since $v_i \notin \bar{P}^R_{s,t}$ and $\phi(v_i) = m(P^R_{s,t})$, we know that $m(\bar{P}^R_{s,t}) > m(P^R_{s,t})$, and we can apply the same argument (as above) recursively to $\bar{P}^R_{s,t}$. Since there are only a finite number of possible levels in $\phi$ (i.e., the function $m$ cannot increase indefinitely), then this recursive argument must eventually produce an alternate path $\bar{P}^R_{s,t}$, such that $M_{\bar{P}^R_{s,t}} = \emptyset$.

Therefore, for any shortest path $P^R_{s,t}$ of the form (2) above, there also exists an alternate and equivalent shortest path of the form (1) above. Since the query algorithm performs a shortest path search amongst all and only the paths of the form (1), then the query algorithm is correct for any $s \in V$, $t \in V$, and $R \subseteq \Sigma$. ■

Another property that we wish to discuss in this work, which has not been previously addressed even for static Contraction Hierarchies, is that of edge minimality for a given ordering $\phi$. One might be easily tempted to believe that, when processing edges $e_{\downarrow} = (u, v)$ and $e_{\uparrow} = (v, w)$ in arbitrary order, where $R = \Sigma \setminus \{\ell(e_{\downarrow}) \cup \ell(e_{\uparrow})\}$, if $P^{R}_{u,w} \nsubseteq G'_v$ then a shortcut edge $(u, w)$ is absolutely necessary for correctness. However, this is not always the case. Consider the example graph in Figure 3.4. If we start the contraction of $v$ by first processing edges $e_{\downarrow} = (u, v)$ and $e_{\uparrow} = (v, w)$, then clearly $P^{R}_{u,w} \nsubseteq G'_v$ (since $G'_v$ contains only the edges $(u, x)$ and $(y, w)$). Regardless, it turns out that there is still no need to add a $(u, w)$ shortcut edge for this scenario (since $P^{R}_{u,w} \neq \langle u, v, w \rangle$). To demonstrate why, consider what happens if we had first processed the edges $e_{\downarrow} = (x, v)$ and $e_{\uparrow} = (v, y)$. In this case, we would have added a shortcut edge $(x, y)$ with a weight of 3. If we then process edges $e_{\downarrow} = (u, v)$ and $e_{\uparrow} = (v, w)$, $P^{R}_{u,w} \subseteq G'_v$ will be true, in which case, no shortcut is necessary. In fact, in the extreme case for this degenerate example, if we process the pairs of edges in the order $\langle e_{\downarrow} = (u, v), e_{\uparrow} = (v, w) \rangle$, then $\langle e_{\downarrow} = (u, v), e_{\uparrow} = (v, y) \rangle$, then $\langle e_{\downarrow} = (x, v), e_{\uparrow} = (v, w) \rangle$, and finally $\langle e_{\downarrow} = (x, v), e_{\uparrow} = (v, y) \rangle$, this will result in the addition of 4 separate shortcut edges (one for each pair). However, if we process these same pairs of edges in the reverse order of that above, we will have added only 1 shortcut edge $(x, y)$. Also note that we cannot simply remove edges $(u, v)$ or $(v, w)$ from this graph, since they may be necessary depending on the incoming label constraint (e.g., $P^{\{b\}}_{u,v} = (u, v)$).

Therefore, even in the case where $P^{R}_{u,w} \neq \langle u, v, w \rangle$, if we are not careful to process the adjacent edges in the correct order, we may be unable to find a valid path $P^{R}_{u,w} \subseteq G'_v$, resulting in unnecessary shortcut edges. In particular, we need a way to ensure that, when processing edges $e_{\downarrow} = (u, v)$ and $e_{\uparrow} = (v, w)$, either $P^{R}_{u,w} = \langle u, v, w \rangle$ or $P^{R}_{u,w} \subseteq G'_v$ always holds true. The following lemma suggests that this property will be met if we process each contracted node's adjacent edges in order of increasing weight (as shown in Algorithm 2).

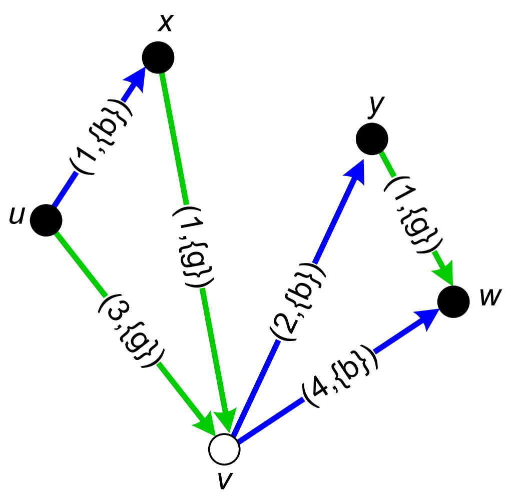


Figure 3.4: Counter-example showing lack of minimality when edges are considered in arbitrary order.

**Lemma 3** *Let $e_\downarrow = (u, v)$ and $e_\uparrow = (v, w)$ be the pair of edges currently being processed by the KLCSP-Contraction algorithm during contraction of node $v \in G'$. Either $P^R_{u,w} = \langle u, v, w\rangle$ or $P^R_{u,w} \subseteq G'_v$ must hold true.*

**Proof.** Suppose for the sake of contradiction that $P^R_{u,w} \neq \langle u, v, w\rangle$ and $P^R_{u,w} \nsubseteq G'_v$. Note that $P^R_{u,w} \nsubseteq G'_v$ implies that $v \in P^R_{u,w}$, while $P^R_{u,w} \neq \langle u, v, w\rangle$ implies that $(u, v) \notin P^R_{u,w}$, or $(v, w) \notin P^R_{u,w}$, or both.

Consider the case where $e = (v, w) \notin P^R_{u,w}$. Since we know that $v \in P^R_{u,w}$, then $P^R_{u,w} = \langle u, \ldots, v, y, \ldots, w\rangle$ such that $e' = (v, y) \in E \cup E'$ (i.e., if $(v, w) \notin P^R_{u,w}$ and $v \in P^R_{u,w}$, then $v$ must reach node $w$ through some other edge $e' = (v, y)$). This means that $w(e') < w(e)$, otherwise we could construct a lesser-cost path $P^R_{u,w}$ that actually includes $(v, w)$. However, since we process all outgoing edges $e_\uparrow$ in order of increasing weight in the construction algorithm, then $w(e') < w(e)$ implies that we must have already processed the pair $\langle e_\downarrow = (u, v), e_\uparrow = (v, y)\rangle$. By definition, this means that $P^R_{u,y} \subseteq G'_v$, so, using this subpath, we can construct a path from $u$ to $w$ that avoids $v$ such that $P^R_{u,w} \subseteq G'_v$, leading to a contradiction. A symmetric argument holds for the case where $e = (u, v) \notin P^R_{u,w}$, relative to the fact that we process all incoming edges in order of increasing weight as well. ■

**Theorem 4** *Given a fixed node ordering function, $\phi$, the edge set $E'$ constructed by the KLCSP-Contraction algorithm is **minimal** (i.e., there is no algorithm which can produce a smaller set of shortcut edges, while still guaranteeing correctness).*

**Proof.** Suppose there exists some algorithm which can construct a set of shortcut edges, $E''$, from the ordering $\phi$, such that $|E''| < |E'|$, and the set $E''$ is correct for any possible restricted label set $R \subseteq \Sigma$. This means there must exist some edge $e = (u, w)$, such that $e \in E'$ and $e \notin E''$. Since $e \in E'$, then by definition, there must also exist some node $v$, such that $\phi(u) > \phi(v) < \phi(w)$, $e_\downarrow = (u, v), e_\uparrow = (v, w) \in E \cup E'$, and $w(e) = w(e_\downarrow) + w(e_\uparrow)$. Let $R = \Sigma \setminus \{\ell(e_\downarrow) \cup \ell(e_\uparrow)\}$. By Lemma 3, we know that, when processing $e_\downarrow$ and $e_\uparrow$ to contract node $v$, either $P^R_{u,w} = \langle u, v, w\rangle$ or $P^R_{u,w} \subseteq G'_v$ must hold true. If $P^R_{u,w} \subseteq G'_v$, then, by definition, if such a path exists, the index construction algorithm would not have added a shortcut from $u$ to $w$, contradicting the fact that $e \in E'$. However, if $P^R_{u,w} = \langle u, v, w\rangle$, then $E''$ is incorrect for the query to find $P^R_{u,w}$, since $e \notin E''$. Either case leads to a contradiction. ■

### 3.4.6 Optimizations

As indicated in the KLCSP-Contraction index construction algorithm, during the contraction of a given node $v$, where $I^\downarrow_v = \{(u, v) \in E \cup E' \mid \phi(u) > \phi(v)\}$ and $O^\uparrow_v = \{(v, w) \in E \cup E' \mid \phi(v) < \phi(w)\}$, the algorithm performs a total of $|I^\downarrow_v| \cdot |O^\uparrow_v|$ calls to KLCSP-Dijkstra[4]. While correct and minimal (for a given ordering $\phi$), the overall efficiency of the contraction of $v$ can be improved by instead performing only

[4]Pairs $\langle e_\downarrow = (u, v), e_\uparrow = (v, w)\rangle$ where $u = w$ are ignored.

a single local search from the source, $u$, of each incoming edge $e_{\downarrow} = (u, v) \in I_v^{\downarrow}$ until all nodes in the set $W = \{w \in V \mid (v, w) \in O_v^{\uparrow}\}$ have been settled, or until a distance of $w(e_{\downarrow}) + max\{w(e_{\uparrow}) \mid e_{\uparrow} = (v, w) \in O_v^{\uparrow}, w \neq u\}$ has been reached (this is similar to the approach used in [53]). Using this approach we can set $R \leftarrow \Sigma \setminus \ell(e_{\downarrow})$ and pass this restricted label set to the augmented version of KLCSP-Dijkstra. Note that this does not affect the correctness of the resulting index, since the set $R$ that we pass to KLCSP-Dijkstra in this case is a superset of the restricted label set passed to the KLCSP-Dijkstra calls in the original algorithm, for all possible pairs of incoming and outgoing edges. This means that any resulting witness paths are still valid (i.e., they are more constrained than normal) and this approach can only result in a superset of (potentially superfluous) shortcuts to that of the original approach. Therefore, by taking this approach, we lose the property of minimality. However, initial experiments indicate that this approach scales much better in practice.

A more complex bidirectional version of this technique is used in [53] in which they first perform a single-hop backward search from all nodes $w \in W$ to their immediate neighbors in $X = \{x \in V \mid (x, w) \in E \cup E', w \in W, x \neq v\}$, and then perform the forward search from $u$ to the target set $X$ (instead of $W$). This allows the distance bound of the forward search to be further reduced to $w(e_{\downarrow}) + max\{w(e_{\uparrow}) - min\{w(e) \mid e = (x, w) \in E \cup E'\} \mid e_{\uparrow} = (v, w) \in O_v^{\uparrow}, w \neq u\}$. We further adapt this technique to our own language constrained variant by performing the restricted forward search from $u$ as indicated in the paragraph above and by relaxing only edges $e = (x, w)$ for each node $w \in W$ in the single-hop backward search if $\ell(e) \subseteq \{\ell(e_{\downarrow}) \cup \ell(e_{\uparrow})\}$, where $e_{\uparrow} = (v, w)$, thus still preserving correctness.

Additionally, we employ the technique of using *hop limits* [53], in which we specify a limit on the number of hops that the paths on our local search can take. Each local search is aborted once the number of hops on the paths found by the search exceeds the specified constant limit. This can greatly speed up the local search times, but, like our other optimization, may result in unnecessary shortcuts being added during contraction. We use this optimized version of our algorithm for all subsequent experimental results presented in this chapter.

## 3.5 Experiments

### 3.5.1 Test Environment

All experiments were carried out on a 64-bit server machine running Linux CentOS 5.3 with 2 quad-core CPUs clocked at 2.53 GHz with 72 GB RAM (although only one core was used per experiment). Our implementation of the CHLR technique is an extended implementation of the original Contraction Hierarchies source code, written in C++, and further detailed in [50]. Our implementation of the ALT algorithm (used for comparison against CHLR) is based on the algorithm described in [36]. All programs were compiled using gcc version 4.1.2 with optimization level 3.

### 3.5.2 Test Dataset

For our experiments, we used the continent-wide graph dataset of North America (this includes only the US and Canada), represented by a total of $21,133,774$ nodes and $52,523,592$ edges. $6,779,795$ of these edges support one or more labels in this dataset, with 0.21 labels per edge, on average. Table 3.1 offers some additional information on the 16 different real-world labels supported in the North American graph dataset. This dataset (including labels) was derived from NAVTEQ transportation data products, under their permission.

Table 3.1: Graph Label Support for North America

| **Label** | **# Edges** |
|---|---|
| Ferry | 2,610 |
| Toll Road | 47,388 |
| Unpaved Road | 3,645,458 |
| Private Road | 1,662,314 |
| Limited Access Road | 682,396 |
| 4-Wheel-Drive-Only Road | 139,284 |
| Parking Lot Road | 160,850 |
| Hazmat Prohibited | 45,950 |
| All Vehicles Prohibited | 64,414 |
| Delivery Vehicles Prohibited | 148,010 |
| Trucks Prohibited | 475,472 |
| Taxis Prohibited | 147,628 |
| Buses Prohibited | 151,272 |
| Automobiles Prohibited | 114,192 |
| Pedestrians Prohibited | 1,253,030 |
| Through Traffic Prohibited | 2,050,562 |

### 3.5.3 Node Ordering

Our initial experiments were focused on determining a good approach for node ordering in the context of Kleene language constrained shortest paths. For this experiment, we took an approach similar to that of [50], in which we considered several different ordering metrics, along with several different combinations of weighted coefficients for each metric tested. In particular, we considered 6 unique ordering metrics (the first 5 of which come from [50]): *edge difference*, *contracted neighbors*, *original edges*, *search space depth*, *local search space size*, and a new priority term introduced here, which represents the number of new multi-edges introduced during the contraction of a node (*new multi-edges*).

For each metric, we defined a range of possible values for their associated weight coefficient (e.g, $0 - 300$), as well as an incremental step size (e.g., 100). We then carried out experiments on all possible combinations of coefficients for these metrics, using the specified ranges and step sizes. In all, we tested $4,096$ (i.e., $4^6$) combinations of the 6 different ordering metrics on a subgraph of the North American graph, representing the state of Virginia (with $483,504$ nodes and $1,113,602$ edges). For each configuration of coefficient values for these 6 metrics, the graph index was constructed using that particular configuration, and then a series of $10,000$ uniform random shortest path queries were run on the index (the same random pairs were used for each configuration for consistency). For each pair of nodes in the set of random test cases, we ran both a non-restricted search (i.e., no labels were restricted; $R = \emptyset$) and a fully-restricted search (i.e., all labels were restricted; $R = \Sigma$)[5].

From these results, we calculated the product of the construction time of the index and the average overall query time (considering both the unrestricted and restricted results together), and then chose the configuration with the smallest such product value. The smallest of these products can be seen as a good compromise of construction time and query time. From these experiments, we found that a combination of only 2 particular ordering metrics was sufficient to produce the best overall results for the graphs tested here. In particular, for all subsequent experiments carried out here, we have chosen to use only the *edge difference* metric, with a weighted coefficient of 100, and the *original edges* metric, with a weighted coefficient of 200.

### 3.5.4 Comparative Results

In Table 3.2, we present the results of this approach when applied to the full North American graph. This table compares both the preprocessing and query results of CHLR against the bidirectional adaptation of Dijkstra's algorithm, as well as the ALT algorithm, constructed using 64 landmarks (ALT-64). As with the original node ordering

[5]This is feasible since not all edges support labels in our test datasets.

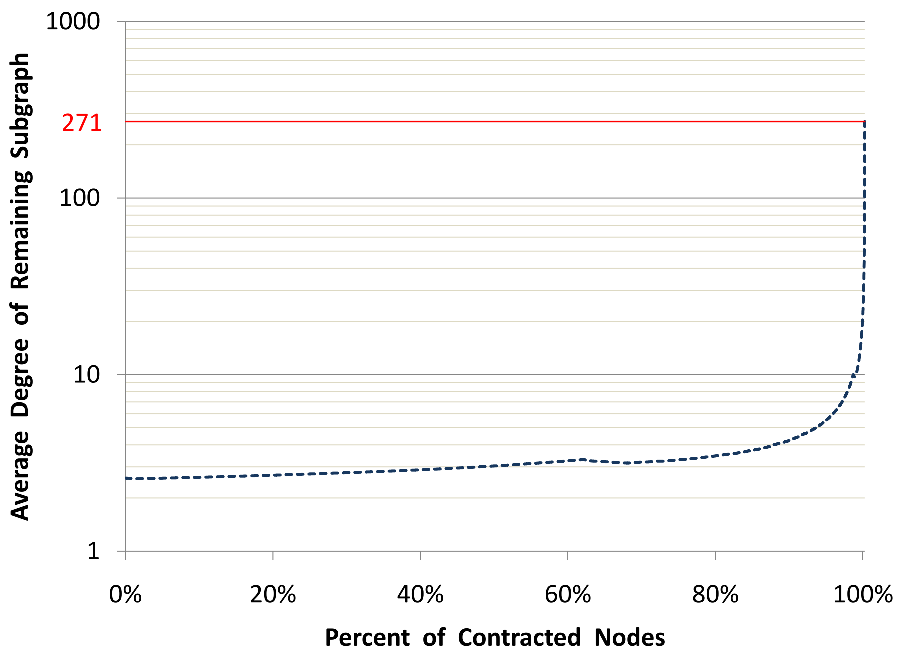
experiments, for the queries, we take the averages of 10,000 random unrestricted queries (where $R = \emptyset$) and 10,000 random restricted queries (where $R = \Sigma$). Even though the CHLR technique requires nearly 3 times the preprocessing time than that of ALT-64 for the North American graph, we are able to achieve 3 orders of magnitude improvements in both search space and query times over both the Dijkstra algorithm and ALT-64, on average (this is due primarily to the effectiveness of the shortcut edges in CHLR, which greatly reduce the resulting search space, and thus, the query times). However, as we will see in later experiments, the overall performance of these techniques can strongly depend on the chosen set of restricted labels.

Table 3.2: Experiments on the North American Graph Dataset

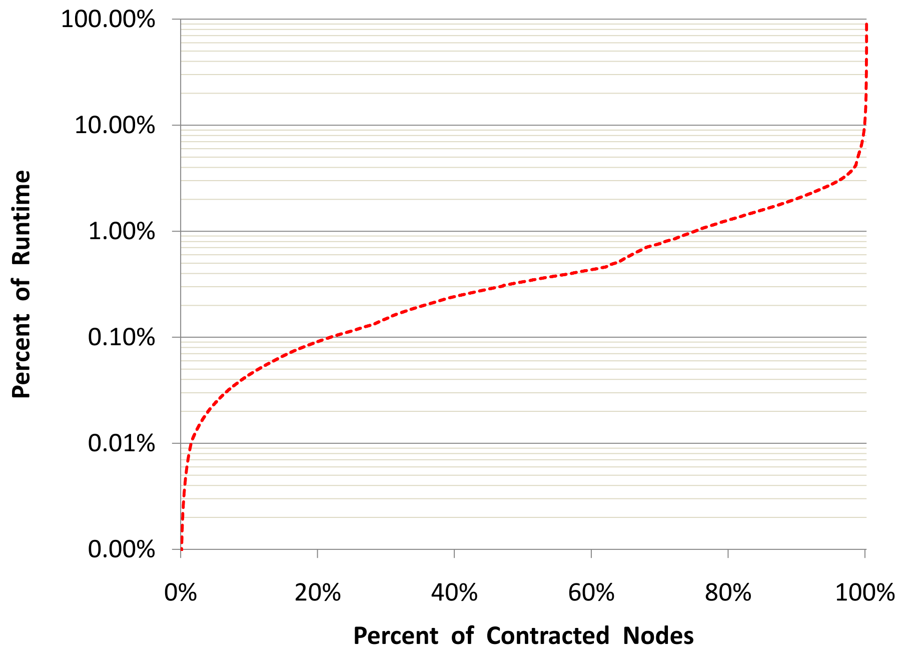
| | **Preprocessing** | | **Queries** | |
|---|---|---|---|---|
| | Time | Space | # Settled | Time |
| **Technique** | [H:M] | [B/node] | Nodes | [ms] |
| Bidir. Dijkstra | 0:00 | 0 | 6,799,486 | 3,043.89 |
| ALT-64 | 0:49 | 512 | 1,141,430 | 1,528.80 |
| CHLR | 2:10 | 62 | 993 | 2.18 |

## 3.5.5 Degree Limits

An adverse side effect to the fact that this new approach must now support multi-edges is that of degree explosion during the graph index construction. As indicated in Figure 3.5(a), as the index construction proceeds for the North American graph, the average degree of the remaining subgraph quickly grows from 10 to 271 during contraction of the last 2% of the nodes. This degree explosion can also be seen to have a strong impact on the overall runtime of the index construction algorithm in practice, where, for the North American graph, roughly 90% of the runtime was spent contracting only the last 1% of the nodes (see Figure 3.5(b)).



(a) Average Degree Progression



(b) Runtime Percentage

Figure 3.5: Effects of Degree Explosion During Construction of the North American Graph Dataset

In order to combat this effect, we introduce the concept of a *degree limit* within the construction algorithm, in which contraction of the remaining nodes is aborted as soon as the average degree of the remaining nodes reaches some critical threshold, as defined by the limit. The remaining (uncontracted) nodes in the graph make up what are called the *core* nodes of the graph index (a concept introduced and explored in [69] and also in [50] for many-to-many shortest path searches and in [23] for goal-directed routing). Once the contraction is aborted after reaching the degree limit, then, for all remaining nodes, $v$, in the core, we set $\phi(v) = |V|$ [6]. To maintain correctness of results, we then need only adjust our search graphs as follows. We set $G_{\uparrow} = (V, E_{\uparrow})$, where $E_{\uparrow} = \{e = (v,w) \in E \cup E' \mid \phi(v) \leq \phi(w) \land \ell(e) \cap R = \emptyset\}$ and $G_{\downarrow} = (V, E_{\downarrow})$, where $E_{\downarrow} = \{e = (u,v) \in E \cup E' \mid \phi(u) \geq \phi(v) \land \ell(e) \cap R = \emptyset\}$ (i.e., we no longer maintain a *strict* total node ordering like we did before). Using these search graphs, the algorithm still maintains correctness; however, searching in the core now becomes more exhaustive due to the relaxed filtering.

Table 3.3 shows the results of our experiments over several different degree limits on the North American graph. As can be seen, the index construction time can be greatly reduced by using reasonable degree limits, without sacrificing too much of the overall speed of any subsequent queries on the index. Even for the smallest degree limit of 10, with the worst query times, we are still able to outperform the ALT-64 results from Table 3.2 by an order of magnitude (on average), requiring only 6 minutes of preprocessing time.

[6]The function $\phi$ is no longer a bijective function in this context.

Table 3.3: Degree Limit Experiments on the North American Graph Dataset

| **Degree Limit** | **Preprocessing** Time [H:M] | Space [B/node] | **Queries** # Settled Nodes | Time [ms] | **Core Size** |
|---|---|---|---|---|---|
| 10 | 0:06 | 60 | 238,513 | 130.14 | 252,719 |
| 20 | 0:13 | 61 | 59,244 | 40.44 | 64,153 |
| 30 | 0:18 | 62 | 28,732 | 25.32 | 30,863 |
| 40 | 0:23 | 62 | 16,807 | 17.12 | 17,677 |
| 50 | 0:29 | 62 | 11,212 | 11.63 | 11,541 |
| 100 | 0:57 | 62 | 3,577 | 4.96 | 3,184 |
| 200 | 1:43 | 62 | 1,236 | 2.63 | 498 |

### 3.5.6 Restriction Cardinality

Here we present experiments on the overall effects of the number of restricted labels chosen for a given KLCSP query. For this set of experiments, we compare the CHLR technique against the ALT-64 technique. Since the North American graph supports only 16 different labels, we perform 17 sets of experiments, one for each possible size of the restricted label set, $|R| = 0, \ldots, 16$. For each of the 17 possible cardinalities of $R$, we perform a set of $10,000$ uniform random shortest path queries. For each of the random pairs of vertices in the test set for a given cardinality, $i$, we choose a random restricted label set $R \subseteq \Sigma$, such that $|R| = i$. The results of this experiment are presented as a box-and-whisker plot in Figure 3.6.

An interesting property emerges from our proposed CHLR technique, as compared to ALT-64 in these experiments. In particular, we can see that, the more restricted the shortest path query is, the better the CHLR technique performs, in general. Alternatively, the performance of ALT-64 actually becomes much worse as the query becomes more restricted (by up to an order of magnitude). The improvements in performance of the CHLR technique as the queries become more restricted can be attributed to the fact that more of the shortcut edges are also now likely to be restricted, thus pruning the search space even more than in the relatively unrestricted cases. The degradation of performance for ALT-64 is primarily due to the fact that the potential functions computed during preprocessing become much weaker in general as the dynamic constraints on the graph continue to change, as indicated in [36].

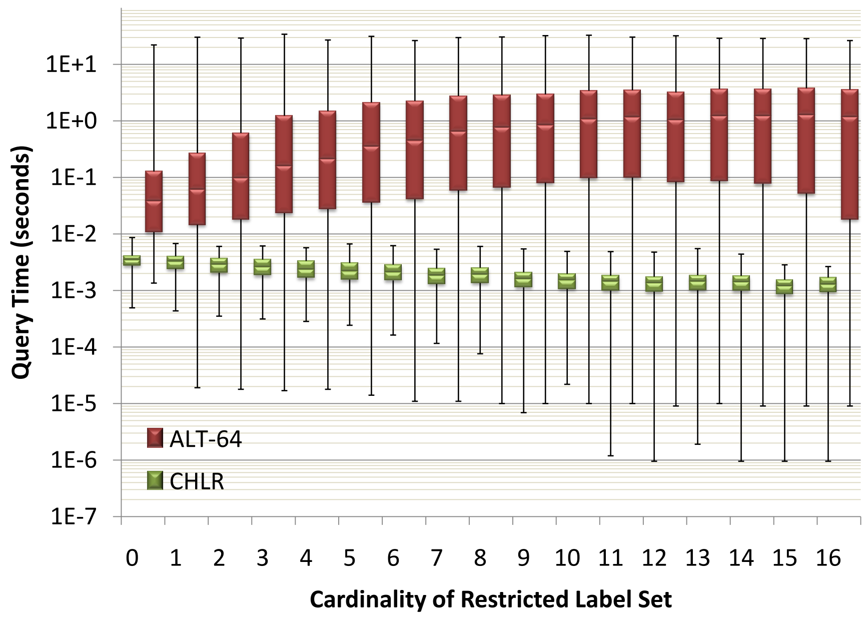


Figure 3.6: Experiments on Restriction Cardinality

## 3.6 Conclusion

We have presented and formalized a new shortest path query type as a variant of language constrained shortest path problems. We have also successfully extended the graph indexing technique known as Contraction Hierarchies to efficiently support this new dynamically constrained query type. Experimental results on real-world graph data indicate that this new technique is several orders of magnitude better than Dijkstra's algorithm and the ALT algorithm, both in terms of query time and search space. Additionally, the performance of this technique also seems to improve under more heavily constrained query scenarios, making it a perfect candidate for supporting this new query type. While this technique has proven highly applicable on real-world road network data, further exploration of the overall robustness of our technique on different synthetically labeled graph configurations might allow to better determine the properties of graph labeling which can affect the relative performance and scalability of our proposed technique. Additional work in this area also includes extending the concepts of this research to more complex edge restriction types, such as height and weight restrictions for road networks. We further explore such complex edge restrictions in the following chapter.

We conclude this chapter by presenting an alternative perspective and algorithmic approach for guaranteeing edge minimality in the resulting graph index, as originally discussed in Section 3.4.5.

### 3.6.1 Alternative Index Construction

Another way of looking at the previous problem suggested by Figure 3.4 is that, by always omitting the junction $v$ from the induced subgraph $G'_v$, the local search will never be able to find witness paths of the form $P^R_{u,w} = \langle u, \ldots, x, v, y, \ldots, w \rangle$ such that $u \neq x$ and/or $y \neq w$, resulting in the possibility of adding $(u, w)$ shortcuts unnecesssarily. One alternative solution to that previously suggested is not to omit $v$ from $G'_v$, but rather, to include $v$, and instead model a "turn restriction" in the local search, in which we do not allow the local search to perform a transition from the incoming edge $(u, v)$ to the outgoing edge $(v, w)$. This will ensure that we find the best possible path from $u$ to $w$ other than the path $\langle u, v, w \rangle$, including any valid witness paths of the above form. If this alternate path cost is less than or equivalent to the cost of path $\langle u, v, w \rangle$, then no shortcut is needed, and both minimality and correctness remain preserved.

Using this alternative approach, the order of the local searches relative to $v$

then becomes irrelevant, making this methodology more efficient in practice (since we no longer have to rely on sorting adjacent edges). However, this requires a more complex redefinition of the local search algorithm.

Here we present the pseudocode for these alternative local search and index construction algorithms, as well as a brief discussion of correctness and minimality. For the purposes of this discussion, we shall call the new local search and index construction algorithms KLCSP-Dijkstra-Alt and KLCSP-Contraction-Alt, respectively.

We start with the revised local search algorithm (KLCSP-Dijkstra-Alt; see Algorithm 3). This local search algorithm behaves almost exactly the same as before, except we now keep track of the parent node for each node in the current shortest path tree. We store this information in the newly added $p$ array. This information is used to ensure that we do not allow the local search to make a transition from the incoming edge $(s, r)$ to the outgoing edge $(r, t)$, as defined by the input parameter constraints. This "turn restriction" is enforced in Line 13 when deciding which edges to relax from the current node during the search. Here, the search will skip the relaxation of the edge $(u, v)$ if $(p[u] = s) \wedge (u = r) \wedge (v = t)$ is true, which indicates the restricted transition from $(s, r)$ to $(r, t)$. We note, however, that this constraint alone is not sufficient to fully guarantee that we always find the best alternate path from $s$ to $t$, other than $\langle s, r, t \rangle$.

For example, consider the graph presented in Figure 3.4. Assume that we have reduced the weight of edge $(u, v)$ in this graph to 2. If we call KLCSP-Dijkstra-Alt to find the shortest path from $u$ to $y$ other than $\langle u, v, y \rangle$, then node $v$ will be relaxed from parent edge $(u, v)$ first during the search (giving $p[v] = u$ and $d[v] = 2$). By the time we relax edge $e = (x, v)$ in the local search, we will not be able to improve the value $d[v]$ at Line 15 (since $d[x] + w(e) = d[v]$). If we leave $p[v] = u$, then we will be unable to find *any* valid path from $u$ to $y$, since we will ultimately restrict the relaxation of edge $(v, y)$

due to the transition from $(u, v)$. However, there is still an equivalent cost shortest path from $u$ to $y$ other than $\langle u, v, y\rangle$: $\langle u, x, v, y\rangle$.

To ensure that we are able to find such alternate, equivalent paths, we must include the additional condition at Line 23 to always "prefer" equivalent-cost paths from node $s$ to node $r$ other than the incoming edge $(s, r)$ in the local search, thus eliminating this problem. Note that we do not have to add similar constraints for preferring equivalent-cost paths from $r$ to $t$ other than $(r, t)$, since the edge $(r, t)$ will only be relaxed if $p[r] \neq s$.

The index construction algorithm (KLCSP-Contraction-Alt; see Algorithm 4) is then changed to omit the ordering of the edges relative to $v$ by weight (since this is no longer necessary, as we will show below), as well as to include $v$ in the induced subgraph $G'_v$, and, finally, to call the new local search algorithm.

For the remainder of the discussion, we must first clarify some potentially troublesome notation. We note that, in the context of the original KLCSP-Contraction algorithm pseudocode, the induced subgraph $G'_v$ is defined such that $v \notin G'_v$ when contracting node $v$. However, in the context of the new KLCSP-Contraction-Alt algorithm pseudocode, we have that $v \in G'_v$. We shall refer here only to this latter subgraph definition of $G'_v$.

In Lemma 3, we showed that, by processing adjacent edges in order of increasing weight when contracting node $v$, we can guarantee that, when processing edges $e_\downarrow = (u, v)$ and $e_\uparrow = (v, w)$, either $P^R_{u,w} = \langle u, v, w\rangle$ or $P^R_{u,w} \subseteq G'_v \setminus \{v\}$ must (already) be true. Here, we prove a slightly different claim for the KLCSP-Contraction-Alt algorithm.

**Lemma 5** *Let $e_\downarrow = (u, v)$ and $e_\uparrow = (v, w)$ be the pair of edges currently being processed by the KLCSP-Contraction-Alt algorithm during contraction of node $v \in G'$. Either*

**Algorithm 3** KLCSP-Dijkstra-Alt$(G, s, r, t, R)$

**Input:** *Graph* $G = (V, E, w, \Sigma, \ell)$, $s, r, t \in V$, *restricted alphabet* $R \subseteq \Sigma$

**Output:** *Cost of shortest path* $P^R_{s,t}$, such that $P^R_{s,t} \neq \langle s, r, t \rangle$

```
 1: $PQ \leftarrow \emptyset$
 2: for all $v \in V$ do
 3:     $d[v] \leftarrow \infty$
 4:     $p[v] \leftarrow null$
 5: end for
 6: $d[s] \leftarrow 0$
 7: $PQ.Insert(s, d[s])$
 8: while $\neg PQ.Empty()$ do
 9:     $u \leftarrow PQ.ExtractMin()$
10:     if $u = t$ then
11:         return $d[t]$
12:     end if
13:     for all $e = (u, v) \in E : (p[u] \neq s) \vee (u \neq r) \vee (v \neq t)$ do
14:         if $\ell(e) \cap R = \emptyset$ then
15:             if $d[u] + w(e) < d[v]$ then
16:                 $d[v] \leftarrow d[u] + w(e)$
17:                 $p[v] \leftarrow u$
18:                 if $v \notin PQ$ then
19:                     $PQ.Insert(v, d[v])$
20:                 else
21:                     $PQ.DecreaseKey(v, d[v])$
22:                 end if
23:             else if $d[u] + w(e) = d[v] \wedge u \neq s \wedge v = r$ then
24:                 $p[v] \leftarrow u$
25:             end if
26:         end if
27:     end for
28: end while
29: return $\infty$
```

**Algorithm 4** KLCSP-Contraction-Alt($G, \phi$)

**Input:** *Graph* $G = (V, E, w, \Sigma, \ell)$ *and bijective node order function* $\phi : V \to \{1, ..., |V|\}$

**Output:** *Augmented graph* $G' = (V, E \cup E', w, \Sigma, \ell)$, *where* $E'$ *represents newly-added shortcut edges*

```
 1: $G' \leftarrow G$
 2: $E' \leftarrow \emptyset$
 3: for all $v \in V$ ordered by $\phi$ do
 4:     for all $e_{\downarrow} = (u, v) \in E \cup E' : \phi(u) > \phi(v)$ do
 5:         for all $e_{\uparrow} = (v, w) \in E \cup E' : \phi(v) < \phi(w) \wedge w \neq u$ do
 6:             $G'_v \leftarrow G'[\{z \in V \mid \phi(v) \leq \phi(z)\}]$
 7:             $R \leftarrow \Sigma \setminus \{\ell(e_{\downarrow}) \cup \ell(e_{\uparrow})\}$
 8:             $shortcutCost \leftarrow w(e_{\downarrow}) + w(e_{\uparrow})$
 9:             $witnessCost \leftarrow$ KLCSP-Dijkstra-Alt($G'_v, u, v, w, R$)
10:             if $shortcutCost < witnessCost$ then
11:                 $e' \leftarrow (u, w)$
12:                 $w(e') \leftarrow shortcutCost$
13:                 $\ell(e') \leftarrow \{\ell(e_{\downarrow}) \cup \ell(e_{\uparrow})\}$
14:                 $E' \leftarrow E' \cup \{e'\}$
15:                 $G' \leftarrow (V, E \cup E', w, \Sigma, \ell)$
16:             end if
17:         end for
18:     end for
19: end for
20: return $G'$
```

*$P^R_{u,w} = \langle u, v, w\rangle$ is true or $P^R_{u,w} \subseteq G'_v \setminus \{v\}$ will **eventually** be true (specifically, by the time we are finished contracting node $v$).*

**Proof.** It suffices to consider the case where $P^R_{u,w} \neq \langle u, v, w\rangle$ in $G'_v$ and $P^R_{u,w} \nsubseteq G'_v \setminus \{v\}$. This implies that $v \in P^R_{u,w}$, and, therefore, $P^R_{u,w} = \langle u, \ldots, x, v, y, \ldots, w\rangle$ such that $u \neq x$ and/or $y \neq w$. However, since $P^R_{x,y} = \langle x, v, y\rangle$, then when the construction algorithm (eventually) processes the edges $e_\downarrow = (x, v)$ and $e_\uparrow = (v, y)$, the algorithm will be forced to add shortcut edge $(x, y)$, by definition. Therefore, when the contraction of $v$ is complete, there must exist a path $P^R_{u,w} = \langle u, \ldots, x, y, \ldots, w\rangle \subseteq G'_v \setminus \{v\}$. ■

Note that this property holds true even in the context of the original KLCSP-Contraction algorithm. However, for the original construction algorithm, we had to prove the stronger claim that, if $P^R_{u,w} \neq \langle u, v, w\rangle$, then $P^R_{u,w} \subseteq G'_v \setminus \{v\}$ must (already) be true. This is because the previous algorithm would only avoid adding a $(u, w)$ shortcut if this latter condition already held. However, the new local search algorithm is able to find witness paths of the form shown in the above lemma (to detect that $P^R_{u,w} \subseteq G'_v \setminus \{v\}$ will eventually be true), and no shortcut edge will be added in this scenario.

We now have that the KLCSP-Contraction algorithm and the KLCSP-Contraction-Alt algorithm will both only add a shortcut edge $(u, w)$ if $P^R_{u,w} = \langle u, v, w\rangle$ (and this is the only shortest path) when processing edges $e_\downarrow$ and $e_\uparrow$ (based on the properties of Lemmas 3 and 5, respectively). Therefore, they will generate the exact same shortcut edge set for a given ordering $\phi$. Correctness and minimality of the KLCSP-Contraction-Alt algorithm thus follows from equivalence.

# Chapter 4

# Shortest Paths with Parameterized Restrictions

## 4.1 Introduction

Routing in road networks is one of the showcases of algorithm engineering. In the last decade, much research has been done to develop increasingly more efficient algorithms to solve shortest-path problems in road networks with fixed, non-negative edge weights. However, these algorithms are generally restricted to static query scenarios and further research is needed to develop efficient algorithms for more flexible scenarios. Speaking in broad terms, a shortest-path algorithm for a *flexible scenario* answers a shortest-path query from a source to a target node subject to additional, dynamic query parameters. This chapter considers the flexible scenario with *edge restrictions* regarding properties of the represented roads. A query then can specify the properties of roads that should not be included in the resulting shortest paths. Considering such edge restrictions is of high practical relevance, as they are often necessary for guaranteeing the feasibility of the paths in real-world applications. For example, large delivery trucks may not be able to travel certain roads, depending on their size and weight, due to height and weight limits of overpasses and bridges, respectively. However, smaller vehicles, such as cars, may still be allowed to use such roads. In principle, one could create different graphs for obeying different restrictions, and apply the basic speed-up algorithms already developed. However, this is highly inefficient, as it duplicates a lot of work and results in redundant data storage, since each restriction scenario is based on the same road network. Therefore, we augment existing speed-up techniques to this flexible scenario so that duplicate work is avoided. Furthermore, we develop new algorithmic ingredients tailored to this scenario to make our algorithms more efficient.

### 4.1.1 Related Work

A survey of the many existing techniques for speeding up basic shortest-path queries can be found in [35]. Nearly all of these techniques rely on some variant of the classical Dijkstra's algorithm [39], in which a shortest-path tree is constructed by performing a weighted, unidirectional search from the starting node, $s$, until the target node, $t$, is reached. A common extension of this search algorithm performs a bidirectional search, in which both a forward search from $s$ and a backward search[1] from $t$ are simultaneously performed until the searches meet. A more detailed description of this bidirectional Dijkstra search may be found in [54].

The existing body of research on preprocessing techniques can be classified into three general categories: hierarchical techniques, goal-directed techniques, and combinations of the two. Hierarchical techniques (such as [53, 65, 101]) seek to order the nodes and/or edges within the graph into hierarchically-nested levels, based on some measure of overall importance within the graph structure. Shortest-path queries carried out on

[1]Backward search in a graph $G = (V, E)$ is the equivalent of performing a standard (i.e., forward) search in the graph $\bar{G} = (V, \bar{E})$, where $\bar{E} = \{(v, u) \mid (u, v) \in E\}$.

a hierarchical graph progressively search towards higher (i.e., more important) levels of the graph hierarchy, while bypassing lower (i.e., less important) levels of the hierarchy, thus effectively reducing the overall search space explored by the query. One of the most efficient hierarchical preprocessing techniques to-date is that of Contraction Hierarchies (CH) [53], which will be discussed in further detail in a later section.

Goal-directed search techniques attempt to "direct" the shortest-path search towards some explicit target node (i.e., the "goal"), in order to speed up the overall query time. Two of the most effective goal-directed search techniques based on preprocessing are Arc-Flags [64, 77] and the ALT [54] algorithm. Arc-flags involves partitioning the graph into $k$ separate subgraphs, and, for each edge, a vector of $k$ binary "flags" is established to indicate whether that edge lies on some shortest path into each partition, respectively. This can then be used by the shortest-path search algorithm to filter any edges whose flag for the target node's partition indicates that it cannot belong to the shortest path. The ALT algorithm is based on the concepts of $A^*$ [62] search, which uses a *potential function* to estimate the shortest-path distance from a given node to the search target. ALT involves preprocessing which selects a set of *landmark* nodes, $L$, and then, for each landmark node, $l \in L$, calculates the shortest-path distance to/from every node, $v$, in the graph: $d(l, v)$ and $d(v, l)$, respectively. For any node, $v$, with target node, $t$, the *triangle inequality* provides two lower bounds for each landmark, $l \in L$: $d(l, t) - d(l, v) \leq d(v, t)$ and $d(v, l) - d(t, l) \leq d(v, t)$. The maximum of these lower bounds is used during a bidirectional $A^*$ search.

Goal-directed techniques have also previously been successfully combined with several hierarchical preprocessing techniques (e.g., see [23] for a discussion). Some of the more efficient hybrid implementations rely on the concept known as *Core-ALT* [23], which is a variant of ALT in which landmark distances are only established for some *core* subset of important nodes (e.g., nodes highest in the hierarchy). For nodes not within the core, we rely on their closest core nodes, known as their *proxy* nodes, to establish valid lower bounds from the preprocessed landmark distances in the core. Further details on Core-ALT will be provided in a later section. This technique has already been effectively incorporated into the CH-related preprocessing technique for supporting a flexible scenario with multiple edge weights in [51].

### 4.1.2 Our Contributions

In this chapter, we explore a new type of shortest-path query, in which the query can be dynamically parameterized to constrain the type of edges which may be included in the resulting shortest path. More specifically, each edge has an associated set of boolean and/or scalar values representing both qualitative and quantitative information about the edge, respectively (e.g., toll road: no, max. vehicle height: 5m). The query then specifies a constraint for each value type (e.g., restrict toll road: yes, vehicle height: 3m), thus limiting the edges which may be considered valid for a solution path.

We demonstrate how to correctly and efficiently incorporate the constraints of this new problem type into the well-known Contraction Hierarchies preprocessing technique for speeding up the resulting shortest-path queries. Initial research on extending Contraction Hierarchies for supporting edge restrictions was originally presented in [94]. However, this research focused only on a single type of edge restriction (discussed further below). This chapter serves as an extension of this initial research, with many additional optimizations and experimental results. Specifically, we present several effective algorithmic optimizations for further improving the overall speed and scalability of the resulting preprocessing and query algorithms. This includes the addition of goal-directed search techniques, search space pruning techniques, and generalizing the constraints of the local search during contraction. Experiments are then presented for two of the largest real-world road networks in the world: the North American and European road networks. The results of these experiments showcase the general effectiveness and scalability of our proposed methodology to large-scale, real-world graphs.

The remainder of the chapter is organized as follows. In Section 4.2, we present the concept of edge restrictions as constraints on the allowable edges for a given shortest-path query. We also formalize a universal framework for simultaneously supporting two general classes of edge restriction types: label restrictions [94] (e.g., a boolean toll road label) and parameterized restrictions (e.g., a vehicle height limit). Section 4.3 presents an overview of Contraction Hierarchies, along with the primary extensions necessary to correctly support flexible edge restrictions. Section 4.4 extends this technique further with the concepts of goal-directed search. Section 4.5 presents several additional optimizations to the basic approach given in Section 4.3 for further improving preprocessing and query times. Section 4.6 presents the experimental analysis of this technique. Finally, Section 4.7 concludes the chapter.

## 4.2 Edge Restrictions

Within the context of this chapter, we consider two general types of edge restrictions: label restrictions and parameterized restrictions. For clarity, we first define each of these concepts separately in the subsections below. However, for simplicity, we then establish a single, unified framework for simultaneously supporting both types of edge restrictions.

### 4.2.1 Label Restrictions

Originally introduced in [94], label restrictions are the most basic type of edge restriction possible. In this context, the graph is defined with a fixed set of possible edge classes (e.g., highways, toll roads, ferries, unpaved roads, etc.). Each edge is then assigned a subset of "labels" to indicate which class(es) it belongs to in the graph. A shortest-path query can then be dynamically adjusted according to which labels (i.e., classes) of edges should be excluded from the resulting shortest path. For example, a traveler may wish to find the shortest path between two locations which avoids both toll roads and ferries, or, alternatively, they might prefer to find the shortest path which avoids only unpaved roads. Also note that, in addition to excluding a certain label (e.g., highway), we may also force its presence on all edges of the computed path. This can be easily done by putting an inverted label (e.g., "no highway") on the complementing set of edges and then excluding edges with this inverted label.

### 4.2.2 Parameterized Restrictions

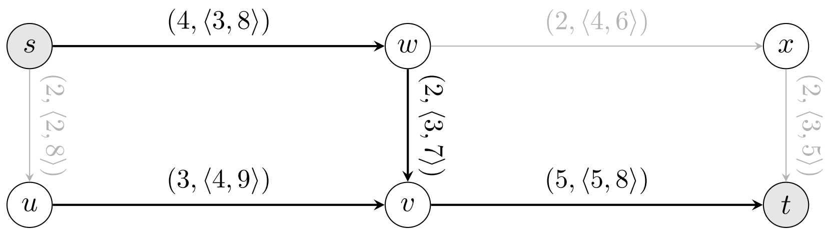
The second class of restrictions, known as parameterized restrictions, are introduced for the first time here. This class of restrictions defines a slightly more flexible form of constraint as compared to the label restrictions, discussed above. For parameterized restrictions, the graph is defined with a fixed set of parameter types (e.g., vehicle height, vehicle weight), whose values are to be specified at query time. Each edge is then assigned a threshold value defining an absolute limit for each parameter type, beyond which the edge will be restricted for that parameter type. For example, if an edge represents a section of road which travels under an overpass or through a tunnel, then its "vehicle height" threshold value will be equal to that of the height clearance for this overpass or tunnel, so that no vehicle taller than this threshold can be routed on this section of road. A shortest-path query can then be dynamically adjusted by providing different combinations of parameter values for each parameter type (e.g., different vehicle types will have different heights, weights, etc., resulting in potentially different shortest paths between the same two locations for the different vehicles).

### 4.2.3 Unified Framework

So far, we have presented a basic intuition for how edge restrictions are intended to work, in general, across two different classes of edge restrictions. We now define and further unify these concepts more formally as follows. Let $G = (V, E)$ be a directed graph, where $V$ is the set of nodes and $E$ is the set of edges, with non-negative edge *weight* function $c : E \to \mathbb{R}_{>0}$ and edge *threshold* function vector $\tau = \langle \tau_1, \tau_2, \ldots, \tau_r \rangle$, such that $\forall\, i \in [1, r]$, $\tau_i : E \to \mathbb{R}_{>0} \cup \{\infty\}$ is a function mapping edges in $G$ to a real-valued *threshold*, or $\infty$ if the edge is not restricted. Given this graph definition, we have that an *Edge-Restricted Shortest-Path* (ERSP) query is of the form $q = \langle s, t, p \rangle$, where $s \in V$ is the starting node, $t \in V$ is the target node, and $p = \langle p_1, p_2, \ldots, p_r \rangle$ is the set of parameters that constrain the query (henceforth referred to as the *query constraint parameters*). Given any query specification, $q$, of the above form, we define the ERSP to be the shortest path, $P_q$, with respect to edge weight function $c$, in the edge-restricted graph $G_p = (V, E_p)$, where $E_p = \{e \in E \mid \forall\, i \in [1, r],\, p_i \leq \tau_i(e)\}$.[2] An example of an edge-restricted graph and query is given in Figure 4.1.

Additionally, this particular form of parameterized edge restrictions subsumes the simpler form of label restrictions discussed earlier. More specifically, for any given

[2]We use threshold values only as upper bounds to simplify the description in this chapter, but our algorithm can easily be modified to handle lower bound threshold values, too.



$(4, \langle 3, 8\rangle)$
$(2, \langle 4, 6\rangle)$
$s$
$w$
$x$
$(2, \langle 2, 8\rangle)$
$(2, \langle 3, 7\rangle)$
$(2, \langle 3, 5\rangle)$
$(3, \langle 4, 9\rangle)$
$(5, \langle 5, 8\rangle)$
$u$
$v$
$t$

Figure 4.1: For this graph, we have threshold function vector $\tau = \langle \tau_1, \tau_2 \rangle$. Each edge, e, is labeled as $(c(e), \langle \tau_1(e), \tau_2(e) \rangle)$. For query $q = \langle s, t, \langle 3, 7 \rangle \rangle$, the original graph is shown here with edge set $E_p$ in thick black and all other (restricted) edges in grey. For this query, $P_q = \langle (s, w), (w, v), (v, t) \rangle$.

class of edge types (e.g., highways, toll roads, ferries), we may simply define a new threshold function, $\tau_j$. For any edge, $e \in E$, we then have that $\tau_j(e) = 0$ if the edge belongs to class $j$ and $\tau_j(e) = \infty$, otherwise. Then, for any associated query in which we wish to restrict all edges of class $j$, we set $p_j = \infty$ in the query definition. It is clear to see that, by definition, no edges with that particular classification can then belong to the edge-restricted graph for this query. This formulation therefore provides us with a single, universal framework which is now general enough to support even the most basic of restrictions (i.e., label restrictions) yet flexible enough to support the more powerful parameterized restrictions.

**Definition 6** *(Minimum Threshold) Given two edges e and e′, we define the* minimum threshold *vector for these two edges to be:*

$$\tau(e) \curlyvee \tau(e') = \langle min(\tau_1(e), \tau_1(e')), min(\tau_2(e), \tau_2(e')), \ldots, min(\tau_r(e), \tau_r(e')) \rangle$$

**Definition 7** *(Maximum Threshold) Given two edges e and e′, we define the* maximum threshold *vector analogously:*

$$\tau(e) \curlywedge \tau(e') = \langle max(\tau_1(e), \tau_1(e')), max(\tau_2(e), \tau_2(e')), \ldots, max(\tau_r(e), \tau_r(e')) \rangle$$

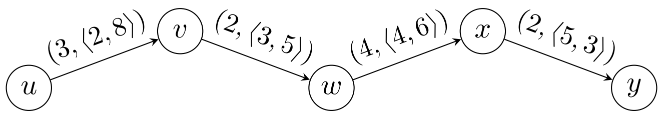


Figure 4.2: For the path $P = \langle (u, v), (v, w), (w, x), (x, y)\rangle$, we have $\tau(P) = \langle 2, 3\rangle$.

The minimum/maximum threshold operators define the most/least restrictive set of threshold values between the two edges' threshold function vectors, respectively.

**Definition 8** *(Path Threshold) For any path, $P = \langle e_1, e_2, \ldots, e_k\rangle$, we define the* path threshold *as the minimum threshold over all of its edges (e.g., see Figure 4.2):*

$$\tau(P) = \tau(e_1) \curlyvee \tau(e_2) \curlyvee \ldots \curlyvee \tau(e_k)$$

The path threshold operation effectively defines the most restrictive set of possible query constraint parameters for which the associated path will still be unrestricted.

**Definition 9** *(Threshold Dominance) Given two threshold function vectors $\tau$ and $\tau'$, we define a* weak dominance *relation as follows:*

$$\tau \preceq \tau' \Leftrightarrow \forall i \in [1, r],\ \tau_i \leq \tau'_i$$

$$\tau \npreceq \tau' \Leftrightarrow \exists i \in [1, r],\ \tau_i > \tau'_i$$

In this context, we say that $\tau'$ weakly dominates $\tau$ ($\tau \preceq \tau'$) if it is no more restrictive than $\tau$ for all possible restrictions. Likewise, we say that $\tau$ is non-dominated by $\tau'$ ($\tau \npreceq \tau'$) if it is less restrictive than $\tau'$ for at least one restriction. We additionally apply this notation to reflect the relation between any query constraint parameters, $p = \langle p_1, \ldots, p_r\rangle$, and the threshold function vectors. For example, we say that $p \preceq \tau(e) \Leftrightarrow \forall i \in [1, r]$, $p_i \leq \tau_i(e)$ to indicate that $p$ is not restricted by $\tau(e)$. A similar notation holds for $\npreceq$ as well.

## 4.3 Contraction Hierarchies with Parameterized Edge Restrictions

The Contraction Hierarchies (CH) technique involves preprocessing a directed graph $G = (V, E)$, with edge weight function $c : E \to \mathbb{R}_{>0}$. Each node in the graph is ordered according to some notion of relative importance. The CH is then constructed by *contracting* the nodes in increasing order of importance. Contracting a node $u$ means removing $u$ from the graph without changing shortest-path distances between the *remaining* (more important) nodes by adding *shortcuts.* A shortcut is an edge $(v, w)$ representing a whole path $\langle v, \ldots, w \rangle$ in the original graph.

**Node Contraction.** When a node $u$ is contracted, we only preserve the shortest-path distances between the neighbors of $u$. This is sufficient, as any shortest path that uses node u in the middle has a subpath between two neighbors of u, which we preserve. Therefore, given two currently uncontracted neighbors, $v$ and $w$, with edges $(v, u)$ and $(u, w)$, we perform a shortest-path search to find the shortest path $P$ between $v$ and $w$ in the remaining graph avoiding $u$. When the length of $P$ is longer than the length of the path $\langle v, u, w \rangle$, we add the necessary shortcut between $v$ and $w$ with weight $c(v, u) + c(u, w)$ to the remaining graph. Otherwise, $P$ is a *witness* that no shortcut is necessary (i.e., the shortest path distances remain preserved even without the shortcut). The search explores usually only a small part of the graph and is therefore called *local witness search.*

**Query.** Once all necessary shortcuts are added to the graph $G$ for a given ordering, shortest-path queries may then be carried out using a bidirectional Dijkstra search variant which performs a simultaneous forward search from $s$ in the *upward* graph $G^{\uparrow} =$

$(V, E^{\uparrow})$, where $E^{\uparrow} = \{(u, w) \in E \mid u \text{ contracted before } w\}$, and backward search from $t$ in the *downward* graph $G^{\downarrow} = (V, E^{\downarrow})$, where $E^{\downarrow} = \{(v, u) \in E \mid\ v \text{ contracted after } u\}$. A tentative shortest-path distance is maintained and is updated only when the two search frontiers meet to form a shorter path. The search in a given direction may be aborted once the minimum key for the priority queue in that direction exceeds the length of the best tentative path seen so far. Once both search directions are finished, the best path seen thus far represents the shortest-path distance.

**Node Ordering.** Assume that each node $u$ has a *priority* on how attractive it is to contract $u$. Then, we contract a most attractive node and update the priorities of its remaining adjacent nodes. We repeat this until all nodes are contracted. This results in an ordering of the nodes and, at the same time, adds all necessary shortcuts to the graph. In practice, the priority values are typically defined as some linear combination of multiple heuristically-useful terms. The reader is referred to Section 3 of [53] for a detailed discussion of many of the most commonly-used terms in practice. The priority defined for our results in this chapter is a linear combination of two terms. The first term is the edge difference between the number of necessary shortcuts to contract $u$ and the number of incident edges to remaining nodes. The second term is the sum of the original edges represented by the necessary shortcuts to contract $u$. We weight the first term with 1 and the second term with 2 (these weights were determined from initial experiments on graphs with label restrictions in [94]).

### 4.3.1 Incorporating Edge Restrictions

In the following, we show how to further preserve shortest-path distances within the CH preprocessing for the more flexible edge-restricted shortest-path queries as well.

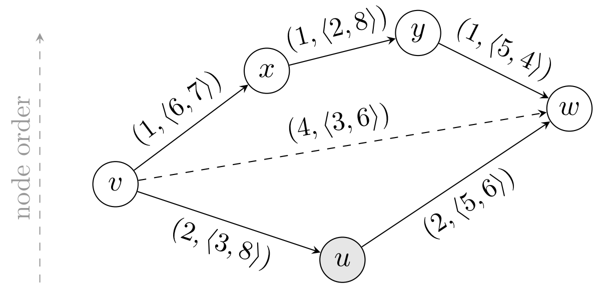


Figure 4.3: Contracting node $u$ for flexible edge restrictions. The shortcut edge is represented by a dashed line.

Given a graph $G = (V, E)$, with weight function $c : E \to \mathbb{R}_{>0}$ and threshold function vector $\tau = \langle \tau_1, \tau_2, \ldots, \tau_r \rangle$, we are able to incorporate the constraints of edge restrictions into the CH preprocessing as follows. As with static CH, we still must establish some absolute priority order for contracting the nodes.

However, now, when contracting a node $u$ we need to consider edge restrictions. The potential shortcut $(v, w)$ represents the path $\langle v, u, w \rangle$ with path threshold $p = \tau(v, u) \curlyvee \tau(u, w)$. Therefore, to avoid the shortcut, we need to find a witness path whose *witness path threshold* weakly dominates $p$. We do this by performing the local witness search in the graph $G_p$, as defined earlier. As before with the static CH preprocessing, we then only add a shortcut edge $(v, w)$ if the resulting path $P$ has greater length than the path $\langle v, u, w \rangle$, and we omit it otherwise. For example, in Figure 4.3, we show the result of contracting node $u$ on the example graph. In this case, we have that $p = \tau(v, u) \curlyvee \tau(u, w) = \langle 3, 6 \rangle$. Since there is no valid path, $P$, for this witness search which ignores node $u$ (i.e., edge $(x, y)$ is restricted for $p_1 = 3$ and edge $(y, w)$ is restricted for $p_2 = 6$), then a shortcut edge $(v, w)$ must be added with weight $c(v, u) + c(u, w) = 4$ and threshold function vector $p$.

Note that contrary to the static scenario, we cannot omit *parallel edges*[3]. We need to keep parallel edges for each Pareto-optimal pair of threshold vector and edge weight. For example, we may have an edge with weight 2 and threshold $\langle 2, 6\rangle$, and a parallel edge with weight 4 and threshold $\langle 3, 8\rangle$. We cannot just keep the edge with weight 2, as a query with constraints $\langle 3, 8\rangle$ must not use this edge. However, we identify edges by their two endpoints in cases where the particular edge is clear from the context.

**Lemma 10** *Consider the contraction of node $u$ by our CH with Flexible Edge Restrictions algorithm. Let $v$, $w$ be two uncontracted neighbors of $u$ with edge $(v, u)$, $(u, w)$. Let $p$ be a set of query constraint parameters with $p \preceq \tau(v, u) \curlyvee \tau(u, w)$. Either there exists a witness path $P$ with $c(P) \leq c(\langle v, u, w\rangle)$ and $p \preceq \tau(P)$ or a shortcut of weight $c(\langle v, u, w\rangle)$ and threshold vector $\tau(v, u) \curlyvee \tau(u, w)$ is added.*

**Proof.** Follows directly from our definition of node contraction and the transitivity of the threshold dominance. ■

After preprocessing is complete, that is, all necessary shortcuts are added to $G$, shortest-path queries of the form $q = \langle s, t, p\rangle$ may then be carried out. We perform a simultaneous forward search from $s$ in the upward edge-restricted graph $G_p^{\uparrow} = (V, E_p^{\uparrow})$, where $E_p^{\uparrow} = \{e = (u, w) \in E \mid u \text{ contracted before } w \wedge p \preceq \tau(e)\}$, and backward search from $t$ in the downward edge-restricted graph $G_p^{\downarrow} = (V, E_p^{\downarrow})$, where $E_p^{\downarrow} = \{e = (v, u) \in E \mid v \text{ contracted after } u \wedge p \preceq \tau(e)\}$.

**Theorem 11** *Given a graph $G = (V, E)$ constructed by the CH with Flexible Edge Restrictions algorithm, our query algorithm is correct for any query.*

**Proof.** Let $q = \langle s, t, p\rangle$ be a fixed query. Lemma 10 ensures that we add all necessary shortcuts such that $G_p = (V, E_p)$ is a CH. As our query corresponds to a CH query

[3]Parallel edges are unique edge instances between the same pair of nodes, but with potentially different properties; e.g., see Figure 4.5 for an example.

on $G_p$, the correctness of our algorithm follows from the correctness of the CH query algorithm. ■

## 4.4 Goal-Directed Search

In this section, we demonstrate how to further enhance the basic approach of our suggested algorithm (presented above) using a variant of the ALT algorithm. The ALT technique has been previously studied within the context of dynamic graphs in [36], where it is noted that the potential functions computed from landmarks in ALT remain correct for dynamic scenarios in which edge weights are only allowed to increase from their original value. Therefore, it is easy to see that ALT remains correct even for ERSP queries, since restricting an edge is equivalent to increasing its weight to infinity for the duration of the query.

### 4.4.1 Goal-Direction in the Core Graph

As discussed briefly in Section 4.1.1, Core-ALT is a variant of ALT in which landmark distances are established only for some relatively small, *core* of the "uppermost" (i.e., most important) nodes in the hierarchy. This can be seen to greatly reduce the space overhead for storing landmark distances, as compared to the original ALT.

The original Core-ALT [23] uses an *uncontracted core*. That is, all nodes but the core nodes get contracted as described in Section 4.3. This is useful when a contraction of the core nodes would be impractical for performance or scalability reasons. For example, experiments have shown that the majority of the preprocessing time is spent contracting the uppermost nodes in the hierarchy, where the remaining subgraph can become quite dense due to the addition of too many shortcuts. Aborting contraction at the core allows us to avoid the additional preprocessing time necessary to contract the remaining, relatively-dense subgraph of the core. Due to the contraction of the non-core nodes, there are shortcuts present in the core to ensure that the shortest-path distances are preserved. So a regular bidirectional Dijkstra search can be performed in the core. Using core landmarks is especially helpful in this context for speeding up this search within the core.

Better query times, however, may be achieved with a *contracted core* [51]. All nodes, including the core nodes, get contracted. Now, we can use a CH query algorithm on the core, that is additionally guided by landmarks.

We incorporate core-based routing as described by [23]. More precisely, for a given query $q = \langle s, t, p \rangle$, a three-phased search algorithm is established as follows. In the first phase, we perform the bidirectional query algorithm exactly the same as before, except we prune the search at any core nodes. Specifically, we record each reached core node and its associated distance from either $s$ or $t$, but we do not relax any of its outgoing/incoming edges, respectively. If we find the shortest path during this phase, we are finished. If not, we continue to the second phase, in which we perform a backward search from the source node, $s$, and a forward search from the target node, $t$, until each search direction reaches its closest core node (i.e., its *proxy* node): $s'$ and $t'$, respectively. This gives us the shortest-path distances $d(s', s)$ and $d(t, t')$, respectively. In the final phase, we then reinsert all core nodes reached during the first phase, with their respective distances, into the priority queues and continue the bidirectional search in the core using the following $A^*$ potential function values[4]. For a given node, $v$, reached by the forward core search, and a landmark $l \in L$, we compute a lower bound on the distance to the

[4]If either search direction fails to reach a core node in the second phase, then we employ a zero-value potential function for that search direction.

target: $max\{d(l,t') - d(l,v) - d(t,t'), d(v,l) - d(t',l) - d(t,t')\}$. Likewise, for a node, $v$, reached by the backward search, we use the following lower bound on the distance to the source: $max\{d(l,v) - d(l,s') - d(s',s), d(s',l) - d(v,l) - d(s',s)\}$. The tightest lower bounds are computed as the maximum over all landmarks.

### 4.4.2 Multiple Landmark Sets

As is discussed in [94], using ALT landmark distances for ERSP queries can result in ineffective potential functions for very heavily-constrained shortest-path queries. This is due primarily to the fact that, even though the potential functions remain correct, the resulting lower bounds become much weaker (i.e., less accurate) when large numbers of edges become restricted.

In an attempt to alleviate this problem, we propose the new concept of using *multiple landmark sets*, $\mathcal{L} = \{L_1, L_2, \ldots, L_k\}$. In this context, we define a landmark set, $L \subseteq V$, as being a set of landmarks created specifically for a single set of query constraint parameters, $p_L = \langle p_1, p_2, \ldots, p_r \rangle$. The distances for the landmark set $L$ are based on the edge-restricted graph determined by the query constraints of $p_L$.

Using this concept of multiple landmarks, for a given query $q = \langle s, t, p \rangle$, we may then choose only from the set of landmarks $\mathcal{L}_p = \{L \in \mathcal{L} \mid p_L \preceq p\}$. That is, we may select only from the sets of landmarks, $L$, whose associated query constraint parameters, $p_L$, are no more restrictive than the incoming query constraint parameters, $p$. By that, we maintain valid lower bounds for the resulting potential function values. We then select the best set of active landmarks to use from $\mathcal{L}_p$, i.e., the few landmarks which provide the best potential function values from $s$ to $t$, for both search directions. Using multiple landmark sets allows us to derive better potential functions, in general, by more closely approximating the shortest path distances associated with the dynamic query constraints of $p$.

One possibility to support this for flexible edge restrictions would be to establish a unique landmark set for all possible combinations of query constraint values. However, this is clearly infeasible for any real-world implementation. A more realistic approach would therefore be to establish a unique landmark set specific to each unique threshold function, $\tau_i$, in the threshold function vector for the graph. That is, $\mathcal{L} = \{L_1, L_2, \ldots, L_r, L_{r+1}\}$, such that, $\forall i \in [1, r]$, $p_{L_i} = \langle p_1 = 0, \ldots, p_i = \infty, \ldots, p_r = 0\rangle$ (i.e., each landmark set $L_i$ restricts only edges with finite threshold values for $\tau_i$). Landmark set $L_{r+1}$ is established for the fully *unrestricted* set of query constraint parameters, $p_{L_{r+1}} = \langle 0, 0, \ldots, 0\rangle$, to ensure that there is always at least one set of valid landmarks to choose from.

## 4.5 Optimizations

In this section, we present several additional algorithmic enhancements to the basic approach of our algorithm, in order to further optimize the overall speed and scalability of this approach. Many of the ideas presented in subsection 4.5.1 (below) are further extensions to approaches originally proposed in [53] and [94].

### 4.5.1 Improved Local Search

**Multi-Target Local Search.** With flexible edge restrictions, to ensure that only a minimal number of shortcut edges are added for a given node ordering, a local witness search must be carried out separately for every pair of neighbors for which a shortcut may need to be added [94]. This is necessary, since the exact constraints applied to any particular local witness search will depend upon the threshold values of the specific pair of edges being bypassed by a given (potential) shortcut edge. Therefore, when contracting a node, $u$, where $I_u^{\downarrow} = \{(v,u) \in E \mid v \text{ contracted after } u\}$ and $O_u^{\uparrow} = \{(u,w) \in E \mid u \text{ contracted before } w\}$, the minimal algorithm must perform a total of $|I_v^{\downarrow}| \cdot |O_v^{\uparrow}|$ separate local witness searches[5]. However, if we do not require to add a minimal number of shortcut edges, the overall efficiency of the contraction of $u$ can be improved as follows. We may instead perform only a single local witness search from the source, $v$, of each incoming edge $e_{\downarrow} = (v,u) \in I_u^{\downarrow}$ until all nodes in the set $W = \{w \in V \mid (u,w) \in O_u^{\uparrow}\}$ have been settled, or until a distance of $w(e_{\downarrow}) + max\{w(e_{\uparrow}) \mid e_{\uparrow} = (u,w) \in O_u^{\uparrow}, w \neq v\}$ has been reached [53]. We use the maximum threshold of all of the outgoing edges in $O_u^{\uparrow}$ to obtain the path threshold $p = \tau(v,u) \curlyvee \big( \curlywedge_{\forall e \in O_u^{\uparrow}} \tau(e)\big)$ (e.g., see Figure 4.4), similar to the approach used in [94]. We perform the multi-target local witness search in $G_p$. However, this does not affect the correctness of the resulting CH graph, since the local search constraint parameters, $p$, are guaranteed to be as or more restrictive than the query constraint parameters for any explicit pair with $(v,u)$ as the incoming edge. Therefore, any resulting witness paths are still valid, even though this approach may result in the addition of unnecessary shortcuts. As we will see in later experiments, this multi-target approach scales much better in practice.

**Limited Local Search.** We limit the depth of the shortest-path tree of the local search to 7 [53]. This way, we can only guarantee to compute upper bounds to the nodes. Shortcut decisions based on these upper bounds add all necessary shortcuts, but sometimes also some unnecessary ones. However, this limit also greatly accelerates precomputation.

[5]Pairs $\langle e_{\downarrow} = (v,u), e_{\uparrow} = (u,w)\rangle$ where $v = w$ may be ignored.

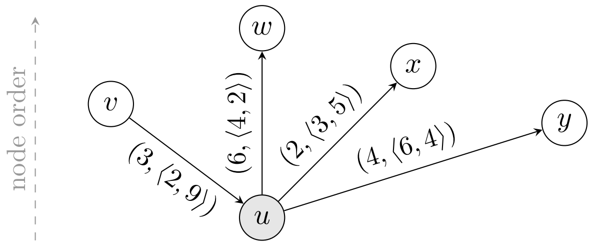

node order
$v$
$w$
$x$
$y$
$u$
$(3, \langle 2, 9\rangle)$
$(6, \langle 4, 2\rangle)$
$(2, \langle 3, 5\rangle)$
$(4, \langle 6, 4\rangle)$

Figure 4.4: Multi-target local search from $v$ to $W = \{w, x, y\}$. In this scenario, the query constraint parameters are defined as $p = \tau(v,u) \curlyvee (\tau(u,w) \curlywedge \tau(u,x) \curlywedge \tau(u,y)) = \langle 2,9\rangle \curlyvee (\langle 4,2\rangle \curlywedge \langle 3,5\rangle \curlywedge \langle 6,4\rangle) = \langle 2,9\rangle \curlyvee \langle 6,5\rangle = \langle 2,5\rangle$.

**Local Edge Reduction.** During the course of the contraction phase, it may also be beneficial to see if we can quickly detect and remove any edges which provably can not belong to any shortest paths for any possible parameter value combinations. The intended effect of this approach is to further reduce the size of the resulting CH graph, giving us better overall preprocessing and query times.

One simple, yet effective way of doing this is as follows. After performing a multi-target local search (discussed above) from the source, $v$, of some incoming edge $(v, u)$, we have computed a shortest-path tree rooted at node $v$. Let $P_x = \langle e_1, e_2, \ldots, e_k\rangle$ be a path in the tree from root node $v$ to any other node $x$. Then, for all outgoing edges $(v, x)$ from $v$, we may remove edge $(v, x)$ if $v$ is not the parent of $x$ in the shortest path tree (i.e., there is a shorter path from $v$ to $x$ other than $(v, x)$) and $\tau(v, x) \preceq \tau(P_x)$. In this case, it is easy to show that edge $(v, x)$ can never belong to any shortest path, for any possible parameter value combination, since any parameter values for which edge $(v, x)$ would be valid would also be valid for the path $P_x$, which is shorter than edge $(v, x)$. Therefore, edge $(v, x)$ can easily be removed in this scenario without affecting the correctness of any subsequent shortest-path queries on the CH graph.

**Biased Local Search.** Additionally, we may further minimize the number of necessary local searches per node contraction as follows. Let *in-degree*$(u)$ and *out-degree*$(u)$ represent the number of incoming and outgoing edges, respectively, for node $u$. If *in-degree*$(u) \leq$ *out-degree*$(u)$, then we can perform forward multi-target local searches from the source nodes of all incoming edges (as suggested above); however, if *in-degree*$(u) >$ *out-degree*$(u)$, then we can instead perform backward multi-target local searches from the target nodes of all outgoing edges. But as road networks have mostly bidirected edges, we did not implement this optimization.

### 4.5.2 Search Space Pruning

Here we present several useful techniques that allow us to quickly detect and skip over (or "prune") certain unnecessary edges or nodes during the bidirectional CH search query.

**Parallel Edge Skipping.** Due to the presence of multi-edges within the resulting CH graphs, a naïve shortest-path query on the upward/downward edge-restricted graphs may perform redundant or unnecessary work on any parallel edges during a given search. For example, in Figure 4.5, a query with constraint parameters $p = \langle 1, 3 \rangle$ is valid for all three parallel $(u, v)$ edges. If we relax these edges in an arbitrary order (e.g., in order of decreasing weight), this may result in multiple calls to the priority queue *DecreaseKey* function for the same target node $v$. However, if we sort and store the adjacent edges of a given node first by their target node and then by their weight, then, when we relax the first valid (i.e., unrestricted) edge leading to a given target node, we can automatically skip over any remaining parallel edges leading to that same target node, since no remaining parallel edges to that node can improve the current path distance

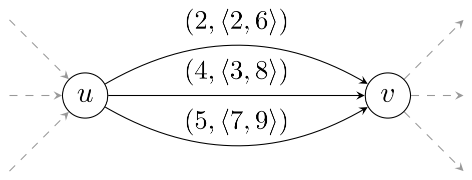


Figure 4.5: Skipping parallel edges. Edges to a given target node are accessed in order of increasing weight to allow pruning. For example, for a query with constraint parameters $p = \langle 1, 3\rangle$, we may skip the bottom two parallel edges from node $u$ to node $v$, since we can successfully relax the topmost edge.

(see Figure 4.5). This will save on unnecessary calls to the *DecreaseKey* function. Additionally, and perhaps more importantly, when incorporating goal-directed search into the query, skipping of parallel edges will also implicitly save on redundant calls to compute the potential function value for the same target node, which can be relatively expensive, depending upon the potential function being used.

**Computing Lower Bounds.** Instead of deriving the lower bounds from all landmarks in $\mathcal{L}_p$, we only use a subset of 4 landmarks that give the highest lower bounds on the $s$-$t$ distance [54]. This speeds up the overall query, as the computation of the lower bounds is much faster but still provides good lower bounds.

**Node Pruning Using Lower Bounds.** In the ALT algorithm, the heuristic potential function serves to establish a lower bound on the possible shortest-path distance of a given query. Since we maintain a tentative upper bound on the shortest-path distance for our current query, we may also skip over any nodes whose resulting key value (which is a lower bound on the length of a path that visits that node) is greater than or equal to the current upper bound seen thus far [57]. This simple optimization can also be seen to have a significant impact on the resulting shortest-path query times.

**Witness Restrictions.** Our algorithm described so far tends to have a much larger search space (i.e., a larger number of relaxed edges and settled nodes) for less restricted queries than for more restricted queries. For example, experiments on the European road network suggest that an unrestricted query can explore up to twice as many nodes, on average, compared to a fully-restricted query on the same network. This comes from the fact that more restricted queries will filter out more edges from the search. However, a lot of shortcuts are added during the node contraction that are only necessary for more restricted queries, while, for less restricted queries, there would be a witness preventing the shortcut. So we introduce the new concept of *witness restrictions*, that stores the information about these witnesses with the shortcuts. More precisely, this information is a set of witness path thresholds. There can be more than one witness path threshold for a given shortcut edge. For example, in Figure 4.6, we would store the set $\{\langle 1,6\rangle, \langle 2,5\rangle\}$ with the shortcut $(v, w)$. The shortcut is necessary, e.g., for a query with constraint parameters $\langle 3,5\rangle$, as this restricts both potential witnesses. But a query with constraint parameters $\langle 1,6\rangle$ does not need the shortcut, as a witness is available. Note that the witness paths considered for the set of witness path thresholds must be strictly shorter than the shortcut. This is necessary, as the shortcut may be used as a witness when all the interior nodes on the witness path get contracted before its source and target node. Also note that it is sufficient to store a Pareto-optimal set of witness path thresholds.

However, storing a whole set of witness path thresholds would require a variable amount of storage overhead per edge, which is impractical. Also, it is too time-consuming to compute all Pareto-optimal witness path thresholds. Therefore, for practicality, we will only store a single witness path threshold $\tau^*(e)$ with a shortcut $e$. For non-shortcuts, $e$, we set $\tau^*(e) = \langle -\infty, -\infty, \dots, -\infty\rangle$. During preprocessing, when a

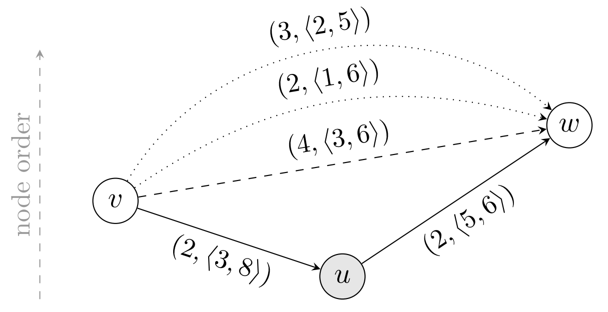

node order
$(3, \langle 2, 5\rangle)$
$(2, \langle 1, 6\rangle)$
$(4, \langle 3, 6\rangle)$
$v$
$w$
$u$
$(2, \langle 3, 8\rangle)$
$(2, \langle 5, 6\rangle)$

Figure 4.6: Contracting node $u$ with two witnesses (dotted) available for a subset of potential query constraint parameters.

new shortcut edge, $e = (v, w)$, is added to the CH graph, we then perform an additional unconstrained shortest-path query that tries to find the shortest overall witness path, $P_{v,w}$. We then set $\tau^*(v, w) = \tau(P_{v,w})$, and continue as before. Note that in our actual implementation, we delay the computation of $\tau^*(e)$ until one of its endpoints gets contracted. This improves the preprocessing performance, as the computation is then executed on a smaller remaining graph, and the edge reduction technique may have removed $e$ in between.

Our edge-restricted search graphs are now filtered as follows. The upward edge-restricted graph $G_p^{\uparrow} = (V, E_p^{\uparrow})$ is now defined such that $E_p^{\uparrow} = \{e = (u, w) \in E \mid u \text{ contracted before } w \ \wedge \ p \preceq \tau(e) \wedge p \npreceq \tau^*(e)\}$. The downward edge-restricted search graph is defined analogously. As will be seen in later experiments, using witness restrictions can result in smaller search space sizes and query times, especially for less restricted queries (e.g., resulting in speedups up to a factor of 1.5, on average, for unrestricted queries). The only downside of witness restrictions is that they increase the space required to store an edge. We therefore also propose to store them only for edges inside a core of the most important nodes, similar to the core-based landmarks described in Section 4.4.1.

## 4.6 Experiments

In this section, we present several detailed experiments highlighting the performance of our proposed algorithm for supporting edge restrictions in various query scenarios. Sections 4.6.1 and 4.6.2 describe our testing environment and our chosen test datasets, respectively. Section 4.6.3 provides experimental results showcasing our many proposed optimizations, and we present a progressive overview of how each successive optimization contributes to improve the overall performance of our solution. In Section 4.6.4, we examine the performance characteristics of the number of chosen restrictions for a given query. Finally, Sections 4.6.5 and 4.6.6 wrap up the experimental results with an in-depth comparison of our proposed solution against static CH.

### 4.6.1 Test Environment

All experiments were carried out on a 64-bit server machine running Linux CentOS 5.3 with 2 quad-core CPUs clocked at 2.53 GHz with 72 GB RAM (although only one core was used per experiment). All programs were written in C++ and compiled using gcc version 4.1.2 with optimization level 3.

### 4.6.2 Test Dataset

Experiments were carried out on two of the largest available real-world road networks: a graph of North America[6] and a graph of Europe[7]. Our North American graph has a total of $21,133,774$ nodes and $52,523,592$ edges, while our European graph has $40,980,553$ nodes and $94,680,598$ edges. Table 4.1 summarizes the differ-

[6]This includes only the US and Canada.

[7]This includes Albania, Andorra, Austria, Belarus, Belgium, Bosnia and Herzegovina, Bulgaria, Croatia, Czech Republic, Denmark, Estonia, Finland, France, Germany, Gibraltar, Greece, Hungary, Ireland, Italy, Latvia, Liechtenstein, Lithuania, Luxembourg, Macedonia, Moldova, Monaco, Montenegro, Netherlands, Norway, Poland, Portugal, Romania, Russia, San Marino, Serbia, Slovakia, Slovenia, Spain, Sweden, Switzerland, Turkey, Ukraine, United Kingdom, and Vatican City.

Table 4.1: Supported Restriction Types for the North American and European Graphs.

| | **Edges** | |
|---|---|---|
| **Restrictions** | North America | Europe |
| Ferry | 2,610 | 11,334 |
| Toll Road | 47,388 | 237,304 |
| Unpaved Road | 3,645,458 | 14,434,228 |
| Private Road | 1,662,314 | 1,957,380 |
| Limited Access Road | 682,396 | N/A |
| 4-Wheel-Drive-Only Road | 139,284 | N/A |
| Parking Lot Road | 160,850 | N/A |
| Hazmat Prohibited | 45,950 | N/A |
| All Vehicles Prohibited | 64,414 | 2,326,232 |
| Delivery Vehicles Prohibited | 148,010 | 3,674,338 |
| Trucks Prohibited | 475,472 | 5,347,498 |
| Taxis Prohibited | 147,628 | 3,765,140 |
| Buses Prohibited | 151,272 | 3,811,704 |
| Automobiles Prohibited | 114,192 | 3,772,628 |
| Pedestrians Prohibited | 1,253,030 | 1,653,448 |
| Through Traffic Prohibited | 2,050,562 | 7,210,664 |
| Height Limit | 23,873 | N/A |
| Weight Limit | 24,627 | N/A |

ent real-world restrictions supported by each graph dataset. Both datasets (including restrictions) were derived from NAVTEQ transportation data products, under their permission. Of the 18 unique restrictions, only two of these are true parameterized restrictions: Height and Weight Limit. The remaining 16 restrictions are label restrictions only. For the parameterized restrictions, the data distinguishes between 29 different height limit values and 57 different weight limit values.

Unless otherwise stated, all query performance results are averaged over the shortest-path distance queries between $10,000$ source-target pairs. Source and target nodes are selected uniformly at random. For each source-target pair, we performed an unrestricted and a fully-restricted query and we report the mean performance. This covers both ends of the restriction spectrum.

### 4.6.3 Engineering an Efficient Algorithm

There is a significant performance difference between the basic implementation of the idea of Section 4.3 and an efficient algorithm using all the techniques and optimisations of Sections 4.4 and 4.5. We analyze this in Table 4.2 for the North American network. Single-target (i.e., pairwise-edge) local search provides no feasible precomputation, as it does not finish within 3 days. With multi-target local search, we significantly decrease precomputation time to 16 hours. Additionally using edge reduction decreases the precomputation to 7 hours, and also reduces the query time from 7.3 ms to 4.6 ms. Skipping parallel edges reduces the number of relaxed edges by 70%. But as we still need to traverse the edges in the graph, and of course the same number of nodes is settled, the query time is only reduced by 7%. Using witness restrictions increases the precomputation by less than 20 minutes and improves the query time by 22%. However, it increases the space-consumption by 9 B/node. Using witness restrictions only on a core, we even get a slightly improved query time and virtually no space overhead for witness restrictions. Compared to the baseline bidirectional Dijkstra search, $\mathrm{MEPW_C}$ has a speed-up of more than 1,000 and even requires *less* space[8].

We can significantly decrease the precomputation time when we do not contract core nodes. As a query would then settle almost all nodes in the core, we use ALT in the core to improve query performance. We can trade precomputation time for query time by choosing different core sizes. It is important to use lower-bound pruning, as this reduces query time especially on large cores by around 22%. The resulting algorithm $\mathrm{MEPW_C G_1 10kUL}$ with a core size of 10,000 nodes has a preprocessing time of just 50 minutes, but the query time is more than 63% larger compared to $\mathrm{MEPW_C}$. Leaving

[8]This is possible as edges need to be stored only with the less important node.

Table 4.2: Experiments on the North American graph showing combinations of results for (S)ingle-Target Local Search, (M)ulti-Target Local Search, (E)dge Reduction, (P)arallel Edge Skipping, (W)itness Restrictions ($W_C$ for core-only witness restrictions for a core size of 10,000), and (G)oal-Directed Search for both single ($G_1$) and multiple ($G_N$) landmark sets with fixed core sizes of 10,000 (10k), 5,000 (5k), and 3,000 (3k) for both (U)ncontracted and (C)ontracted cores, with the option of (L)ower-Bound Pruning. All results are averaged for both unrestricted and fully-restricted queries over 10,000 random source-target pairs. The baseline bidirectional (D)ijkstra search (with no preprocessing) is also presented for comparison. Entries with a value of - were unavailable due to incompletion of the preprocessing within the allowed time.

| | **Preprocessing** | | **Queries** | | | |
|---|---|---|---|---|---|---|
| **Algorithm** | Time [H:M] | Space [B/node] | Time [ms] | Settled Nodes | Stalled Nodes | Relaxed Edges |
| D | 0:00 | 35 | 3,462.75 | 7,212,135 | N/A | 17,961,846 |
| S | 72:00+ | - | - | - | - | - |
| M | 16:05 | 33 | 7.29 | 1,033 | 624 | 49,436 |
| ME | 6:56 | 32 | 4.61 | 946 | 563 | 31,320 |
| MEP | 6:56 | 32 | 4.29 | 946 | 563 | 8,979 |
| MEPW | 7:15 | 41 | 3.36 | 834 | 450 | 6,769 |
| MEP$W_C$ | 7:11 | 32 | 3.27 | 855 | 470 | 6,881 |
| MEP$W_C$$G_1$10kU | 0:49 | 32 | 6.88 | 3,806 | 135 | 51,437 |
| MEP$W_C$$G_1$5kU | 1:19 | 32 | 5.34 | 2,342 | 184 | 36,607 |
| MEP$W_C$$G_1$3kU | 1:56 | 32 | 4.51 | 1,744 | 227 | 28,377 |
| MEP$W_C$$G_1$10kUL | 0:49 | 32 | 5.35 | 2,715 | 133 | 28,536 |
| MEP$W_C$$G_1$5kUL | 1:19 | 32 | 4.22 | 1,794 | 182 | 20,754 |
| MEP$W_C$$G_1$3kUL | 1:56 | 32 | 3.66 | 1,410 | 223 | 16,191 |
| MEP$W_C$$G_N$10kUL | 0:56 | 37 | 4.38 | 1,971 | 133 | 19,428 |
| MEP$W_C$$G_N$5kUL | 1:26 | 34 | 3.87 | 1,440 | 182 | 15,706 |
| MEP$W_C$$G_N$3kUL | 2:02 | 33 | 3.42 | 1,192 | 223 | 12,803 |
| MEP$W_C$$G_N$10kCL | 7:21 | 37 | 1.18 | 491 | 133 | 3,073 |
| MEP$W_C$$G_N$5kCL | 7:20 | 34 | 1.50 | 578 | 180 | 3,675 |
| MEP$W_C$$G_N$3kCL | 7:20 | 33 | 1.75 | 648 | 221 | 4,072 |

Table 4.3: Experiments on the European graph with the same settings as in Table 4.2.

| | **Preprocessing** | | **Queries** | | | |
|---|---|---|---|---|---|---|
| **Algorithm** | Time [H:M] | Space [B/node] | Time [ms] | Settled Nodes | Stalled Nodes | Relaxed Edges |
| D | 0:00 | 33 | 6,273.21 | 11,366,174 | N/A | 25,842,792 |
| S | 72:00+ | - | - | - | - | - |
| M | 40:11 | 31 | 10.09 | 1,240 | 761 | 58,854 |
| ME | 17:27 | 30 | 6.60 | 1,222 | 704 | 42,815 |
| MEP | 17:27 | 30 | 6.05 | 1,222 | 704 | 15,301 |
| MEPW | 18:22 | 39 | 4.90 | 1,091 | 572 | 11,717 |
| MEPW$_C$ | 18:10 | 30 | 4.52 | 1,119 | 601 | 11,897 |
| MEPW$_C$G$_1$10kU | 2:03 | 30 | 15.31 | 5,757 | 196 | 111,698 |
| MEPW$_C$G$_1$5kU | 3:36 | 30 | 12.99 | 3,726 | 287 | 87,378 |
| MEPW$_C$G$_1$3kU | 5:40 | 30 | 10.75 | 2,764 | 368 | 68,973 |
| MEPW$_C$G$_1$10kUL | 2:03 | 30 | 13.50 | 4,380 | 192 | 69,579 |
| MEPW$_C$G$_1$5kUL | 3:36 | 30 | 11.03 | 2,914 | 277 | 58,413 |
| MEPW$_C$G$_1$3kUL | 5:40 | 30 | 9.36 | 2,251 | 354 | 47,068 |
| MEPW$_C$G$_N$10kUL | 2:15 | 33 | 11.09 | 3,410 | 192 | 51,873 |
| MEPW$_C$G$_N$5kUL | 3:49 | 31 | 9.50 | 2,484 | 277 | 49,193 |
| MEPW$_C$G$_N$3kUL | 6:07 | 31 | 8.64 | 1,999 | 354 | 41,068 |
| MEPW$_C$G$_N$10kCL | 18:29 | 33 | 2.68 | 791 | 191 | 8,505 |
| MEPW$_C$G$_N$5kCL | 18:28 | 31 | 3.65 | 990 | 276 | 11,806 |
| MEPW$_C$G$_N$3kCL | 18:28 | 31 | 3.86 | 1,096 | 355 | 12,337 |

only 3,000 nodes uncontracted results in roughly 2 hours precomputation time, but increases the query time only by 12%. As we need to store the landmark distances only for core nodes, the space consumption does not visibly change.

We can further decrease the query time by using multiple landmark sets. This slightly increases the precomputation time by about 6–7 minutes. The space consumption increases by 3% (for 3k core) to 16% (for 10k core). The resulting query time for the 3k core is now only 5% above MEPW$_C$.

The fastest query times are achieved using landmarks on a large contracted core. On a 10k core, our MEPW$_C$G$_N$10kCL algorithm achieves a query time of 1.18 ms, that is 2,900 times faster than bidirectional Dijkstra.

Table 4.3 provides the performance results on the European network. We see

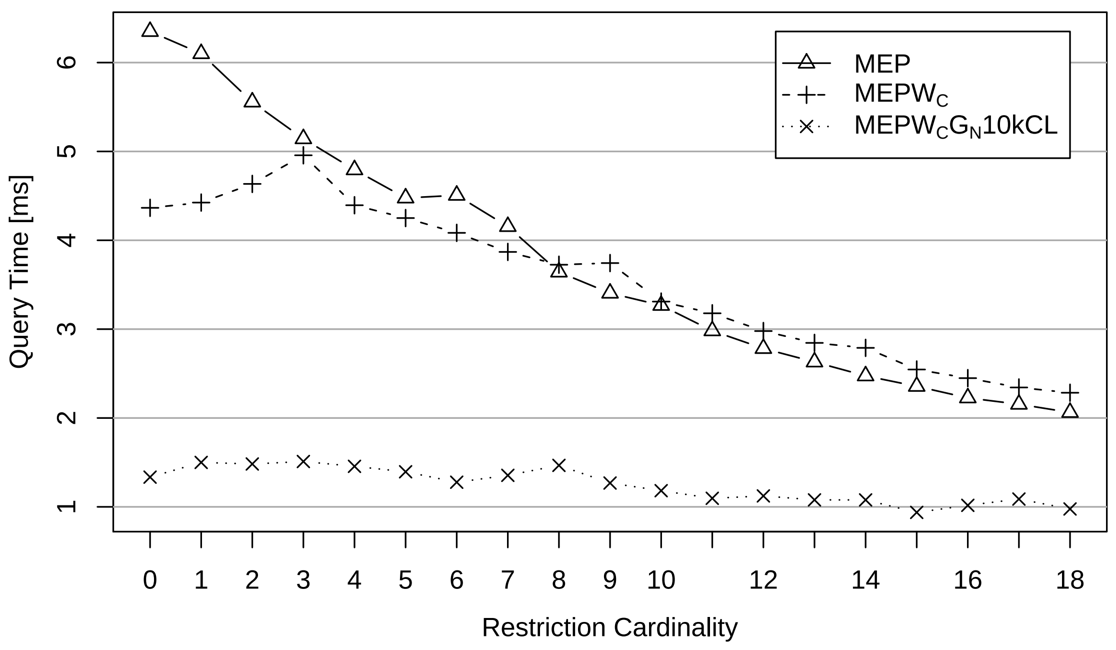


Figure 4.7: Experiments on the North American graph comparing the query times of the MEP, MEPW$_\mathrm{C}$, and MEPW$_\mathrm{C}$G$_N$10kCL configurations across different restriction cardinalities.

a similar performance compared to the North American network, although the absolute numbers are larger as the network is larger. In general, landmarks do not decrease the query time as much as for the North American network. The speed-up of MEPW$_\mathrm{C}$ over Dijkstra's algorithm is $\approx$ 1,300, while the best speed-up with landmarks is $\approx$ 2,300.

### 4.6.4 Restriction Cardinality

To assess the performance of our algorithm for different values of query constraint parameters, we measure query performance for different restriction *cardinalities* of the query constraint parameters $p = \langle p_1, p_2, \dots, p_r \rangle$. We define the cardinality of $p$ in this context as the number of $p_i \neq 0$. For uniformity with label restrictions, the parameterized restrictions are either set to 0 or the maximum value. On North America, for

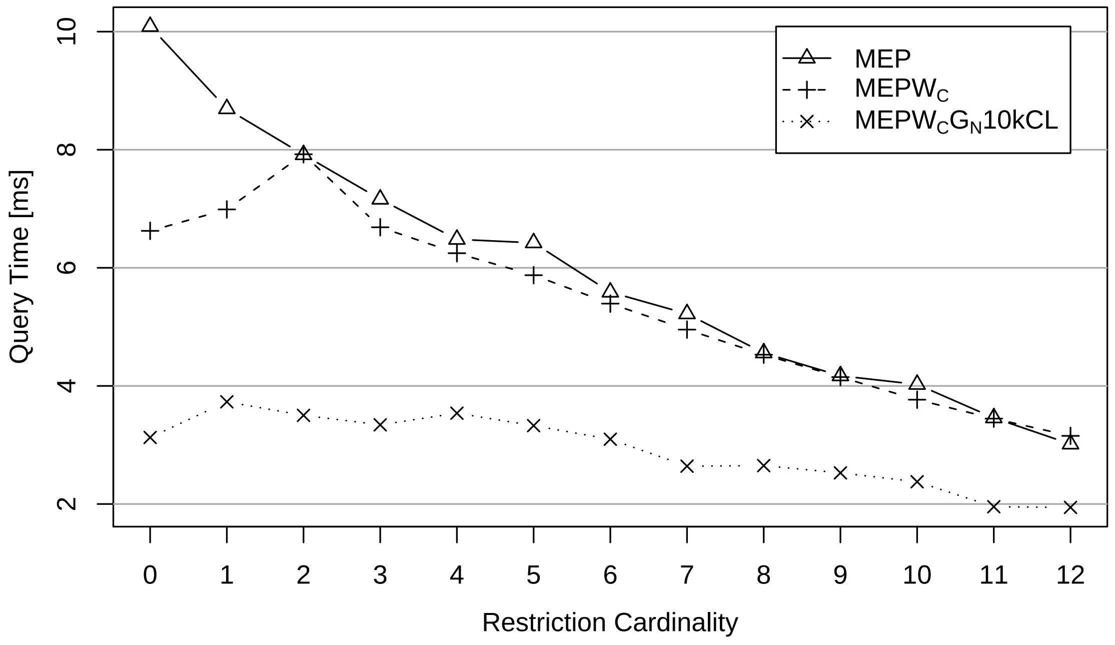


Figure 4.8: Experiments on the European graph comparing the query times of the MEP, MEPW$_{\mathrm{C}}$, and MEPW$_{\mathrm{C}}$G$_N$10kCL configurations across different restriction cardinalities.

each possible cardinality (0-18), we test the average query times of 10,000 source-target pairs, where each query is based on a uniform random set of query constraint parameters of the specified cardinality. We provide query times for three different algorithms in Figure 4.7. We see that without witness restrictions (MEP), the query time decreases with increasing restriction cardinality, as we need to relax fewer edges. As expected, witness restrictions improve the performance of less restricted queries with small restriction cardinality up to 30%. For more restricted queries, MEP is slightly faster, but the overall performance is better for MEPW$_{\mathrm{C}}$. Witness restrictions decrease the factor between unrestricted and restricted queries from 3.1 to 2.0. For MEPW$_{\mathrm{C}}$G$_N$10kCL, our algorithm with the best query times, this factor is reduced even further to 1.4, providing very good performance independent of the restriction cardinality.

On the European graph, we observe similar results in Figure 4.8, where we have only twelve different restrictions. The absolute query times are larger, as the network is bigger. Witness restrictions improve the time of unrestricted queries by 34%.

### 4.6.5 Comparison to Static Contraction Hierarchies

In Table 4.4 we evaluate the performance of static (inflexible) contraction hierarchies for unrestricted queries ignoring any restrictions (IGNORE), and fully restricted queries having any restricted edge removed from the graph (REMOVE). We see that the preprocessing is much faster as it does not need to consider restrictions. The preprocessing for IGNORE STATIC is 25 times faster than FLEXIBLE on North America, but we can still consider our FLEXIBLE preprocessing as efficient, as we consider $2^{16} \cdot 29 \cdot 57 = 108,331,008$ different choices of query constraint parameters, and on average, there are 29.4 different shortest paths over choices of query constraint parameters (Table 4.5). Also the query times of IGNORE are 11 times faster than FLEXIBLE, as the node order is tailored to the restriction set and not used for all possible restrictions. To evaluate this further, we perform the contraction with the node order computed by our flexible algorithm. The resulting query time is now only 4 times faster than FLEXIBLE, although the number of settled nodes and relaxed edges is almost the same. A big advantage of our flexible algorithm is its space-efficiency: its space consumption increases by less than 50%, mainly due to the larger edge data structure storing thresholds. This larger edge data structure also affects the cache-efficiency of our query algorithm, thus explaining some of its slowdown. For fully restricted queries (REMOVE), we see that the gap in the query time is much smaller, only a factor of 5. This is expected, as our flexible query algorithm can skip a lot of edges for such heavily-restricted queries. Also, it seems that the flexible node order is somewhat ‘closer’ to the REMOVE node

Table 4.4: Static contraction results based on ignoring edge restrictions (IGNORE) and removing edges with restrictions (REMOVE), compared to our flexible algorithm ($\mathrm{MEPW_C}$). We also give query performance in the case where we use the flexible node order (FLEX). In this case we report the flexible node ordering time (for MEP, this is the same node order as $\mathrm{MEPW_C}$ but does not compute witness restrictions) + the static contraction time. The performances are compared to our flexible algorithm (FLEXIBLE).

| **Graph** | **Query Constraint Parameters** | **Algorithm** | **Preprocessing** Time [H:M] | Space [B/node] | **Queries** Time [ms] | Settled Nodes | Stalled Nodes | Relaxed Edges |
|---|---|---|---|---|---|---|---|---|
| North America | IGNORE | STATIC | 0:17 | 22 | 0.41 | 795 | 382 | 3,207 |
| | | STATIC (FLEX) | 6:56 + 0:04 | 22 | 1.07 | 1,030 | 550 | 9,539 |
| | | FLEXIBLE | 7:11 | 32 | 4.34 | 1,068 | 588 | 9,765 |
| | REMOVE | STATIC | 0:11 | 19 | 0.42 | 727 | 344 | 3,361 |
| | | STATIC (FLEX) | 6:56 + 0:03 | 19 | 0.53 | 641 | 353 | 3,997 |
| | | FLEXIBLE | 7:11 | 32 | 2.20 | 641 | 353 | 3,997 |
| Europe | IGNORE | STATIC | 0:28 | 21 | 0.35 | 659 | 336 | 2,346 |
| | | STATIC (FLEX) | 17:27 + 0:07 | 21 | 1.79 | 1,370 | 697 | 16,475 |
| | | FLEXIBLE | 18:10 | 30 | 6.01 | 1,419 | 745 | 16,786 |
| | REMOVE | STATIC | 0:21 | 17 | 0.82 | 1,062 | 575 | 6,014 |
| | | STATIC (FLEX) | 17:27 + 0:04 | 17 | 0.94 | 819 | 456 | 7,008 |
| | | FLEXIBLE | 18:10 | 30 | 3.12 | 819 | 456 | 7,007 |

order, as using it only slightly increases the query time from 0.42 ms to 0.53 ms.

For the European graph, the gap between the static and our flexible algorithm is a bit larger. The preprocessing is now 39 times slower, although there are only 15.6 different shortest paths on average (Table 4.5). The graph seems to significantly change with restricted edges, indicated by the much higher number of edges carrying restrictions, c.f. Table 4.1. One reason is possibly that, in Europe, many countries demand toll on all their highways, whereas in North America only few highways require toll, mostly on the east coast of the United States. The results of [105] suggest that restricting these

Table 4.5: Average and maximum number of unique shortest paths for individual s-t pairs in the North American and European graphs over all possible query constraint parameters (when tested on 1000 s-t pairs).

| **Graph** | Average Path Count | Max Path Count |
|---|---|---|
| North America | 29.4 | 338 |
| Europe | 15.6 | 135 |

Table 4.6: Flexible contraction with restrictions, based on 2 static node orderings (from above): IGNORE and REMOVE. The preprocessing time is split into the static node ordering time + the flexible contraction time. We compare it to the flexible algorithm MEPW$_C$. Results are based on averages of fully restricted and unrestricted queries.

| | | **Preprocessing** | | **Queries** | | | |
|---|---|---|---|---|---|---|---|
| | | Time | Space | Time | Settled | Stalled | Relaxed |
| **Graph** | **Node Order** | [H:M] | [B/node] | [ms] | Nodes | Nodes | Edges |
| North America | FLEXIBLE | 7:11 | 32 | 3.27 | 855 | 470 | 6,881 |
| | IGNORE | 0:17 + 4:04 | 48 | 5.84 | 977 | 544 | 6,780 |
| | REMOVE | 0:11 + 48:00+ | - | - | - | - | - |
| Europe | FLEXIBLE | 18:10 | 30 | 4.52 | 1,119 | 601 | 11,897 |
| | IGNORE | 0:28 + 39:21 | 45 | 22.09 | 1,426 | 930 | 20,215 |
| | REMOVE | 0:21 + 48:00+ | - | - | - | - | - |

highways highly affects the efficiency of our algorithms. Therefore, the balancing act of creating a node order suiting all restrictions is much harder, affecting the preprocessing and query time. Still, our algorithm is reasonably fast for web services, and more importantly, requires only about the space of two static contraction hierarchies.

### 4.6.6 Static Node Ordering

We are further interested in using statically computed node orders for our flexible contraction. In practice, the node ordering is much slower than contraction given a node order, as computing and updating the node priorities takes the majority of the time. Therefore, we hope that by performing a static node ordering, we can significantly decrease the overall precomputation time. The results are summarized in Table 4.6. We used the node orders IGNORE and REMOVE computed in the previous section. The IGNORE node order improves the precomputation time, as it only takes about 4 hours for the contraction plus an additional 17 minutes to compute the static node order, instead of 7:11. However, the query time increases by 79%. We analyzed this in more detail and found out that an unrestricted query takes now 2.53 ms and a fully restricted query takes 9.14 ms. So the static node order is only good for unrestricted queries, and is even faster than the flexible one for such queries. But restricted queries become slow, and the space consumption also increases by 60%, as a lot more shortcuts are necessary, since this static node order does not consider restrictions. We strongly see the blindness of a static node order towards unconsidered restrictions for the REMOVE node order. There, our contraction did not finish within two days. Therefore, we recommend the usage of static node orders only for practitioners with profound knowledge of the algorithm and of their most commonly-expected query constraint scenarios, as, otherwise, a lot of things can go wrong with the use of static orders.

## 4.7 Conclusion

In this chapter, we extended the ideas from [53] and [94] to engineer an efficient algorithm that takes different edge restrictions into account. For continental sized road networks, the preprocessing takes several hours, with query times of only a few milliseconds. The speed-up over Dijkstra's algorithm is more than three orders of magnitude. We achieve this by incorporating goal-direction into our algorithm, and by using witness restrictions, allowing to skip shortcuts that are only required for more restricted queries.

# Part III

# Route Planning with Detour Constraints

# Chapter 5

# Generalized Shortest Paths

## 5.1 Introduction

Imagine you are traveling in a new city. On the way to your destination, you wish to stop by a gas station to refuel your vehicle, stop at a post office to drop off a postcard, and stop by one of your favorite fast-food restaurant chains to pick up dinner. Naturally, you wish to do this with the least amount of overall detour on the way to your final destination.

Furthermore, it is reasonable to assume that you do not particularly care exactly *which* gas station, post office, or favorite fast-food chain location you visit, since each will ultimately provide the same type of intended service[1]. Since each location type is primarily accessible via the road network, the goal then is to find a path through the network that visits one location from each of these location types, in the least amount of overall time (or distance, or monetary cost, etc.). Such scenarios are common for personal navigation needs.

Additional applications of this problem type exist in the logistics industry

[1]Note that the location types are user-defined and may therefore be made more specific, if desired; e.g., only consider gas stations of a certain brand.

as well. For example, for long-haul trucking, the route may require multiple days of traveling long distances, necessitating the recurrent need for refueling, lodging, eating, etc. throughout the trip. Selecting the optimal location(s) amongst all of these options to minimize travel time and cost is an important part of the route planning process.

This basic problem is more commonly known as the Generalized Traveling Salesman Path Problem (GTSPP) [96], and it is known to be NP-hard, as the solution must determine not only which location from each location type to visit, but also the optimal order in which to visit them. However, for many scenarios, the order in which each location type is visited during the trip can often be very important (or even required). For example, in the previous personal navigation scenario, you might wish to refuel first if your vehicle is low on gas, and to get dinner last so that it is still fresh when you arrive at your destination. Additional scenarios from logistics in which the visit order of each location type is crucial include deliveries which require specialized equipment, such as moving equipment or regulated containers (e.g., for storing temperature-sensitive materials). In many cases, the vehicle must first pick up the equipment along the way at any one of several pickup (or rental) locations, then pick up the package and make the delivery. Only after the delivery can the equipment then be returned at any one of the other pickup (or rental) locations. Problems such as these, for which the visit order of each location type is fixed, are called here Generalized Shortest Path (GSP) queries.

In the remainder of the chapter, we propose an efficient algorithmic approach for solving GSP queries in real-world road networks. In Section 5.2, we formalize the GSP query, and give a discussion of related work and our contributions. In Section 5.3, we show how GSP queries may additionally be used to find approximate solutions for the (NP-hard) GTSPP. Section 5.4 details our approach towards engineering a scalable algorithm for solving large-scale GSP queries. Section 5.5 presents our experimental results of this approach for GSP queries on the continent-wide road network of North America. Section 5.6 concludes the chapter.

## 5.2 Generalized Shortest Path Queries

Generalized Shortest Paths (GSP) are shortest paths that must visit at least one node from each of a set of node categories, in a specified order. We define this concept more formally as follows.

**Definition 12** *(Graph) Let $G = (V, E, w)$ be a directed graph, where $V$ is the set of vertices in $G$, $E \subseteq V \times V$ is the set of edges in $G$, and $w : E \rightarrow \mathbb{R}_+$ is a function mapping edges in $G$ to a positive, real-valued weight. Let $|V| = n$ and $|E| = m$.*

**Definition 13** *(Path) Let $P_{s,t} = \langle v_1, v_2, \ldots, v_q \rangle$ be any path in $G$ from some vertex $s = v_1 \in V$ to some vertex $t = v_q \in V$, such that, for $1 \leq i < q$, $(v_i, v_{i+1}) \in E$. Let $w(P_{s,t}) = \sum_{1 \leq i < q} w(v_i, v_{i+1})$ be the weight, or cost, of $P_{s,t}$.*

**Definition 14** *(Shortest Path) Given any $s, t \in V$, a least-cost (or "shortest") path is a path $P'_{s,t}$ in $G$ such that, $\forall$ $P_{s,t}$ in $G$, $w(P'_{s,t}) \leq w(P_{s,t})$. The shortest path cost, $w(P'_{s,t})$, is most often formally referred to as $d(s, t)$.*

For each GSP problem instance, we represent the set of possible locations for each requested category, as well as their intended sequence of traversal for the query, through a *category sequence*, as defined below.

**Definition 15** *(Category Sequence) A category sequence defines a fixed sequence of node sets $C = \langle C_1, C_2, \ldots, C_k \rangle$, where, for $1 \leq i \leq k$, $C_i = \{c_{i,1}, c_{i,2}, \ldots, c_{i,|C_i|}\} \subseteq V$ represents a unique category of available locations. The sequence determines the order in which each category must be visited.*

**Definition 16** *(GSP Query) A GSP query may be represented as a three-tuple $\langle s, t, C\rangle$, such that $s, t \in V$ and $C = \langle C_1, C_2, \ldots, C_k\rangle$ represents a category sequence. The query is said to have category count $k$ and category density $g = \max\limits_{1\le i\le k}\{|C_i|\}$. See Figure 5.1 for an example query.*

**Definition 17** *(Satisfying Path) For any category sequence, $C$, a path, $P_{s,t}$, is said to satisfy the category sequence if there exists a subsequence of nodes $\langle v_{i_1}, v_{i_2}, \ldots, v_{i_k}\rangle$ from $P_{s,t}$, such that $1 \le i_1 \le i_2 \le \ldots \le i_k \le q$ and for $1 \le j \le k$, $v_{i_j} \in C_j$. This is formally written as $P_{s,t} \models C$.*

**Definition 18** *(GSP Solution) Given any GSP query $\langle s, t, C\rangle$, a GSP is a path $P'_{s,t}$ in $G$ such that $P'_{s,t} \models C$ and $\forall\ P_{s,t}$ in $G$ where $P_{s,t} \models C$, $w(P'_{s,t}) \le w(P_{s,t})$. This optimal solution path is more formally referred to as $P^C_{s,t}$.*

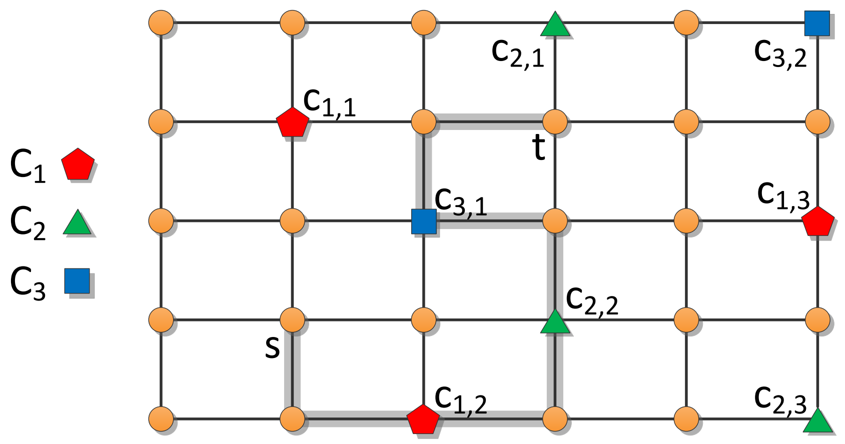


Figure 5.1: GSP Example Query $\langle s, t, \langle C_1, C_2, C_3\rangle\rangle$. The edges have unit-cost weight. An optimal path is shaded in grey.

A variant of GSP query, which has important theoretical implications (discussed in Section 5.3), is the following.

**Definition 19** *(RTO-GSP) Return-to-Origin (RTO) GSP queries are a special variant of GSP in which the destination is the same as the origin; i.e., $s = t$.*

For simplicity in the remainder of this discussion, we shall focus solely on retrieving only the optimal solution cost, $w(P^C_{s,t})$, for each query, rather than constructing the resulting path. Maintaining parent pointers for subsequent path retrieval is a straightforward extension to this model.

To further simplify our later formulations, given a category sequence $C = \langle C_1, C_2, \ldots, C_k \rangle$, we shall also assume two implicit "dummy" categories, $C_0 = \{s\}$ and $C_{k+1} = \{t\}$, such that $C = \langle C_0, C_1, \ldots, C_k, C_{k+1} \rangle$.

### 5.2.1 Related Work

GSP queries have been established using various formulations in prior research. The first work to address this type of query was in [106], in which two specific algorithms were proposed for this query type: LORD and PNE. The LORD algorithm was designed primarily to work with point sets in Euclidean space. The PNE algorithm was proposed for work within graphs, such as road networks. However, the PNE algorithm requires issuing multiple nearest-neighbor queries in the graph, and this approach can demonstrably result in overlapping search effort throughout the graph.

The GSP query may also be seen as a special class of *language constrained shortest path* queries [18], in which the solution paths are constrained by formal language requirements on labels associated with the edges/nodes of the graph. Such constraints require a valid solution path to contain edge/node labels in a fixed pattern or sequence. In this context, the GSP query requires that the labels apply to the nodes instead of the edges, and the node labels of the paths must match a regular expression of the form $(C_0 * C_1 * ... * C_k * C_{k+1})$, where $*$ represents the Kleene closure of the wildcard, matching any sequence of possible node labels. An extension to the classical Dijkstra shortest path algorithm [39] has been given in [18] for solving similar regular expression constrained queries. The algorithm relies on searching a product graph constructed from the original road network graph and from the finite automaton graph used to represent the regular language constraint. As this approach requires only a single graph search, it avoids the unnecessary search overlap of the PNE algorithm. However, this approach may still require searching many (or even all) nodes in the product graph.

The first (and, until now, only) pre-processing technique designed to support GSP queries was proposed in [107], in which the pre-processing constructs a series of *additively weighted voronoi diagrams* (AWVD) for a given category sequence. While efficient, this approach requires a priori knowledge of the set(s) of possible node categories and their fixed sequence(s), thus limiting its applicability for real-world GSP query scenarios. For example, no two travelers are likely to consider the exact same (sub)set of restaurants (or grocery stores, etc.) as desirable (e.g., based on brand preferences), nor are they likely to consider them in the exact same fixed sequence amongst other categories for each distinct trip. In order for the AWVD approach to account for all possible subsets and all possible sequences of category locations would require $\Omega(2^{|V|})$ preprocessing time and storage overhead, which is intractable in practice. Therefore, a more flexible approach is needed.

Pre-processing techniques have recently shown great success in speeding up various other shortest path related queries as well. The most prominent of these techniques to-date is known as the Contraction Hierarchies (CH) technique [53].

CH pre-processing establishes an ordering of the nodes in the graph, $\phi : V \to \{1, \ldots, |V|\}$, and then *contracts* the nodes in this order. To contract a node, $v$, means to remove it from the graph by adding new edges, called *shortcut* edges, as necessary, to preserve shortest paths in the remaining subgraph. Specifically, for each pair of incoming and outgoing edges, $(u, v)$ and $(v, x)$, respectively, if the path $\langle u, v, x\rangle$ is a unique shortest path, then a new shortcut edge $(u, x)$, with weight $w(u, v) + w(v, x)$, is added to bypass $v$ in the remaining subgraph. The pre-processing results in a new graph $G' = (V, E \cup E', w)$, where $E'$ represents the shortcut edges.

*Point-to-Point (PTP)* shortest path queries from $s \in V$ to $t \in V$ may then be carried out in the resulting graph as follows. A forward Dijkstra [39] search from $s$ in the graph $G^{\uparrow} = (V, E^{\uparrow})$, where $E^{\uparrow} = \{(u, v) \in E \cup E' \mid \phi(u) < \phi(v)\}$ is run simultaneously with a backward Dijkstra search from $t$ in the graph $G^{\downarrow} = (V, E^{\downarrow})$, where $E^{\downarrow} = \{(u, v) \in E \cup E' \mid \phi(u) > \phi(v)\}$. A tentative shortest path cost is maintained and is updated only when the two search frontiers meet to form a shorter path. The search in a given direction may be stopped once the minimum key for the priority queue in that direction exceeds the cost of the best path seen thus far.

In addition to PTP queries, since our problem definition can have multiple nodes per category, we further consider *Many-to-Many (MTM)* shortest path query scenarios involving multiple sources, $S \subseteq V$, and multiple targets, $T \subseteq V$. An efficient query technique for MTM queries using CH has also been proposed in [53]. Rather than carrying out $|S| \cdot |T|$ unique PTP CH queries, the query first computes $|T|$ backward Dijkstra searches in $G^{\downarrow}$, one for each target $t \in T$. For each node, $v$, reached by the backward search from some $t \in T$, the value $d_t^{\downarrow}(v)$, representing the cost of the best path found from $v$ to $t$, is stored along with the identifier $t$ in a bucket at $v$. Then the query performs $|S|$ forward Dijkstra searches in $G^{\uparrow}$, one for each source $s \in S$. For each node, $v$, reached by the forward search from some $s \in S$, the value $d_s^{\uparrow}(v)$, representing the cost of the best path found from $s$ to $v$, is also maintained. For each pair $s \in S$ and $t \in T$, the cost of the shortest $s$-$t$ path may then be computed as $d(s,t) = \underset{\forall v \in V}{min}\{d_s^{\uparrow}(v) + d_t^{\downarrow}(v)\}$[2]. This process requires only $|S| + |T|$ CH graph searches, plus the overhead to intersect the forward and backward searches.

Note that $d_s^{\uparrow}(v) \geq d(s,v)$ and $d_t^{\downarrow}(v) \geq d(v,t)$, in general, as these costs are only guaranteed to be equivalent for the highest ranking node on a shortest $s$-$t$ path (however, this property is still sufficient for the correctness of CH queries).

**Lemma 20** *For the highest-ranking node, $v$, on any shortest $s$-$t$ path, $d_s^{\uparrow}(v) = d(s,v)$ and $d_t^{\downarrow}(v) = d(v,t)$* [53].

As mentioned previously, a closely related problem to the GSP is the Generalized Traveling Salesman Path Problem (GTSPP). Unlike GSP, in which the category sequence is fixed a priori, GTSPP solutions must additionally optimize the sequence of the selected category locations. As the well-known Traveling Salesman Path Problem (TSPP), with fixed source $s$ and destination $t$, is merely a special instance of GTSPP in which each node defines its own unique location category, then it is clear that GTSPP is also NP-hard. As we shall demonstrate later in this chapter, our polynomial-time GSP solutions can also serve as a good approximation for the more-complex GTSPP queries as well. Additional approximation algorithms for GTSPP on undirected graphs are presented in [79]. Among these, the Minimum-Distance (MD) algorithm was proposed, which selects the node, $z_i$, from each category, $C_i$, that satisfies the following:

$$z_i = \underset{\forall c_{i,j} \in C_i}{argmin}\{d(s, c_{i,j}) + d(c_{i,j}, t)\} \tag{5.1}$$

The authors prove (in the full version of [79]) that the path formed by *any* permutation of these nodes is a $(k+1)$-approximation for GTSPP on undirected graphs.

[2]It is not required to iterate over every node, $v \in V$, in practice; only those (few) nodes reached by both search directions are needed. See [53].

### 5.2.2 Our Contributions

The following outlines our contributions towards establishing an efficient algorithm for supporting GSP queries:

1. To the best of our knowledge, this is the first work to formalize a dynamic programming formulation for this particular constrained shortest path query variant. Furthermore, this is also the first work to utilize the concepts of the CH approach for this query variant. Using concepts from this model, and building from our proposed dynamic programming formulation, we engineer a very fast and scalable GSP query algorithm.

2. Unlike the AWVD pre-processing approach from [107], which requires a priori knowledge of the set(s) of categories as well as their expected sequence(s), our associated pre-processing approach requires neither. Our supported queries are fully-dynamic, in that the client issuing the query can construct any possible set of node categories and specify any possible sequence of such categories. Such flexibility is considered necessary for practical, real-world applications.

3. We demonstrate that, regardless of the chosen category sequence, GSP queries closely approximate GTSPP queries on the same set of categories to within a factor of $O(k)$, for most well-formed graphs. This suggests that we may also apply our proposed (polynomial-time) algorithm for solving GSP queries to find $O(k)$-approximate solutions for the NP-hard GTSPP queries.

4. We present a detailed experimental analysis of our proposed solution on GSP and RTO-GSP queries applied to the continent-wide road network of North America (with over 50 million edges). We further examine the results of our solution when

solved on queries with varying category counts and densities. As a baseline, we compare our solution against an adaptation of the regular expression constrained Dijkstra's algorithm variant presented in [18]. Our proposed algorithm results in speedups of up to several orders of magnitude over the Dijkstra variant, depending on the overall number and densities of the categories.

## 5.3 GSP Approximates GTSPP

Given any GSP query $\langle s, t, C\rangle$, let $\pi$ be any permutation on the (original) input sequence $C = \langle C_1, C_2, \ldots, C_k\rangle$, such that $C_\pi = \langle C_{\pi_1}, C_{\pi_2}, \ldots, C_{\pi_k}\rangle$. By definition, solving the GTSPP requires determining the best overall permutation, $\pi'$, such that, for all permutations, $\pi$, $w(P_{s,t}^{C_{\pi'}}) \leq w(P_{s,t}^{C_\pi})$. $P_{s,t}^{C_{\pi'}}$ thus represents an optimal GTSPP solution on $C$.

**Theorem 21** *For any GSP query $\langle s, t, C\rangle$ on an undirected (symmetric) graph, $w(P_{s,t}^C) \leq (k+1) \cdot w(P_{s,t}^{C_{\pi'}})$.*

**Proof.** The proof follows implicitly from the results of the MD approximation algorithm proposed in [79]. Since the path formed by any permutation $\pi$ of the nodes selected by the MD algorithm is a $(k+1)$-approximation for GTSPP, the MD algorithm must produce a $(k+1)$-approximation when using our given input sequence $\langle C_1, C_2, \ldots, C_k\rangle$. It follows then that $w(P_{s,t}^C)$ is also a $(k+1)$-approximation, as it is an optimal solution for this sequence. ■

Unfortunately, however, the proof of approximation of the MD algorithm from [79] does not hold under asymmetry (i.e., cases where $d(u, v) \neq d(v, u)$). Nevertheless, we can prove the following bound for directed, asymmetric graphs.

**Theorem 22** *For any GSP query $\langle s, t, C \rangle$ on a directed (asymmetric) graph, $w(P_{s,t}^{C}) \leq k \cdot w(P_{s,t}^{C_{\pi}'}) + (k-1) \cdot d(t,s)$, and this bound is tight.*

**Proof.** Consider the set of nodes $\{z_1, z_2, \ldots, z_k\}$ selected by the MD algorithm (using equation (5.1)) for the original input sequence $C$. Since $w(P_{s,t}^{C})$ is optimal for this sequence, we have the following inequality:

$$w(P_{s,t}^{C}) \leq d(s, z_1) + \sum_{i=1}^{k-1} d(z_i, z_{i+1}) + d(z_k, t) \tag{5.2}$$

$$\leq d(s, z_1) + \sum_{i=1}^{k-1} (d(z_i, t) + d(t, s) + d(s, z_{i+1}))$$

$$+ d(z_k, t) \tag{5.3}$$

$$= \sum_{i=1}^{k} (d(s, z_i) + d(z_i, t)) + \sum_{i=1}^{k-1} d(t, s) \tag{5.4}$$

$$\leq k \cdot w(P_{s,t}^{C_{\pi}'}) + (k-1) \cdot d(t, s) \tag{5.5}$$

The right-hand side of inequality (5.2) represents the cost of the MD solution for this sequence. Inequality (5.3) (and by equivalence, (5.4)) follows from the triangle inequality. Inequality (5.5) holds by the MD selection logic for each node $z_i$, since, for $1 \leq i \leq k$, $(d(s, z_i) + d(z_i, t)) \leq (d(s, z_i') + d(z_i', t)) \leq w(P_{s,t}^{C_{\pi}'})$, where $z_i'$ is the node chosen from category $C_i$ in the optimal GTSPP path $P_{s,t}^{C_{\pi}'}$ (this inequality holds regardless of the chosen permutation of categories).

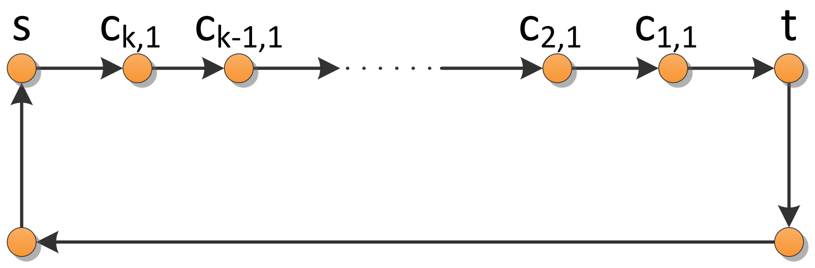


Figure 5.2: Directed Cycle Graph

To prove that this bound is tight (in the worst case), consider the simple directed cycle graph in Figure 5.2. For this graph, suppose we have an input GSP query sequence $C = \langle C_1, C_2, \ldots, C_k \rangle$, but the optimal sequence for the GTSPP query on the same categories is $C_{\pi'} = \langle C_k, C_{k-1}, \ldots, C_1 \rangle$ (i.e., the reverse of the input sequence). Since the GTSPP solution path is the shortest path between $s$ and $t$ and the GSP solution path must loop back on itself $(k-1)$ times to achieve the desired sequence, this gives us $w(P^C_{s,t}) = k \cdot d(s,t) + (k-1) \cdot d(t,s) = k \cdot w(P^{C_{\pi'}}_{s,t}) + (k-1) \cdot d(t,s)$. ■

This suggests that the complexity of approximation in the asymmetric graph scenario can depend quite heavily on the degree of asymmetry found within the graph of interest. For example, if the graph is asymmetric, but with bounded asymmetry (i.e., for all $s, t \in V$, $d(t,s) \leq c \cdot d(s,t)$ for some constant $c$), then the above approximation is still an $O(k)$-approximation (because $d(t,s) \leq c \cdot d(s,t) \leq c \cdot w(P^{C_{\pi'}}_{s,t})$). We note that most road networks tend to have bounded-asymmetry for sufficiently long-distance paths as well, but this bound degrades for closely-located node pairs (e.g., due to one-way streets). However, inherent in this theorem is also the implication that, the closer the distance from target to source in any asymmetric graph, the closer the solution is to being a truly $O(k)$-approximate solution. This leads directly to the following corollary.

**Corollary 23** *For any RTO-GSP query $\langle s, s, C \rangle$ on a directed (asymmetric) graph, $w(P^C_{s,s}) \leq k \cdot w(P^{C_{\pi'}}_{s,s})$.*

Note that these proofs do not suggest that we cannot further improve upon these bounds by somehow more intelligently selecting our chosen sequence. Rather, this suggests only that these are the best possible bounds we can hope to achieve, given any arbitrary input category sequence. A more detailed look into designing algorithms for better overall approximations are beyond the scope of this chapter.

We have shown that, by fixing any arbitrary sequence of categories and solving a GSP instance on these categories, this generally provides a very good approximation, typically within a factor of $O(k)$, of the GTSPP solution for the same set of categories. Therefore, any algorithm designed for optimally solving GSP queries (such as the one presented in the following sections) may additionally be utilized to approximately solve the NP-hard GTSPP queries. In our experimental results (Section 5.5), we further examine how well our GSP solutions approximate GTSPP solutions in practice.

## 5.4 Engineering a Scalable GSP Algorithm

### 5.4.1 Dynamic Programming Formulation

In this section, we formulate a new dynamic programming solution to the GSP problem. We construct a real-valued matrix, $X$, with $(k+2)$ rows[3] and $g$ columns[4]. We populate the matrix as follows:

$$X[i,j] = \begin{cases} 0 & \text{if } i = 0 \\ \\ \underset{1 \le \ell \le |C_{i-1}|}{min} \{X[i-1,\ell] + d(c_{i-1,\ell}, c_{i,j})\} & \text{if } i > 0 \end{cases}$$

**Lemma 24** *The value $X[i,j]$ represents the optimal solution cost for the GSP query $\langle s, c_{i,j}, \langle C_0, C_1, \ldots, C_i \rangle \rangle$.*

**Proof.** We prove this by induction on the category sequence $0 \le i \le k+1$. For the base case, where $i = 0$, then this claim is trivially true, since $\langle s, c_{0,1}, \langle C_0 \rangle \rangle = \langle s, s, \langle \{s\} \rangle \rangle$, and $d(s,s) = 0$. For the induction step, when $i > 0$, our induction hypothesis assumes that this claim holds true for all values $X[i-1, \bullet]$. Since each $c_{i,j}$ is itself a member of category $C_i$, then it suffices to find the least-cost path which visits categories $\langle C_0, C_1, \ldots, C_{i-1} \rangle$

[3]The two additional rows account for the dummy categories $C_0 = \{s\}$ and $C_{k+1} = \{t\}$.

[4]W.l.o.g., we may assume $|C_i| = g$, for all $1 \le i \le k$.

and ends at $c_{i,j}$. It follows from our induction hypothesis that this is exactly the value computed by $\underset{1 \leq \ell \leq |C_{i-1}|}{min} \{X[i-1,\ell] + d(c_{i-1,\ell}, c_{i,j})\}$. ■

**Corollary 25** *The value $X[k+1,0]$ represents the optimal solution cost for the query $\langle s,t,C\rangle$; i.e., $X[k+1,0] = w(P_{s,t}^{C})$.*

**Quadratic Impact of Category Density.** Constructing the above matrix requires $O(kg^2)$ steps, not accounting for the runtime of establishing the shortest path costs between pairs of nodes from adjacent categories (this will be addressed in the following subsections). In most real-world navigation scenarios, $k \ll g$ is expected to be true. Therefore, the primary inefficiency in this formulation comes from the quadratic impact of the density, $g$, in deriving this matrix. The goal of the following optimizations is then to effectively minimize the overall impact of category density on the efficiency of constructing the matrix as well as its effect on the shortest path query overhead.

**Incorporating Contraction Hierarchies.** In order to minimize the overhead from the construction of the $X$ matrix, we must also minimize the overhead for computing the shortest path costs, $d(c_{i-1,\ell}, c_{i,j})$, between each pair of nodes in adjacent categories. One possibility would be to preprocess all-pairs-shortest-path costs. However, this is impractical as it requires $O(|V|^2)$ space overhead. A logical next consideration would then be to utilize the very fast point-to-point (PTP) CH shortest path queries. Unfortunately, however, even this approach has its limitations, as computing individual pairwise shortest paths still requires $O(kg^2)$ individual CH queries overall. Instead we may rely upon the previously-established many-to-many (MTM) CH shortest path query approach, whereby we compute MTM costs from source nodes, $C_i$, to target nodes, $C_{i+1}$, for each adjacent category pair. Using the MTM CH search approach then requires us to perform only $2 \cdot (kg+1)$ unique CH graph searches (i.e., $g$ forward searches per category, $g$ backward searches per category, and 1 forward and 1 backward search for $s$ and $t$, respectively). This straightforward approach already helps to minimize the strong impact of category density on our problem, resulting in $O(kg)$ CH searches overall. However, as we will demonstrate later, this approach also has its scalability and performance limits.

### 5.4.2 Optimizing for the Problem Structure

In order to engineer an alternative approach for more efficiently solving this problem, we first look at the structure of the initial solution. We start by examining the nature of the computation required to construct our proposed $X$ matrix entries using the MTM CH algorithm previously suggested. For $i > 0$:

$$\begin{aligned} X[i,j] &= \min_{1 \le \ell \le |C_{i-1}|} \{X[i-1,\ell] + d(c_{i-1,\ell}, c_{i,j})\} \\ &= \min_{1 \le \ell \le |C_{i-1}|} \{X[i-1,\ell] + \underbrace{\min_{\forall v \in V} \{d^{\uparrow}_{c_{i-1,\ell}}(v) + d^{\downarrow}_{c_{i,j}}(v)\}\}}_{\text{CH Formulation}} \end{aligned}$$

As is highlighted in the horizontally-bracketed area of the formulation above, the MTM CH algorithm requires to perform exactly one forward search from each origin node and one backward search from each destination node. The cost between any pair of origin/destination nodes is then simply taken as the minimum sum of forward and backward search costs over all (reached) nodes.

Using this new CH-specific matrix formulation, we may re-arrange the terms of the equation to come up with an alternative, yet equivalent, formulation as follows:

$$\begin{aligned} X[i,j] &= \min_{1 \le \ell \le |C_{i-1}|} \{X[i-1,\ell] + \min_{\forall v \in V} \{d^{\uparrow}_{c_{i-1,\ell}}(v) + d^{\downarrow}_{c_{i,j}}(v)\}\} \\ &= \min_{\forall v \in V} \{ \min_{1 \le \ell \le |C_{i-1}|} \{X[i-1,\ell] + d^{\uparrow}_{c_{i-1,\ell}}(v)\} + d^{\downarrow}_{c_{i,j}}(v)\} \\ &= \min_{\forall v \in V} \{\rho^{\uparrow}_{i-1}(v) + d^{\downarrow}_{c_{i,j}}(v)\} \end{aligned}$$

where, for all $0 \leq i \leq k$:

$$\rho_i^{\uparrow}(v) = \underset{1 \leq \ell \leq |C_i|}{min} \{X[i,\ell] + d_{c_{i,\ell}}^{\uparrow}(v)\}$$

Next, we establish an efficient method for computing the values of $\rho_i^{\uparrow}$. We show how to construct these values using only a single forward CH search per category, instead of the previous $g$ unique forward CH searches per category.

To calculate the values $\rho_i^{\uparrow}$ for category $C_i$, we first create a temporary super-source node, $s_i'$, which we add to the CH search graph, along with temporary edges $(s_i', c_{i,j})$ of cost $X[i,j]$, for all $c_{i,j} \in C_i$. We then perform a single forward Dijkstra search in the (sparse) CH search graph, starting from the super-source node $s_i'$.

**Lemma 26** *For any node, $v$, reached during the forward search from $s_i'$, $d_{s_i'}^{\uparrow}(v) = \rho_i^{\uparrow}(v)$.*

**Proof.** By construction, for each node $c_{i,j} \in C_i$ where there exists a path from $c_{i,j}$ to $v$ in $G^{\uparrow}$, there must also exist a path from node $s_i'$ to $v$ of cost $w(s_i', c_{i,j}) + d_{c_{i,j}}^{\uparrow}(v) = X[i,j] + d_{c_{i,j}}^{\uparrow}(v)$ in our (temporarily) modified $G^{\uparrow}$. Since the Dijkstra search will find the minimum over all of these paths from $s_i'$ to node $v$, this gives us:

$$d_{s_i'}^{\uparrow}(v) = \underset{1 \leq \ell \leq |C_i|}{min} \{X[i,\ell] + d_{c_{i,\ell}}^{\uparrow}(v)\} = \rho_i^{\uparrow}(v)$$

■

Note that we do not actually have to explicitly manipulate the structure of the graph with these temporary super-source nodes and related edges to achieve this computation. Instead, we simply first insert each node, $c_{i,j} \in C_i$, into the priority queue with initial cost equal to $X[i,j]$ before beginning our forward Dijkstra search. One can easily verify that this is implicitly equivalent to the previously-suggested graph manipulation, but is much faster in practice. The pseudocode for this optimized forward search algorithm is presented in Algorithm 7 (including an additional optimization in line 4, discussed later on in Section 5.4.4).

Using this first optimization to the MTM CH approach, the number of unique CH graph searches is reduced from $2 \cdot (kg + 1) = kg + kg + 2$ down to $k + kg + 2$ (due to only one forward search per category), effectively cutting the number of unique CH graph searches roughly in half (for $k \ll g$).

### 5.4.3 Optimizing for the Graph Structure

While it is tempting to try and apply a similar approach as before to reduce the number of backward searches from each category as well, the problem structure utilized by our previous optimization is unfortunately not quite the same for the backward CH graph searches. However, performing a unique backward search for each node in a given category still results in much redundant search effort. This is because much of the search space from each individual node is likely shared by many other nodes from that category. Therefore, we wish to further account for this potential redundancy and allow for only a single backward search approach, (somewhat) similar to our previous optimization. The pseudocode is presented in Algorithm 8 (including an additional optimization in line 7, discussed in Section 5.4.4).

As it turns out, there is a very nice property of CH search graphs that we may readily take advantage of for our purposes. Specifically, every CH search graph is acyclic, by definition (i.e., searches are only allowed to move upward in node rankings, and can thus never return back to any previously visited nodes). Using a straightforward variant of depth-first search (DFS; see Algorithm 9), we first establish a reverse topological ordering of the nodes belonging to the unioned backward search spaces for each node in the current category, $C_i$ (see Algorithm 8, lines 6-10). This requires only linear-time in total, as it avoids repetition from previously searched nodes in the same category (effectively, a single DFS). After establishing a reverse topological ordering of the unioned backward search space, we process each reached node, $v$, in reverse topological order, computing the value $\rho_i^{\downarrow}(v)$ as follows (see Algorithm 8, lines 11-20):

$$\rho_i^{\downarrow}(v) = min\{\rho_{i-1}^{\uparrow}(v), \underset{\substack{\forall(u,v)\in E\cup E': \\ \phi(u)>\phi(v)}}{min}\{\rho_i^{\downarrow}(u) + w(u,v)\}\} \tag{5.6}$$

**Lemma 27** *For any node, $v$, reached during the reverse-topological backward search, $\rho_i^{\downarrow}(v)$ represents the optimal cost for the query $\langle s, v, \langle C_0, C_1, ..., C_{i-1}\rangle\rangle$.*

**Proof.** Let $\{v_1, v_2, \ldots, v_z\}$ be the nodes reached by our backward DFS in reverse topological order. We prove this lemma by induction on the reverse topological ordering sequence $1 \leq j \leq z$. For the base case, where $j = 1$ (i.e., $v_j = v_1$ is the first node in this ordering), there cannot exist any higher-ranking nodes with a path to $v_1$ in our search graph, by definition. Evaluating equation (5.6) for node $v_1$ then gives us $\rho_i^{\downarrow}(v_1) = \rho_{i-1}^{\uparrow}(v_1) = d_{s'_{i-1}}^{\uparrow}(v_1) = \underset{1\leq\ell\leq|C_{i-1}|}{min}\{X[i-1,\ell] + d_{c_{i-1,\ell}}^{\uparrow}(v_1)\}$ by Lemma 26. As $v_1$ is the highest ranking node on the shortest path from the super-source $s'_{i-1}$ (introduced in the previous section), then it follows from Lemma 20 that $\rho_i^{\downarrow}(v_1) = d(s'_{i-1}, v_1) = \underset{1\leq\ell\leq|C_{i-1}|}{min}\{X[i-1,\ell] + d(c_{i-1,\ell}, v_1)\}$. Therefore, as $\rho_i^{\downarrow}(v_1)$ is the cost of the minimum cost path to $v_1$ which visits all categories $\langle C_0, C_1, ..., C_{i-1}\rangle$ in this order, then it is the optimal cost for query $\langle s, v_1, \langle C_0, C_1, ..., C_{i-1}\rangle\rangle$. For the induction step, where $j > 1$, our induction hypothesis assumes that this claim is correct for all nodes that come before $v_j$ in reverse topological order. If $v_j$ is the highest ranking node on the shortest path from $s'_{i-1}$, then its correctness again follows via the same logic as above. Therefore, assume $v_j$ is not the highest ranking node on the shortest path from $s'_{i-1}$, and let node $u_j$ be the node such that $\phi(u_j) > \phi(v_j)$ and $(u_j, v_j)$ is the last edge along the shortest path from $s'_{i-1}$ to $v_j$ (such a node must exist, by properties of the CH preprocessing). As we have already processed any higher ranking nodes leading to $v_j$ along this shortest path (based on the reverse topological ordering), then by our induction hypothesis, $\rho_i^{\downarrow}(u_j) = d(s'_{i-1}, u_j)$ is optimal for the category sequence $\langle C_0, C_1, \ldots, C_{i-1}\rangle$ and thus the evaluation of (the right half of) equation (5.6) for node $v_j$ will result in $\rho_i^{\downarrow}(v_j) = d(s'_{i-1}, v_j) = \underset{1 \le \ell \le |C_{i-1}|}{min} \{X[i-1, \ell] + d(c_{i-1,\ell}, v_j)\}$, proving our claim as before. ■

Since each $c_{i,j}$ is a member of $C_i$, this further implies that the value $\rho_i^{\downarrow}(c_{i,j})$ must also be optimal for the query $\langle s, c_{i,j}, \langle C_0, C_1, ..., C_i\rangle\rangle$. We therefore need only perform a post-processing step after the above relaxation steps to iterate over each node, $c_{i,j} \in C_i$, and assign the matrix value $X[i, j] = \rho_i^{\downarrow}(c_{i,j})$ (see Algorithm 8, lines 21-24).

This second optimization allows us to further reduce the number of unique CH graph searches from $k + kg + 2$ down to $3 \cdot (k + 1)$ (i.e., one forward super-source search, one backward depth-first search coverage, and one reverse topological search per adjacent category pair). We have now engineered a solution whose algorithmic approach requires only $O(k)$ total CH-related graph searches and $O(kg)$ matrix construction steps. As a result, we have effectively eliminated the category density's impact on the number of unique graph searches required to construct a solution.

### 5.4.4 Density Reduction via Pruning

Despite the algorithmic improvements presented thus far to reduce the impact of category density on our approach, we still have not fully eliminated the category density's effect on the overall runtime of our algorithm. This is because our $O(k)$ graph searches will still run in time proportional to the unioned CH search spaces of each individual category node. That is to say, the larger the density of any given category, the larger the unioned CH-graph search space will be. Therefore, one additional optimization to further reduce the runtime of our algorithm is to employ an aggressive pruning strategy to help minimize the possible sizes of each category's unioned search space.

To facilitate this approach, we require a method for quickly establishing a valid upper bound on the optimal solution cost. To achieve this, we propose to incorporate the use of a constant-time heuristic function, $h : V \times V \to \mathbb{R}_{\geq 0}$, which quickly estimates the shortest-path cost between any pair of nodes. We say that $h$ is an *admissible* heuristic function if $h(s,t) \leq d(s,t)$ for all $s,t \in V$ (i.e., it always underestimates the shortest-path cost between any given pair of nodes).

Given an admissible heuristic function, $h$, we may establish a valid upper bound, $\mu$, using a fast, greedy nearest-neighbor strategy (see pseudocode in Algorithm 6). Starting at node $x_0 = s$, for $1 \leq i \leq k+1$, we select the node $x_i = \underset{\forall c_{i,j} \in C_i}{argmin}\{h(x_{i-1}, c_{i,j})\}$, giving us the greedy node sequence $\langle x_0, \ldots, x_{k+1} \rangle$. We then compute the true shortest-path costs $d(x_{i-1}, x_i)$ for $1 \leq i \leq k+1$ using a series of successive (and very fast) PTP CH queries, computing $\mu = \sum_{1 \leq i \leq k+1} d(x_{i-1}, x_i)$. As the path formed by node sequence $\langle x_0, \ldots, x_{k+1} \rangle$ is, by definition, a *satisfying* path for $C$, the cost $\mu$ is also thus a valid upper bound on $w(P^C_{s,t})$.

After establishing $\mu$, we progressively prune each category using a user-provided input weighting parameter, $\alpha \geq 1$, intended to adjust the desired level of pruning. For each category, we prune the search as follows.

Before beginning the forward search from each category, $C_i$, we prune any node, $c_{i,j} \in C_i$, if $\alpha \cdot (X[i,j] + h(c_{i,j}, t)) > \mu$, as $X[i,j] + h(c_{i,j}, t)$ is a lower bound on any solution path containing $c_{i,j}$. Similarly, we wish to prune the nodes before beginning the backward search from each category, $C_i$, as well. However, unlike the forward search scenario, when performing a backward search from $C_i$, we do not yet know the matrix values $X[i,j]$ for any nodes $c_{i,j} \in C_i$ (as these values are not set until *after* the backward search). Therefore, we must instead rely on a lower bound value for $X[i,j]$ using the established matrix values from the previous category. We define this value as $\beta = \underset{1 \leq \ell \leq |C_{i-1}|}{min} \{X[i-1,\ell]\}$, as this is clearly a lower bound on any $X[i,j]$. Now, before beginning the backward search from each category, $C_i$, we prune any node, $c_{i,j} \in C_i$, if $\alpha \cdot (\beta + h(c_{i,j},t)) > \mu$.

The proposed $\alpha$-pruning strategy helps to significantly reduce the sizes of the resulting search spaces for both the forward and backward graph searches, further reducing the overall runtime. After completing the search process for each (pruned) adjacent category pair to construct the resulting X matrix, we now return the solution cost $min\{\mu, X[k+1,0]\}$.

**Theorem 28** *For query $\langle s,t,C\rangle$, the $\alpha$-pruning technique guarantees that $min\{\mu, X[k+1,0]\} \leq \alpha \cdot w(P^C_{s,t})$.*

**Proof.** If $\mu \leq \alpha \cdot w(P^C_{s,t})$, then the proof is complete since we return $min\{\mu, X[k+1,0]\}$. Therefore, suppose that $\mu > \alpha \cdot w(P^C_{s,t})$. In this case, we prove an even stronger claim that the solution cost returned is optimal (i.e., $X[k+1,0] = w(P^C_{s,t})$). It suffices to show that the algorithm will never prune any nodes, $c_{i,j} \in C_i$, belonging to any optimal solution path. This is because, for the forward search, $\alpha \cdot (X[i,j] + h(c_{i,j},t)) \leq \alpha \cdot w(P^C_{s,t}) < \mu$ for any node $c_{i,j} \in P^C_{s,t}$ by the admissibility condition of our heuristic function, $h$, and our initial assumption above. Similarly, for the backward search, $\alpha \cdot (\beta + h(c_{i,j},t)) \leq \alpha \cdot (X[i,j] + h(c_{i,j},t)) \leq \alpha \cdot w(P^C_{s,t}) < \mu$ for any node $c_{i,j} \in P^C_{s,t}$, because $X[i,j] \geq \beta = \underset{1 \leq \ell \leq |C_{i-1}|}{min} \{X[i-1,\ell]\}$. The optimality of $X[k+1,0]$ follows from this, and thus $min\{\mu, X[k+1,0]\} \leq \alpha \cdot w(P^C_{s,t})$, for $\alpha \geq 1$. ■

Note that any admissible heuristic function will work in this algorithm. However, the accuracy and estimation time of the chosen function will impact the overall pruning capability and runtime of our algorithm, respectively. We discuss the heuristic function used in the experimental results section. Starting from Algorithm 5, the pseudocode for our fully-optimized algorithm is presented in Algorithms 5-9.

We have presented a GSP solution based on Contraction Hierarchies which we have progressively engineered to eliminate (as much as possible) the negative impacts of the category density (as discussed in Section 5.4.1) on deriving our solution. Specifically, we have shown how to go from a relatively-straightforward solution requiring $O(kg^2)$ graph searches to an optimized solution requiring only $O(k)$ graph searches.

**Algorithm 5** GSP-Query$(s, t, C, \alpha)$

```
Input: $s, t \in V$, $C = \langle C_1, C_2, \dots, C_k \rangle$, $\alpha \in \mathbb{R}_{\geq 1}$
Output: $min\{\mu, X[k+1, 0]\} \leq \alpha \cdot w(P^C_{s,t})$
Global Variables: $G' = (V, E \cup E', w)$, $arrays\ \rho^{\uparrow}/\rho^{\downarrow}$, $h$
Invariant: $\forall v \in V, \forall i \in [0, k+1]$, $\rho^{\uparrow}_i(v) = \rho^{\downarrow}_i(v) = \infty$
1: $\mu \leftarrow EstablishUpperBound(s, t, C)$
2: $C_0 \leftarrow \{s\}$
3: $C_{k+1} \leftarrow \{t\}$
4: $X[0, 0] \leftarrow 0$
5: for $i = 1 \rightarrow k + 1$ do
6:     $ForwardSearch(C, X, i - 1, t, \alpha, \mu)$
7:     $BackwardSearch(C, X, i, t, \alpha, \mu)$
8: end for
9: return $min\{\mu, X[k+1, 0]\}$
```

**Algorithm 6** EstablishUpperBound($s, t, C$)

```
Input: $s, t \in V$, $C = \langle C_1, C_2, \ldots, C_k \rangle$
Output: $\mu \geq w(P_{s,t}^{C})$
Global Variables: $G' = (V, E \cup E', w)$, arrays $\rho^{\uparrow}/\rho^{\downarrow}$, $h$
Invariant: N/A
1: $\mu \leftarrow 0$
2: $x_0 \leftarrow s$
3: for $i = 1 \rightarrow k$ do
4:     $x_i = \underset{\forall c_{i,j} \in C_i}{argmin}\{h(x_{i-1}, c_{i,j})\}$
5:     $\mu \leftarrow \mu + PTP\text{-}CH\text{-}Query(x_{i-1}, x_i)$
6: end for
7: $x_{k+1} \leftarrow t$
8: $\mu \leftarrow \mu + PTP\text{-}CH\text{-}Query(x_k, x_{k+1})$
9: return $\mu$
```

## 5.5 Experiments

We present experimental results examining the overall performance characteristics of our proposed algorithms under various query scenarios. Sections 5.5.1 and 5.5.2 summarize our testing environment and the dataset used for our experiments, respectively. Section 5.5.3 briefly recaps the many different algorithm variants tested throughout our experiments. In Section 5.5.4, we present an experimental analysis of the impact of category density on the performance of these algorithms for both GSP queries and RTO-GSP queries. We consider both query variants to cover a range of different query localities, with GSP queries representing non-local queries (where the paths may cover long distances) and RTO-GSP queries representing very local queries (where the paths are generally very short). Section 5.5.5 similarly examines the impact of the number of chosen categories on performance. Finally, in Section 5.5.6, we explore approximate solution results for both GSP and GTSPP queries.

### 5.5.1 Test Environment

All experiments were run on a 64-bit server machine running Linux CentOS 5.3 with 2 quad-core CPUs clocked at 2.53 GHz with 18 GB RAM (only one core was used per experiment). All programs were written in C++ and compiled using gcc version 4.1.2 with optimization level 3.

### 5.5.2 Test Dataset

All experiments were performed on a graph of the continent-wide road network of North America[5], having a total of $21,133,774$ nodes and $52,523,592$ edges. The edge weight function, $w$, is based on the travel time (in minutes) required to cross each edge. This dataset was derived from NAVTEQ transportation data products, under their permission. For our experiments on this dataset, we have chosen the Pre-Computed Cluster Distances (PCD) heuristic function from [82] as our $\alpha$-pruning function, $h$. PCD partitions the graph into $r$ partitions and computes an $r \times r$ matrix of the shortest path costs between the closest nodes from each pair of partitions. The heuristic function $h(u, v)$ returns the matrix value between $u$'s partition and $v$'s partition, which is a lower bound on the true shortest path cost from $u$ to $v$. Given enough partitions, this approach generally provides very good lower bound estimates. For our experiments, we have chosen $r = 10,000$. Table 5.1 summarizes the times and storage overhead from both CH and PCD processing on this dataset (CH was used to speedup PCD computation).

[5]This includes only the US and Canada.

Table 5.1: Preprocessing results on the North American graph

| **Preprocessing Technique** | Time [H:M] | Space [B/node] |
|---|---|---|
| CH [53] | 0:18 | 35 |
| PCD [82] | 0:07 | 23 |

### 5.5.3 Algorithms Tested

In the following sections, we present the results of our discussed algorithms and optimizations for both GSP and RTO-GSP queries, respectively. Specifically, we compare results for the following algorithms (as labeled in each of the associated figures):

- **U. Dijkstra**: the regular expression constrained unidirectional Dijkstra variant adapted from [18].
- **B. Dijkstra**: the regular expression constrained bidirectional Dijkstra variant adapted from [18].
- **PTP-CH**: our algorithm which constructs the dynamic programming matrix by computing pairwise distances using point-to-point CH queries (Section 5.4.1).
- **MTM-CH**: our algorithm which constructs the dynamic programming matrix by computing all distances between adjacent categories using many-to-many CH queries (Section 5.4.1).
- **SSFS**: our MTM-CH algorithm (above) enhanced with the super-source forward search optimization (Section 5.4.2).
- **SSFS + RTBS**: our SSFS algorithm (above) enhanced with the reverse-topological backward search optimization (Section 5.4.3).
- **Alpha Pruning**: our SSFS + RTBS algorithm (above) enhanced with $\alpha$-pruning

(Section 5.4.4). Unless otherwise stated, $\alpha = 1$ for all experiments (for optimal solution results).

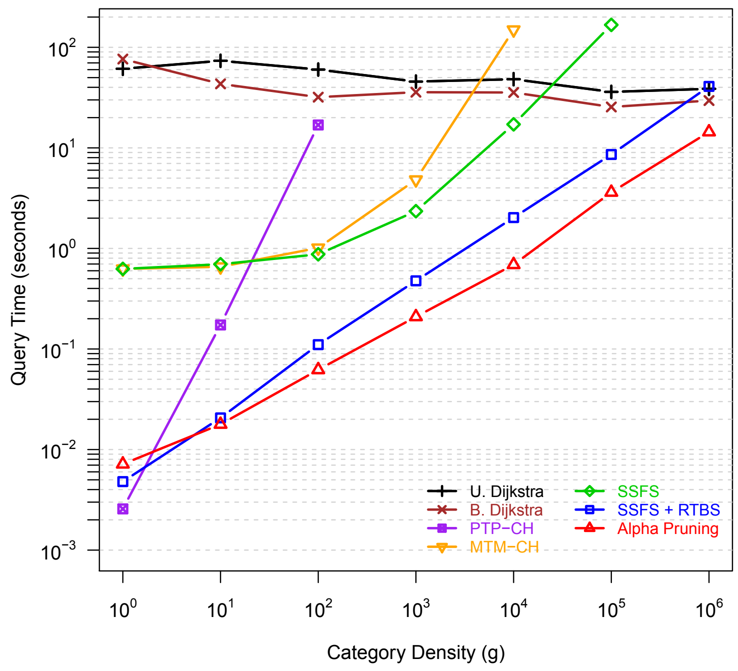
### 5.5.4 Category Density Experiments

We first examine the impact of the category density for a given query on the performance of our discussed algorithms. In Figure 5.3(a) and Figure 5.3(b) we present the results of our algorithms for both GSP and RTO-GSP queries, respectively. Note the logarithmic scale for both axes.

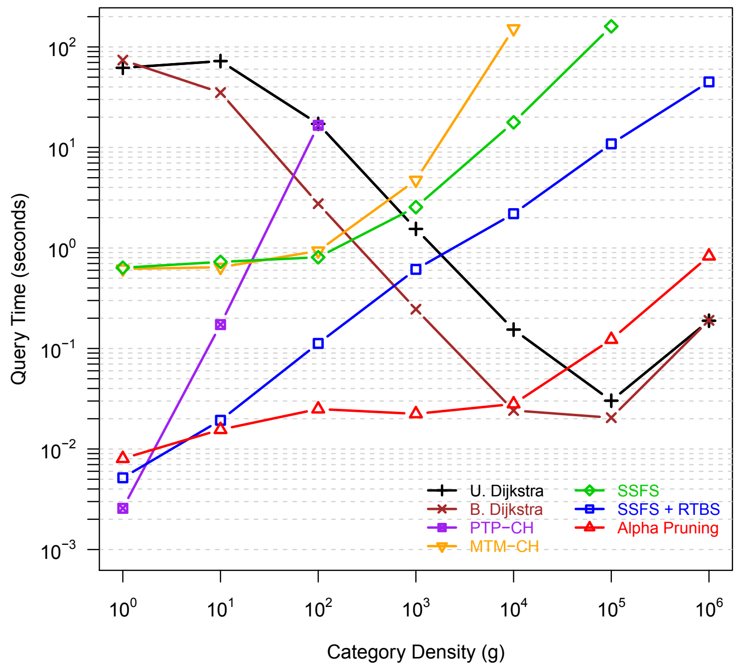
For each set of experiments in this section, we fixed the category count at $k = 5$. For every $0 \leq i \leq 6$, we constructed 100 random query instances in which each of the 5 categories were populated with $g = 10^i$ nodes selected uniformly at random. All source and target nodes, $s$ and $t$, were additionally selected uniformly at random. The numbers presented in the figures represent the average query times across these 100 queries for each density value tested.

Starting with the GSP query experiments of Figure 5.3(a), we first examine the results of the regular expression constrained Dijkstra variants: U. Dijkstra and B. Dijkstra. As can be seen, both algorithms are fairly consistent across varying densities at the national scale. Except for the case where $g = 1$, the bidirectional variant is also consistently faster than the unidirectional variant by an average factor of 1.4. However, B. Dijkstra still requires 40 seconds runtime on average and nearly 30 seconds, even in the best case.

Among our proposed algorithms, we start with the more straightforward adaptations of PTP-CH and MTM-CH. The PTP-CH algorithm is by far the fastest approach for the scenario where $g = 1$ (since this is equivalent to solving a multi-point shortest



(a) $s \neq t, k = 5$



(b) $s = t, k = 5$

Figure 5.3: Category density experiments for (a) GSP queries and (b) RTO-GSP queries

path), but, as it requires a quadratic number of CH searches in the density of the query, it quickly grows beyond practicality and does not scale to problems with $g \geq 1,000$. MTM-CH begins to outperform PTP-CH for cases with $g > 10$, but only scales up to $g \leq 10,000$. This is because computing an MTM cost matrix for larger cases where, e.g., $g \geq 100,000$, would require approximately up to and over 40GB of storage overhead per adjacent category pair.

This is where we start to see our proposed optimizations show improvements. For example, incorporating the SSFS optimization (from Section 5.4.2), while performing the typical MTM backward search, allows us to outperform the MTM-CH approach by a factor of 2.8, on average, and this further increases our scalability to cases of size $g \leq 100,000$. This is because we no longer have to compute and maintain a $g \times g$ cost matrix for each adjacent category pair, but instead require only to compute a $1 \times g$ matrix for each pair.

Further incorporating the RTBS improvement (from Section 5.4.3), gives us the SSFS + RTBS algorithm which now starts to show significantly better performance and overall scalability (a factor of 34 times speed improvement over SSFS, on average), since we have now effectively eliminated the previous negative impacts of the category density. By incorporating the final $\alpha$-pruning optimization (from Section 5.4.4), we are able to achieve yet another factor of 2 improvement, on average, over the SSFS + RTBS algorithm, due to the decrease in overall category density via our pruning strategies. Note that, in the best case (for relatively low-density queries), our fully-optimized algorithm gives up to nearly 4 orders of magnitude speed improvement over the fastest Dijkstra variant. Additionally, it scales quite nicely for extremely high-density problems with up to 1 million nodes per category, for random queries at the national scale.

For the return-to-origin query scenarios in Figure 5.3(b), we see much the same behavior for each algorithm as before, except for two notable differences. The first is that, as the density increases, we begin to see significant improvements in the performance of both Dijkstra variants. This is due primarily to the fact that the RTO-GSP queries are highly-localized queries by their very definition (i.e., $s = t$), and thus, for sufficiently-dense categories, any optimal path should not have to travel very far to satisfy the requested category sequence. For low-density scenarios (e.g., $g < 10,000$), the Dijkstra variants must search relatively longer distances in the graph before finding valid nodes from each successive category. As the density begins to increase (e.g., for high-density scenarios with $g \geq 10,000$), the probability of finding a nearby node belonging to the next required category also increases, and thus runtimes begin to improve drastically for these extreme local cases.

The second notable difference for these RTO-GSP queries is the significant improvement of the $\alpha$-pruning algorithm, compared to the regular GSP query scenarios from earlier. Specifically, we notice that the improvements compared to the SSFS + RTBS algorithm have now increased from a factor of 2 (for the GSP queries) up to a factor of 36, on average, for the return-to-origin scenarios. Again, this is because, as the queries become more localized and the categories become denser, then the average path length of any valid solution is significantly reduced. As a result, this leads to reduced upper bounds for the greedy path cost $\mu$, which in turn increases the potential for pruning outlier nodes that are relatively far away from these short, local paths.

While both Dijkstra algorithms start to outperform the $\alpha$-pruning algorithm for the local RTO-GSP queries with very high densities (e.g., $g \geq 100,000$), our optimized approach is still best overall, as it is the only algorithm presented here which has sub-second query times across all RTO-GSP queries, regardless of the density. This suggests that the fully-optimized $\alpha$-pruning algorithm is the most consistently efficient and scalable of all of the tested algorithms.

Finally, we note that we have designed these experiments (both for GSP and RTO-GSP) such that each category contains exactly the same maximum number of nodes, $g$, per query. This was done intentionally as an adversarial approach to our own proposed methods, as we have previously shown that any CH-based methods are uniquely sensitive to the overall density of each category (i.e., higher density per category implies larger CH search spaces, as discussed in Section 5.4.4). More realistic query scenarios would likely have varying densities per category (i.e., better than the worst-case, maximum density scenarios tested here), thus suggesting that the performance of the $\alpha$-pruning algorithm should not degrade in practice. In contrast, given the results above, the Dijkstra algorithms would be expected to further degrade without as many realistic category options.

As evidence of this expected behavior, consider the RTO-GSP test case from the previous results in which both the bidirectional Dijkstra algorithm and the $\alpha$-pruning algorithm were closest in performance: $g = 10,000$. In Figure 5.4, we compare these two algorithms as follows. For every $0 \leq i \leq 4$, we constructed 100 random query instances in which each of the $k = 5$ categories were populated with a number of nodes selected uniformly at random from the range $[\frac{g}{f}, g]$ where $f = 10^i$ represents the *reduction factor* on the possible category sizes (i.e., as $f$ increases, the variability of the possible category sizes from the maximum density $g$ also increases; $f = 10^0 = 1$ represents the previously tested scenarios). All nodes were selected uniformly at random. From these results, we can see that, as the variability in category sizes increases (allowing for category sizes increasingly smaller than $g$), the Dijkstra algorithm query times degrade by up to an

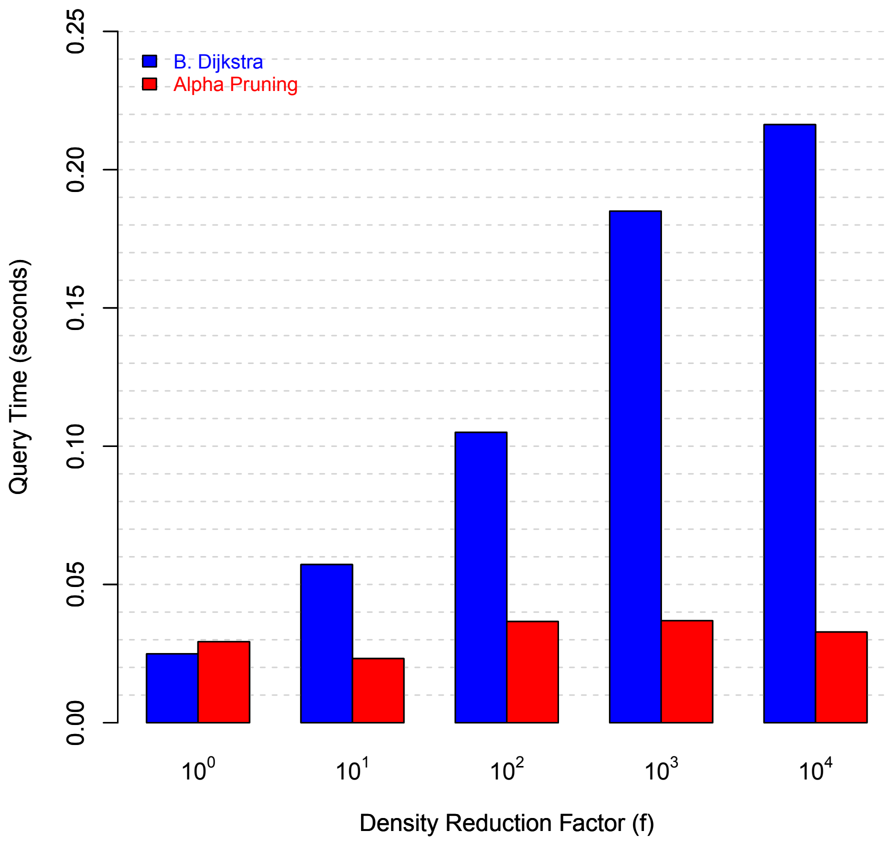


Figure 5.4: Density reduction factor experiments for RTO-GSP queries with $k = 5$ and $g = 10,000$

order of magnitude, whereas the $\alpha$-pruning algorithm runtimes are quite stable ($\pm 25\%$).

### 5.5.5 Category Count Experiments

We proceed by exploring the effect of the overall number of categories per query on the performance of the tested algorithms. The results for both GSP and RTO-GSP queries are presented in Figure 5.5(a) and Figure 5.5(b), respectively.

For each set of experiments in this section, we fixed the category density at $g = 10,000$. For every $1 \leq i \leq 10$, we constructed 100 random query instances, each with $i$ unique categories of density $g$, populated with nodes selected uniformly at random. All source and target nodes, $s$ and $t$, were additionally selected uniformly at random.

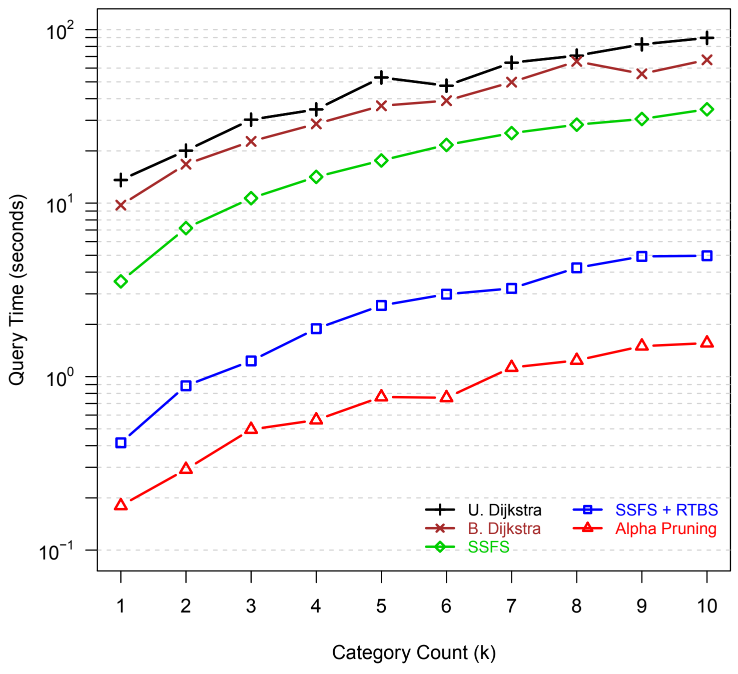
The numbers presented in the figures represent the average query times across these 100 queries for each category count value tested. Since the PTP-CH and MTM-CH algorithms do not scale well for problem sizes of this density, we have omitted these results from this section.

In general, for both figures we see that each tested algorithm variant scales quite well, with regard to category count. This (along with the previous section's results) confirms our expectation that the presented algorithms are significantly more sensitive to the overall density of the categories than they are to the overall category count. Furthermore, we see similar progressive improvements for our successively-applied optimizations as before.

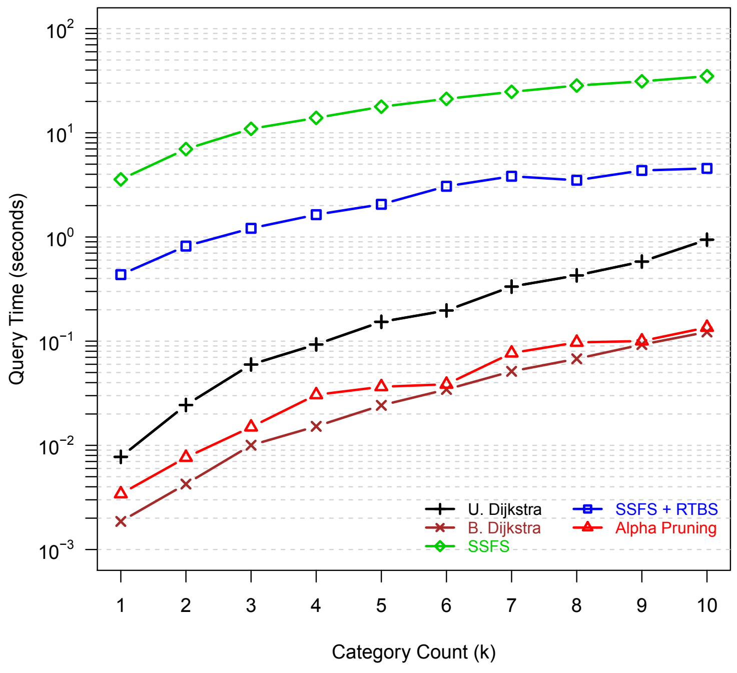
For the GSP queries in Figure 5.5(a), the $\alpha$-pruning technique significantly outperforms all other approaches and is a factor of 50 times faster than the fastest Dijkstra variant, on average, making it a clear winner for such queries. For the RTO-GSP scenarios in Figure 5.5(b), the Dijkstra algorithms again show significant improvement due to the locality of the queries, especially for such high-density scenarios (as indicated in the previous section). For these tests, the bidirectional Dijkstra variant slightly outperforms the $\alpha$-pruning algorithm (by an average factor of 1.5). Nevertheless, the $\alpha$-pruning algorithm again shows significantly more consistent efficiency and scalability when considered across all problem sizes and at all levels of locality.

### 5.5.6 Approximation Results

Thus far, we have examined the proposed $\alpha$-pruning algorithm only for queries in which $\alpha = 1$ (i.e., our solution results were guaranteed to be optimal). However, as seen in Figure 5.3(a), for extremely high-density, nationwide GSP queries with $g =$



(a) $s \neq t, g = 10,000$



(b) $s = t, g = 10,000$

Figure 5.5: Category count experiments for (a) GSP queries and (b) RTO-GSP queries

$1,000,000$, the $\alpha$-pruning algorithm still requires 14 seconds to complete for $k = 5$ categories (less than 3 seconds per category, on average). While this is a significant improvement over the other algorithms tested for this problem, we may improve upon these results even further for such high-density cases, if we are willing to accept some approximation of the optimal solution.

Table 5.2: GSP approximation results with $\alpha$-Pruning for GSP queries with $k = 5$ and $g = 1,000,000$

| $\alpha$ | Time (seconds) | Average Relative Error (%) |
|---|---|---|
| 1.00 | 14.40 | 0.00 % |
| 1.25 | 4.02 | 5.73 % |
| 1.50 | 3.11 | 9.26 % |
| 1.75 | 2.66 | 10.96 % |
| 2.00 | 2.85 | 11.43 % |
| 2.25 | 2.79 | 11.43 % |
| 2.50 | 2.51 | 11.43 % |
| 2.75 | 2.23 | 11.43 % |
| 3.00 | 2.80 | 11.43 % |

Table 5.3: GTSPP approximation results with $\alpha$-Pruning ($\alpha = 1$) for both RTO-GSP and GSP queries with $k = 5$

| | **GSP Queries** | | **RTO-GSP Queries** | |
|---|---|---|---|---|
| $g$ | Avg. Rel. Error (%) | Max Rel. Error (%) | Avg. Rel. Error (%) | Max Rel. Error (%) |
| $10^0$ | 39.01 % | 154.63 % | 28.72 % | 95.12 % |
| $10^1$ | 29.36 % | 123.68 % | 18.91 % | 64.24 % |
| $10^2$ | 8.21 % | 56.29 % | 14.77 % | 49.65 % |
| $10^3$ | 2.72 % | 111.03 % | 13.96 % | 56.86 % |
| $10^4$ | 0.60 % | 13.18 % | 16.58 % | 58.28 % |
| $10^5$ | 0.02 % | 0.55 % | 15.71 % | 61.58 % |
| $10^6$ | 0.00 % | 0.00 % | 18.30 % | 95.27 % |

As discussed in Theorem 28, the $\alpha$-pruning strategy guarantees to produce an $\alpha$-approximate solution to the queries, for any $\alpha \geq 1$. In Table 5.2, we explore the effect of using different $\alpha$ values on the previously-tested GSP queries from Figure 5.3(a) with $g = 1,000,000$ and $k = 5$. We present results on both the average runtimes of our queries, as well as their approximations.

As indicated by these results, we begin to see significant improvements in query times even for only modest increases in the $\alpha$ parameter. However, we additionally start to see a 'plateau' effect in the improvements once $\alpha$ gets sufficiently large. Specifically, for any $\alpha$ values greater than 2, the query times become relatively stable and the average relative errors become bounded, as we are unable to find any alternate solutions once the pruning becomes strong enough (i.e., the $\mu$ value quickly becomes the best value for each query, once we begin to aggressively prune entire categories of nodes).

In addition to approximating GSP solutions for fixed sequences of categories, we have also shown in Section 5.3 that we may further apply our GSP algorithm to closely approximate the more complex GTSPP queries, typically to within a bounded factor of $O(k)$. Experimental results examining the approximations achieved by our GSP algorithm when applied to GTSPP queries are presented in Table 5.3, where we present both the average and maximum relative errors of both GSP and RTO-GSP query solutions across 100 random query instances where $k = 5$ for several different density values. To obtain the optimal GTSPP solution results for comparison, we ran the exact GTSPP algorithm from [96]. We then applied a GSP query for a random, fixed sequence of categories to obtain the resulting GSP solution for comparison. The findings suggest that the relative errors are well within the expected approximation ranges, and for most cases achieve around or better than 30% relative error, on average. Interestingly, for the relatively longer-distance GSP query scenarios (where $s \neq t$), as the density continues to increase, the error achieved by our GSP solution decreases. At $d = 10^6$, the density is so extremely high that we achieve optimal results for the GTSPP queries using this approach, as any arbitrary sequence of categories is now relatively more likely to appear along these longer $s$-$t$ paths.

## 5.6 Conclusion

We have presented a new algorithmic approach towards efficiently solving GSP queries on real-world road networks. Experimental results support the claim that this approach generally outperforms the previous-best algorithm for this query type across many different scales of problem sizes, and often by a significant margin of improvement. Our approach may also be used to efficiently compute approximate solutions for the NP-hard GTSPP queries, where the category sequence is not fixed.

However, there is still room for further additional improvement on this approach. Specifically, results suggest that our proposed approach is currently weakest when solving for extremely high-density GSP queries. We could likely see improvements for such cases by investigating alternative pruning strategies in the hopes of achieving even sparser search spaces than what we currently have. However, for such extreme, high-density cases, another model altogether may also be warranted.

**Algorithm 7** ForwardSearch$(C, X, i, t, \alpha, \mu)$

```
Input: $C = \langle C_0, \ldots, C_{k+1} \rangle$, matrix $X$, $i \in [0, k]$, $t \in V$, $\alpha \in \mathbb{R}_{\geq 1}$, $\mu \in \mathbb{R}_+$
Global Variables: $G' = (V, E \cup E', w)$, arrays $\rho^{\uparrow}/\rho^{\downarrow}$, $h$
Invariant: $\forall\, v \in V$, $\rho_i^{\uparrow}(v) = \infty$
1: $PQ \leftarrow \emptyset$ /* Priority Queue */
2: /* Insert unpruned nodes into the priority queue */
3: for $j = 1 \to |C_i|$ do
4:   if $\alpha \cdot (X[i,j] + h(c_{i,j}, t)) \leq \mu$ then
5:     $\rho_i^{\uparrow}(c_{i,j}) \leftarrow X[i,j]$
6:     $PQ.Insert(c_{i,j}, X[i,j])$
7:   end if
8: end for
9: /* Perform the forward Dijkstra search */
10: while $\neg\, PQ.Empty()$ do
11:   $u \leftarrow PQ.ExtractMin()$
12:   for all $e = (u,v) \in E \cup E' : \phi(u) < \phi(v)$ do
13:     if $\rho_i^{\uparrow}(v) > \rho_i^{\uparrow}(u) + w(e)$ then
14:       $\rho_i^{\uparrow}(v) \leftarrow \rho_i^{\uparrow}(u) + w(e)$
15:       if $v \notin PQ$ then
16:         $PQ.Insert(v, \rho_i^{\uparrow}(v))$
17:       else
18:         $PQ.DecreaseKey(v, \rho_i^{\uparrow}(v))$
19:       end if
20:     end if
21:   end for
22: end while
```

**Algorithm 8** BackwardSearch$(C, X, i, t, \alpha, \mu)$

**Input:** $C = \langle C_0, \ldots, C_{k+1} \rangle$, matrix $X$, $i \in [1, k+1]$, $t \in V$, $\alpha \in \mathbb{R}_{\geq 1}$, $\mu \in \mathbb{R}_+$

**Global Variables:** $G' = (V, E \cup E', w)$, *arrays* $\rho^{\uparrow}/\rho^{\downarrow}$, $h$

**Invariant:** $\forall\, v \in V$, $\rho_i^{\downarrow}(v) = \infty$

```
 1: $Q \leftarrow \emptyset$ /* Queue to store nodes in reverse topological order */
 2: $R \leftarrow \emptyset$ /* Set of nodes reached by backward DFS */
 3: /* Perform backward DFS from unpruned nodes,
 4:      avoiding any previously-reached nodes in R */
 5: $\beta \leftarrow \underset{1 \leq \ell \leq |C_{i-1}|}{min} \{X[i-1, \ell]\}$
 6: for $j = 1 \to |C_i|$ do
 7:     if $\alpha \cdot (\beta + h(c_{i,j}, t)) \leq \mu$ then
 8:         $\langle Q, R \rangle \leftarrow BackwardDFS(c_{i,j}, Q, R)$
 9:     end if
10: end for
11: /* Process nodes in reverse topological order */
12: while $\neg\, Q.Empty()$ do
13:     $v \leftarrow Q.Pop()$
14:     $\rho_i^{\downarrow}(v) \leftarrow \rho_{i-1}^{\uparrow}(v)$
15:     for all $e = (u, v) \in E \cup E' : \phi(u) > \phi(v)$ do
16:         if $\rho_i^{\downarrow}(v) > \rho_i^{\downarrow}(u) + w(e)$ then
17:             $\rho_i^{\downarrow}(v) \leftarrow \rho_i^{\downarrow}(u) + w(e)$
18:         end if
19:     end for
20: end while
21: /* Set the resulting matrix values */
22: for $j = 1 \to |C_i|$ do
23:     $X[i, j] \leftarrow \rho_i^{\downarrow}(c_{i,j})$
24: end for
```

**Algorithm 9** BackwardDFS($v, Q, R$)

**Input:** $v \in V$, queue $Q$, $R \subseteq V$

**Output:** $Q$, $R$

**Global Variables:** $G' = (V, E \cup E', w)$, *arrays* $\rho^{\uparrow}/\rho^{\downarrow}$, $h$

**Invariant:** N/A

```
1: if $v \notin R$ then
2:     for all $e = (u, v) \in E \cup E' : \phi(u) > \phi(v)$ do
3:         $\langle Q, R \rangle \leftarrow BackwardDFS(u, Q, R)$
4:     end for
5:     $Q.Push(v)$
6:     $R \leftarrow R \cup \{v\}$
7: end if
8: return $\langle Q, R \rangle$
```

# Chapter 6

# Generalized Traveling Salesman Paths

## 6.1 Introduction

Within the last decade, the growing online presence of geospatial information systems has made possible many novel applications in the fields of transportation and location-based services. Many massive, online location databases are now being made publicly available for mining spatial locations based on categorical points of interest, thus paving the way for highly-advanced navigation solutions.

As an example, consider a traveler in a new city for the first time. On their way to do some sightseeing at a local attraction, they wish to visit a coffee house, a gas station, and an ATM (in no particular order). However, there may be many such locations to choose from for each of these location types. As the traveler likely does not care exactly *which* gas station, ATM, or coffee house they visit (since each provides the same general type of service[1]), a desirable solution is then any path which visits one of

[1]The locations are user-defined and may be made more specific, as necessary; e.g., only consider gas

each of these location types with the least overall detour on the way to the destination. Such a scenario is a common occurrence for everyday personal navigation needs, and also has many additional applications in transportation, in general.

In this chapter, we establish an algorithmic framework for efficiently solving such problem types on large-scale, real-world road networks. In Section 6.2, we formalize this problem as the Generalized Traveling Salesman Path Problem (GTSPP), and we discuss related work and our contributions. Section 6.3 presents the foundation for our work by formulating GTSPP as a unique graph search problem, including a standard search algorithm for this approach. Section 6.4 extends these ideas into a more advanced search algorithm, based around the Contraction Hierarchies preprocessing technique. We present an experimental analysis of these algorithms on the road network of North America in Section 6.5. We conclude the chapter in Section 6.6.

## 6.2 Generalized Traveling Salesman Path Problems

Let $G = (V, E, w)$ be a weighted directed graph, with $n = |V|$, $m = |E|$, and edge weight function $w : E \to \mathbb{R}_+$. Let $P_{s,t} = \langle v_1, v_2, \ldots, v_q \rangle$ be a path in $G$ from $s = v_1 \in V$ to $t = v_q \in V$. Let $w(P_{s,t}) = \sum_{1 \le i < q} w(v_i, v_{i+1})$ be the total weight, or cost, of $P_{s,t}$. The minimum-weight, or "shortest", path cost from $s$ to $t$ is $d(s,t)$.

A category set, $C = \{C_1, C_2, \ldots, C_k\}$, defines a set of node subsets where, for $1 \le i \le k$, $C_i = \{c_{i,1}, c_{i,2}, \ldots, c_{i,|C_i|}\} \subseteq V$ represents a distinct category of locations. A path, $P_{s,t}$, *satisfies* a category set $C$ if, for $1 \le i \le k$, $P_{s,t} \cap C_i \ne \emptyset$ (i.e., $P_{s,t}$ contains at least one node from each category). This is formally written as $P_{s,t} \models C$. For any GTSPP instance $\langle s, t, C \rangle$, having category count $k = |C|$ and category density

stations of a certain brand.

$g = \max_{1 \le i \le k} \{|C_i|\}$, an optimal solution is a path $P'_{s,t}$ in $G$ such that $P'_{s,t} \models C$ and, $\forall\, P_{s,t}$ in $G$ where $P_{s,t} \models C$, $w(P'_{s,t}) \le w(P_{s,t})$. This optimal solution path is referred to as $P^C_{s,t}$.

### 6.2.1 Related Work

The Generalized Traveling Salesman Problem (GTSP), also known as Errand Scheduling, Group TSP, Set TSP, One-of-a-Set TSP, Multiple-Choice TSP, and TSP with Neighborhoods, was originally introduced in the late 1960s [63, 110] as a generalization of the well-known TSP formulation. Given a set of nodes partitioned into groups, or categories, the goal is to find a minimum-cost tour that visits exactly one node from each category. As TSP is a special case of GTSP in which each node defines its own category, then GTSP is also NP-hard. A review of many original applications of this problem type is presented in [72].

Initial solutions for this problem were based on exact dynamic programming formulations [63, 100, 110]. Other exact algorithms based on integer- and linear-programming techniques are presented in [46, 73, 74, 87]. Much research has also been focused on transforming GTSP instances into standard TSP instances with roughly the same number of total nodes [24, 40, 80]. Under these transformations, an optimal solution to the transformed TSP instance is optimal for the original GTSP instance. However, exact algorithms for standard TSP remain exponential in the total number of nodes.

TSP problem variants having a fixed source node, $s$, and a fixed target node, $t$, are more commonly known as Traveling Salesman *Path* Problems. Therefore, in the remainder of our discussion we will be focused on the more specific Generalized Traveling Salesman Path Problem (GTSPP).

### 6.2.2 Our Contributions

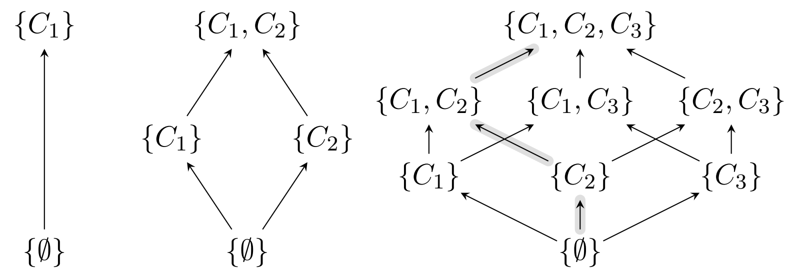
Unfortunately, nearly all of the previous exact algorithms assume a pre-existing, complete graph on the set of category nodes (represented as a cost matrix). Such an assumption is invalid for most real-world navigation scenarios involving road networks, as these cost matrices must be computed explicitly from the underlying road network, and we often do not know the category locations until query time (as they are likely to change from one use case to the next). Furthermore, computing such matrices requires $O(kg)$ graph searches, and can thus be quite time consuming, and potentially even prohibitive, in practice, especially for very large numbers of locations. For example, road networks can have categories with potentially millions of locations, and a complete matrix on such locations would require up to several terabytes of storage space.

To avoid these difficulties, we reformulate GTSPP as a unique graph search problem which does not require the construction of a complete matrix between category locations. Using this as our foundation for a general algorithmic framework, we present two exact graph search algorithms (one with pre-processing, and one without) for efficiently solving GTSPP instances on real-world road networks.

Additionally, while the number of locations to consider may be quite large in practice for many real-world GTSPP transportation and personal-navigation applications (e.g., $g = 100,000$), the number of categories is typically very small (e.g., $k = 3$). For such real-world problems in which $k \ll g$ (and often even $2^k \ll g$) is typically true, our proposed approach proves highly-advantageous because, unlike many of the other algorithms discussed previously, our exact algorithms are exponential only in the number of categories, $k$.

## 6.3 GTSPP Product Graphs

Given any category set $C = \{C_1, C_2, \ldots, C_k\}$, let $\mathcal{B}_k = (\mathcal{P}(C), \subset)$ be the partially-ordered set (poset) defined by the power set of $C$ when ordered by inclusion. Such a poset, $\mathcal{B}_k$, is called a Boolean lattice. The covering graph of a Boolean lattice poset $\mathcal{B}_k = (\mathcal{P}(C), \subset)$ on a category set $C$ is a graph $G(\mathcal{B}_k) = (\mathcal{P}(C), E(\mathcal{B}_k))$, where $E(\mathcal{B}_k) = \{(c, c') \mid c, c' \in \mathcal{P}(C) \wedge c \subset c' \wedge \nexists c'' \in \mathcal{P}(C) : c \subset c'' \subset c'\}$. The covering graph defines a directed acyclic graph (DAG) on $\mathcal{B}_k$. We present examples of the covering graphs for several Boolean lattices in Fig. 6.1. Note that any path in the covering graph from the empty set to the full set represents a specific sequence of categories along the path, based on their order of accumulation (see Fig. 6.1). All $k!$ category sequences are thusly represented as paths in this graph. Also note that the Boolean lattice and its covering graph are exactly the same for *any* set of size $k$, as we can map the set members into the natural numbers $\{1, 2, \ldots, k\}$ (hence the subscript $k$, not $C$, in $\mathcal{B}_k$).



(a) $G(\mathcal{B}_1)$ (b) $G(\mathcal{B}_2)$ (c) $G(\mathcal{B}_3)$

Figure 6.1: Covering graphs for Boolean lattice posets $\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{B}_3$ (from left to right), respectively. The path highlighted in grey for $G(\mathcal{B}_3)$ represents the category traversal sequence $\langle C_2, C_1, C_3 \rangle$, based on the order of category accumulation along the path.

Given any graph $G = (V, E, w)$ and Boolean lattice $\mathcal{B}_k$ for a category set $C$ of size $k$, we define the GTSPP product graph as $G_C = G \times G(\mathcal{B}_k) = (V \times \mathcal{P}(C), E_1 \cup E_2)$, where product nodes are represented as $\langle u, c\rangle$ such that $u \in V$ and $c \in \mathcal{P}(C)$, $E_1 = \{(\langle u, c\rangle, \langle v, c'\rangle) \mid c = c' \wedge (u, v) \in E\}$, and $E_2 = \{(\langle u, c\rangle, \langle v, c'\rangle) \mid u = v \wedge (c, c') \in E(\mathcal{B}_k) \wedge v \in c' \setminus c\}$. The $E_1$ edges represent a copy of each original edge from $G$ for every subset of $C$. For all $(\langle u, c\rangle, \langle v, c'\rangle) \in E_1$, we define $w(\langle u, c\rangle, \langle v, c'\rangle) = w(u, v)$. The $E_2$ edges represent the accumulation of a new category (based on a corresponding covering graph edge) by inclusion of a specific node within that category. For all $(\langle u, c\rangle, \langle v, c'\rangle) \in E_2$, we define $w(\langle u, c\rangle, \langle v, c'\rangle) = 0$. Any path from $\langle s, \emptyset\rangle$ to $\langle t, C\rangle$ in $G_C$ therefore represents a satisfying path in the original graph, based on a specific accumulation sequence of category nodes from each category.

We present a simple example of a GTSPP product graph in Fig. 6.2. For this problem instance, we have two unique categories (each with two unique nodes), and we must find the minimum-cost path from $s$ to $t$ which traverses one node from each category (as shown in green in the original graph). The resulting product graph is shown on the right of the figure. Edges from the set $E_1$ are shown as solid edges, whereas edges from the set $E_2$ are shown as dashed edges. The shortest path from $\langle s, \emptyset\rangle$ to $\langle t, C\rangle$ in the product graph is highlighted in grey, and its cost is equivalent to the optimal solution cost for this GTSPP instance.

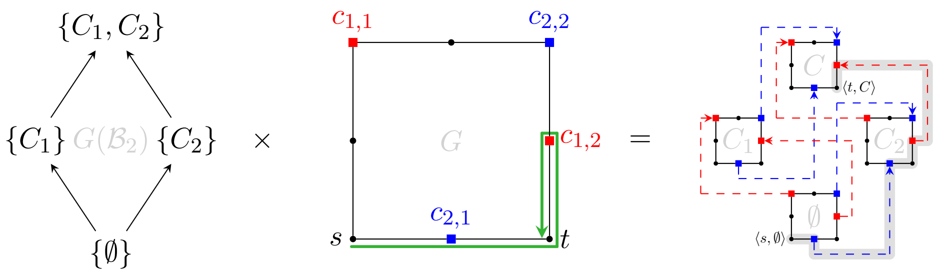


Figure 6.2: Example product graph of graph $G$ (with unit-cost edge weights) for category set $C = \{C_1, C_2\}$.

**Theorem 29** *Given any graph $G$ and category set $C = \{C_1, C_2, \ldots, C_k\}$ (defined on $G$), the shortest path from $\langle s, \emptyset\rangle$ to $\langle t, C\rangle$ in the product graph $G_C$ represents an equivalent-cost, optimal solution for the GTSPP instance $\langle s, t, C\rangle$ in the original graph $G$; i.e., $d(\langle s, \emptyset\rangle, \langle t, C\rangle) = w(P^C_{s,t})$.*

**Proof.** The proof relies on showing that there exists a one-to-one correspondence between the set of all paths from $\langle s, \emptyset\rangle$ to $\langle t, C\rangle$ in $G_C$ and the set of all uniquely-satisfying paths from $s$ to $t$ in $G$, and that the corresponding paths between each graph are of equivalent cost. It follows that a shortest path from $\langle s, \emptyset\rangle$ to $\langle t, C\rangle$ in $G_C$ represents a minimum-cost satisfying path from $s$ to $t$ in $G$, and is thus optimal for GTSPP.

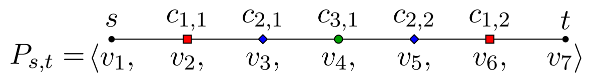

$s$
$c_{1,1}$
$c_{2,1}$
$c_{3,1}$
$c_{2,2}$
$c_{1,2}$
$t$
$P_{s,t} = \langle v_1, \; v_2, \; v_3, \; v_4, \; v_5, \; v_6, \; v_7\rangle$

Figure 6.3: Satisfying path for category set $C = \{C_1, C_2, C_3\}$ with multiple satisfying subsequences.

To establish this correspondence, we must first define what is meant by the term uniquely-satisfying path (USP). Any satisfying path (as defined in Section 6.2) may actually contain multiple "satisfying subsequences", informally defined as a subsequence of nodes along the path which satisfies, or "covers", all categories of interest (i.e., there is at least one node from each category; a more formal definition is presented later below). For example, consider the path $P_{s,t}$ in Fig. 6.3. For the given category set $C = \{C_1, C_2, C_3\}$ there are multiple selections and sequences of category nodes which satisfy this particular category set, even though there is only one path. Specifically, the subsequence $\langle c_{1,1}, c_{3,1}, c_{2,2}\rangle$ is a satisfying subsequence, but so too are $\langle c_{2,1}, c_{3,1}, c_{1,2}\rangle$, $\langle c_{1,1}, c_{2,1}, c_{3,1}\rangle$, and $\langle c_{3,1}, c_{2,2}, c_{1,2}\rangle$. In order to properly demonstrate the correspondence, we require a more explicit method for distinguishing exactly which satisfying subsequence is intended to be associated with a specific satisfying path. We may therefore think of a USP as any combination of a satisfying path, plus a unique satisfying subsequence of nodes belonging to that path. Note also, however, that we cannot simply represent a satisfying subsequence as merely a subsequence of nodes (as done informally above), because some nodes may in fact belong to multiple categories (in which case, the associated visit order of the categories may be ambiguous). For such cases, we need an even more explicit approach to further distinguish not only the subsequence of nodes, but also the intended permutation of categories visited by the subsequence.

A USP instance is thus formally defined as a pair $(P_{s,t}, Z)$, where $P_{s,t} = \langle v_1, v_2, \ldots, v_q \rangle$ represents a satisfying path from $s$ to $t$ in the original graph, $G$, and $Z = \langle (x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k) \rangle$ represents a satisfying subsequence. Specifically, $Z$ is defined such that $\langle v_{x_1}, \ldots, v_{x_k} \rangle$ is a (possibly-stuttering[2]) subsequence of nodes belonging to $P_{s,t}$ where $1 \leq x_1 \leq x_2 \leq \ldots \leq x_k \leq q$, and $\langle C_{y_1}, \ldots, C_{y_k} \rangle$ is a permutation of the categories as they are intended to be visited along the path, where, for $1 \leq i \leq k$, $v_{x_i} \in C_{y_i}$. Continuing from our earlier example, for the first subsequence of nodes discussed, $\langle c_{1,1}, c_{3,1}, c_{2,2} \rangle = \langle v_2, v_4, v_5 \rangle$, we would thus represent the associated USP as $(P_{s,t}, \langle (2,1), (4,3), (5,2) \rangle)$. The cost of a USP instance $(P_{s,t}, Z)$ is defined as $w(P_{s,t})$.

Given this definition of a USP, we may now establish the intended one-to-one correspondence. We start by providing a function (Algorithm 10) that maps any path from $\langle s, \emptyset \rangle$ to $\langle t, C \rangle$ in the product graph $G_C$ to a USP instance. For any such path $P_{\langle s,\emptyset \rangle, \langle t,C \rangle} = \langle \langle v_1, c_1 \rangle, \langle v_2, c_2 \rangle, \ldots, \langle v_{q'}, c_{q'} \rangle \rangle$, we construct its associated USP as follows.

First, we must establish the satisfying path $P_{s,t}$ in the original graph (Algorithm 10: lines 1-6). The function strips each product node in the path of all but the

[2]Where $v_{x_i} = v_{x_{i+1}}$ may be true; e.g., if a node belongs to multiple categories visited consecutively at that node.

**Algorithm 10** MappingFromProductPathToUSP($P_{\langle s,\emptyset\rangle,\langle t,C\rangle}$)

**Input:** *Product-graph path* $P_{\langle s,\emptyset\rangle,\langle t,C\rangle} = \langle\langle v_1, c_1\rangle, \langle v_2, c_2\rangle, \ldots, \langle v_{q'}, c_{q'}\rangle\rangle$

**Output:** *USP* $(P_{s,t}, Z)$

```
1:  $P_{s,t} \leftarrow \langle v_1\rangle$
2:  for $i \leftarrow 2$ to $q'$ do
3:      if $v_i \neq v_{i-1}$ then
4:          $P_{s,t} \leftarrow P_{s,t} \cdot \langle v_i\rangle$ /* Append $v_i$ to $P_{s,t}$ */
5:      end if
6:  end for
7:  $i \leftarrow 1$
8:  for $j \leftarrow 1$ to $q' - 1$ do
9:      if $(\langle v_j, c_j\rangle, \langle v_{j+1}, c_{j+1}\rangle) \in E_2$ then
10:         $x_i \leftarrow j + (1 - i)$
11:         $C_\ell \leftarrow c_{j+1} \setminus c_j$
12:         $y_i \leftarrow \ell$
13:         $i \leftarrow i + 1$
14:     end if
15: end for
16: $Z \leftarrow \langle (x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\rangle$
17: return $(P_{s,t}, Z)$
```

node identifier (i.e., each $\langle v_i, c_i\rangle$ becomes simply $v_i$), and then eliminates any consecutive identical node identifiers (i.e., for any $v_i = v_{i+1}$, it removes the duplicate $v_{i+1}$; note that such duplicates can only occur where the product-graph path crosses an $E_2$ edge). For example, in Fig. 6.2, the shortest path from $\langle s,\emptyset\rangle$ to $\langle t, C\rangle$ in the resulting product graph $G_C$ is $\langle\langle s,\emptyset\rangle,\langle c_{2,1},\emptyset\rangle,\langle c_{2,1},C_2\rangle,\langle t,C_2\rangle,\langle c_{1,2},C_2\rangle,\langle c_{1,2},C\rangle,\langle t,C\rangle\rangle$, and the obtained path in the original graph is therefore $\langle s,c_{2,1},t,c_{1,2},t\rangle$. This process is effectively equivalent to removing all but the $E_1$ edges along the product-graph path, and taking only the original node identifiers. Thus, the USP must also have the same cost as the product-graph path, since the resulting path $P_{s,t}$ has omitted only $E_2$ edges, which have zero cost, and all $(\langle u,c\rangle,\langle v,c'\rangle)\in E_1$ have $w(\langle u,c\rangle,\langle v,c'\rangle)=w(u,v)$, by definition.

Next, we must establish the satisfying subsequence associated with this path for our USP (Algorithm 10: lines 7-16). For any path from $\langle s,\emptyset\rangle$ to $\langle t,C\rangle$ in the product graph $G_C$, there must exist exactly $k$ unique $E_2$ edges, $\langle e_1,\ldots,e_k\rangle$, along this path, by definition of our product graph. This edge sequence defines the composition of our USP's satisfying subsequence. Specifically, for $1\le i\le k$, where $e_i=(\langle v_j,c_j\rangle,\langle v_{j+1},c_{j+1}\rangle)$ for some $j\in[1,q')$, let $x_i=j+(1-i)$ and let $y_i=\ell$ for $\ell\in[1,k]$, such that $c_{j+1}\backslash c_j=C_\ell\in C$. Note that we must offset each $x_i$ above by $(1-i)$ to account for the fact that each $x_i$ indexes into the associated path $P_{s,t}$ which has $k$ fewer nodes than the original product-graph path (i.e., $q=q'-k$), due to the removal of the duplicate node identifiers related to the $E_2$ edges of the product path.

It remains to show that our function is a bijective function (i.e., a one-to-one correspondence, which is both injective and surjective). A full verification of this requires significant detail, so we offer only a brief argument here. One can verify that the function is injective (i.e., no two paths in the product graph map to the same USP) by observing the following. If any pair of distinct paths in the product graph is claimed to map to the same USP, consider the first edge at which these two distinct product graph paths differ. If these differing edges are both from $E_1$, then the satisfying path $P_{s,t}$ in their USP must differ. If one of these differing edges is from $E_1$ and the other is from $E_2$, then the sequence $\langle x_1,\ldots,x_k\rangle$ in their USP must differ. If these differing edges are both from $E_2$, then the sequence $\langle y_1, \ldots, y_k \rangle$ in their USP must differ. Each case leads to a different USP (a contradiction of the supposed claim). Additionally, one can verify that the function is surjective (i.e., there is a path in the product graph which maps to each USP) by reversing the process of the function above (obtaining an inverse function). The pseudocode for an inverse function to reverse this process (i.e., mapping from a USP back to its associated product-graph path) is presented in Algorithm 11 (further discussion is omitted for brevity). Under such bijection, we have demonstrated the intended one-to-one correspondence.

**Algorithm 11** MappingFromUSPToProductPath($P_{s,t}, Z$)

```
Input: USP $(P_{s,t}, Z)$: $P_{s,t} = \langle v_1, \ldots, v_q \rangle$ and $Z = \langle (x_1, y_1), \ldots, (x_k, y_k) \rangle$
Output: Product-graph path $P_{\langle s,\emptyset\rangle,\langle t,C\rangle} = \langle \langle v_1, c_1 \rangle, \langle v_2, c_2 \rangle, \ldots, \langle v_{q'}, c_{q'} \rangle \rangle$
 1: $C' \leftarrow \emptyset$
 2: $P_{\langle s,\emptyset\rangle,\langle t,C\rangle} \leftarrow \langle\rangle$
 3: $j \leftarrow 1$
 4: for $i \leftarrow 1$ to $q$ do
 5:     $P_{\langle s,\emptyset\rangle,\langle t,C\rangle} \leftarrow P_{\langle s,\emptyset\rangle,\langle t,C\rangle} \cdot \langle v_i, C' \rangle$ /* Append $\langle v_i, C' \rangle$ to $P_{\langle s,\emptyset\rangle,\langle t,C\rangle}$ */
 6:     while $j \leq k \wedge i = x_j$ do
 7:         $C' \leftarrow C' \cup C_{y_j}$
 8:         $P_{\langle s,\emptyset\rangle,\langle t,C\rangle} \leftarrow P_{\langle s,\emptyset\rangle,\langle t,C\rangle} \cdot \langle v_i, C' \rangle$ /* Append $\langle v_i, C' \rangle$ to $P_{\langle s,\emptyset\rangle,\langle t,C\rangle}$ */
 9:         $j \leftarrow j + 1$
10:     end while
11: end for
12: return $P_{\langle s,\emptyset\rangle,\langle t,C\rangle}$
```

We may now conclude the proof as follows. Since all satisfying paths from $s$ to $t$ in the original graph must belong to at least one USP in the set of all USPs, and the set of all product-graph paths from $\langle s, \emptyset \rangle$ to $\langle t, C \rangle$ is in one-to-one correspondence with the set of USPs (and have equivalent cost), it follows that a shortest path from $\langle s, \emptyset \rangle$ to $\langle t, C \rangle$ in $G_C$ must also represent a minimum-cost satisfying path from $s$ to $t$ in $G$. ■

It follows from Theorem 29 that any shortest path algorithm will suffice to search the resulting product graph (e.g., Dijkstra's algorithm [39]).

**Theorem 30** *Given any graph $G$ and category set $C = \{C_1, C_2, \ldots, C_k\}$ (defined on $G$), a Dijkstra search from $\langle s, \emptyset \rangle$ to $\langle t, C \rangle$ in the product graph $G_C$ runs in $O(2^k(m + nk + nlogn))$ time.*

**Proof.** A Dijkstra search on a graph with $n$ nodes and $m$ edges takes $O(m + nlogn)$ time (using Fibonacci heaps). The product graph has exactly $2^k n$ nodes and at most $2^k(m + \frac{kg}{2})$ edges (i.e., we have exactly $2^k m$ $E_1$ edges and at most $\sum_{1 \leq i \leq k} (gi\binom{k}{i}) = 2^{k-1}kg$ $E_2$ edges). This gives a running time of $O(2^k(m + kg) + 2^k nlog(2^k n))$. Since $log(2^k n) \in O(k + logn)$ and $g \leq n$, this simplifies to $O(2^k(m + nk + nlogn))$. ■

Note that we do not need to explicitly construct the entire product graph to carry out the proposed search. We may instead perform an equivalent search in this product graph by materializing the nodes of the graph only as they are encountered implicitly during the search. This results in the potential for significant space savings for cases in which a solution path is found before most of the nodes are explored.

## 6.4 Product Graph Search Using Contraction Hierarchies

In this section, we further improve upon our proposed product-graph search approach by incorporating the graph preprocessing technique known as Contraction

Hierarchies (CH) [53], originally designed for solving point-to-point (PTP) shortest path queries. We start with a brief overview of CH, followed by a discussion of how to integrate CH for efficiently solving GTSPP queries.

### 6.4.1 Contraction Hierarchies Overview

**Preprocessing.** CH preprocessing orders the nodes in the graph, $\phi : V \rightarrow \{1, \ldots, |V|\}$, and then *contracts* the nodes in this order. Contracting a node, $v$, removes it (temporarily) from the graph and adds so-called *shortcut* edges (if needed) to preserve shortest path costs in the remaining subgraph. For each pair of incoming and outgoing edges, $(u, v)$ and $(v, x)$, respectively, if the path $\langle u, v, x\rangle$ is a unique shortest path, then a new shortcut edge $(u, x)$ is added with weight $w(u, v) + w(v, x)$ to bypass $v$ in the remaining subgraph. The result is a new graph $G' = (V, E \cup E', w)$, where $E'$ represents the newly-added shortcut edges.

**Query.** The traditional CH shortest path query involves performing a forward Dijkstra search from $s$ in the "upward" graph $G^{\uparrow} = (V, E^{\uparrow})$ where $E^{\uparrow} = \{(u, v) \in E \cup E' \mid \phi(u) < \phi(v)\}$ along with a simultaneous backward Dijkstra search from $t$ in the "downward" graph $G^{\downarrow} = (V, E^{\downarrow})$ where $E^{\downarrow} = \{(u, v) \in E \cup E' \mid \phi(u) > \phi(v)\}$. Let $R_s^{\uparrow} = \{v \in V \mid \exists P_{s,v} \subseteq G^{\uparrow}\}$ be the set of all nodes reachable from $s$ in the upward graph. Similarly, let $R_t^{\downarrow} = \{v \in V \mid \exists P_{v,t} \subseteq G^{\downarrow}\}$ be the set of all nodes from which $t$ is reachable in the downward graph. Let $d_s^{\uparrow}(v)$ and $d_t^{\downarrow}(v)$ represent the shortest path cost from $s$ to $v$ in $G^{\uparrow}$ and from $v$ to $t$ in $G^{\downarrow}$, respectively. The shortest path cost is taken as $d(s, t) = \underset{\forall v \in R_s^{\uparrow} \cap R_t^{\downarrow}}{min} \{d_s^{\uparrow}(v) + d_t^{\downarrow}(v)\}$.

### 6.4.2 Contraction Hierarchies for GTSPP

**Sweeping Search.** We preface this discussion by first establishing an alternative PTP query approach, which does not require bidirectional Dijkstra search, but instead takes advantage of a nice structural property of the resulting CH search graphs.[3] Specifically, each search graph is acyclic by definition (i.e., each search can only increase in node rank order $\phi$), thus allowing us to compute shortest path costs using only linear scans of the search graphs in topological order. Since the shortest path cost can be determined by considering only the reachable search spaces of $s$ and $t$ in $G^{\uparrow}$ and $G^{\downarrow}$ respectively, it suffices to consider only the union of these two search spaces $R = R_s^{\uparrow} \cup R_t^{\downarrow}$. Let $R_\phi = \langle v_1, v_2, \ldots, v_z \rangle$ be the nodes of $R$ arranged in increasing rank order, $\phi$, establishing a valid topological order. For any $v \in R$, let the value $d(v)$ represent the cost of the shortest path from $s$ found so far during the search.

The search begins by initializing $d(s) = 0$ and $d(v) = \infty$ for all $v \neq s \in R$. The search progresses in two phases: an *upsweep* and a *downsweep* phase. The upsweep phase processes each node $v_i$ in increasing order of $\phi$. For each outgoing edge $(v_i, x)$, such that $x \in R$ and $\phi(v_i) < \phi(x)$, we set $d(x) = min\{d(x), d(v_i) + w(v_i, x)\}$. The downsweep phase then processes each node $v_i$ in the opposite (decreasing) order of $\phi$. For each incoming edge $(u, v_i)$, such that $u \in R$ and $\phi(u) > \phi(v_i)$, we set $d(v_i) = min\{d(v_i), d(u) + w(u, v_i)\}$.

**Lemma 31** *Upon completion of the upsweep and downsweep phases,* $d(v) = d(s, v)$ *for all* $v \in R$ *such that* $R_v^{\downarrow} \subseteq R$.

**Proof.** Consider any $v \in R$ such that $R_v^{\downarrow} \subseteq R$ (e.g., $v = t$), for which we must show that $d(v) = d(s, v)$ after completing the sweeping search. Assume there exists a path

[3]A similar approach has been discussed independently in [34].

from $s$ to $v$ (we examine the alternate case later below). For any such path $P_{s,v} = \langle s = v_1, \ldots, v_i, \ldots, v_q = v \rangle$, let $M_{P_{s,v}} = \{v_i \in P_{s,v} \mid 1 < i < q, \phi(v_{i-1}) > \phi(v_i) < \phi(v_{i+1})\}$ (i.e., the set of all local minima in $P_{s,v}$ with respect to $\phi$). We can classify all paths, $P_{s,v}$, into one of two basic forms: (1) those with $M_{P_{s,v}} = \emptyset$ and (2) those with $M_{P_{s,v}} \neq \emptyset$.

From [53], we have that, if there exists a path from $s$ to $v$ in the original graph, there must exist a shortest $s$-$v$ path of the form (1) in the resulting CH graph search space $R$. For $x \in [1, q]$, let $v_x \in R$ be the highest-ranking node on any such shortest path $P'_{s,v} = \langle v_1, \ldots, v_x, \ldots, v_q \rangle$. That is, for $1 \leq i < x$, we have $\phi(v_i) < \phi(v_{i+1})$ (i.e., the rank is strictly increasing from node $s = v_1$ to node $v_x$), and for $x \leq i < q$, we have $\phi(v_i) > \phi(v_{i+1})$ (i.e., the rank is strictly decreasing from node $v_x$ to node $v_q = v$).

As the rank strictly increases along this path up to node $v_x$, we correctly compute the values for the nodes along the subpath $\langle v_1, v_2, \ldots, v_x \rangle$ during the upsweep phase, since, starting with $d(s) = d(v_1) = 0$, we process the nodes of $R_s^{\uparrow} \subseteq R$ in increasing rank order (and thus, in shortest-path order). Similarly, as the rank strictly decreases along this path from node $v_x$, we correctly compute the values for the nodes along the subpath $\langle v_{x+1}, \ldots, v_{q-1}, v_q \rangle$ during the downsweep phase, since we process the nodes of $R_v^{\downarrow} \subseteq R$ in decreasing rank order (and thus, also in shortest-path order). Therefore, upon completion of the sweeping phases, $d(v_q) = d(v) = d(s, v)$.

In the event there is no path from $s$ to $v$, then we have that $d(s, v) = \infty$. If there is no path in the original graph, there will be no path in the CH search space $R$ (as the added shortcut edges represent only existing paths in the original graph), and thus $d(v) = \infty$ remains after initialization, and is also correct. ■

This approach may be further extended to support many-to-many scenarios with multiple source nodes $S = \{s_1, s_2, \ldots, s_{|S|}\} \subseteq V$ and target nodes $T = \{t_1, t_2, \ldots, t_{|T|}\} \subseteq V$ by maintaining $|S|$ separate cost values for each node in the unioned search space $R = \{\bigcup_{i=1}^{|S|} R_{s_i}^{\uparrow}\} \cup \{\bigcup_{i=1}^{|T|} R_{t_i}^{\downarrow}\}$. Only, now, when relaxing the edges of a node during both the upsweep and downsweep phases on $R_\phi$, we relax all $|S|$ costs before progressing to the next node. A similar proof of correctness (like that of Lemma 31) follows.

**LEvel Sweeping Search (LESS).** We incorporate this alternative CH-based search algorithm into our GTSPP product-graph framework as follows. For the remainder of the chapter, we shall assume that our product graph incorporates the preprocessed CH graph. By construction, each product graph is comprised of exactly $(k+1)$ unique *levels* (as shown in Fig. 6.4). For $0 \leq i \leq k$, level $G_i = G_C[\{\langle u, c\rangle \in V(G_C) \mid i = |c|\}]$ is the subgraph induced by product nodes whose associated category subset has cardinality $i$. Each level is further comprised of smaller subgraphs $G^x = G_{|x|}[\{\langle u, c\rangle \in V(G_{|x|}) \mid c = x\}]$, which we shall call here *set-equivalent* (SE) subgraphs (also shown in Fig. 6.4). Within each level $G_i$, by definition of our product graph, there cannot exist any path between two distinct SE subgraphs (since the $E_1$ edges only connect nodes within the same SE subgraph and the $E_2$ edges only connect nodes between consecutive levels). For $i > 0$, this suggests that the costs within each SE subgraph of level $G_i$ must therefore depend solely on the costs established in the previous level $G_{i-1}$.

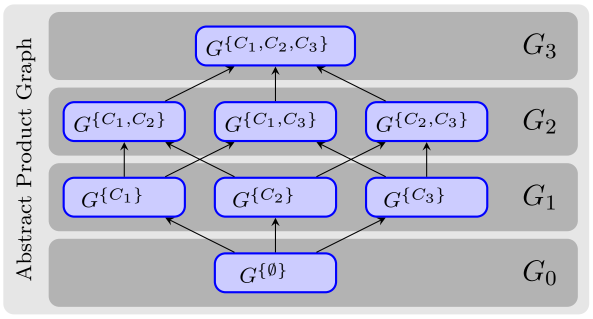


Figure 6.4: An abstraction of the product graph $G_C$ for $C = \{C_1, C_2, C_3\}$, illustrating levels (shown in dark grey) and SE subgraphs (shown in light blue).

We wish to take advantage of this useful property in our GTSPP search. Specifically, this allows us to process our product graph search one level at a time. Furthermore, for each level, we may process the costs for each SE subgraph independently, as these costs cannot influence one another. One solution would be to separately search each SE subgraph (using separate upsweep/downsweep phases), in any given order, for a given level. However, this would require $2^k$ separate upsweep and downsweep phases, one for each SE subgraph, and could be largely redundant as the separate sweeps may re-explore many of the same nodes and edges. An equivalent and more cache-efficient formulation is to instead perform only $(k+1)$ upsweep/downsweep phases (one for each level), maintaining a separate cost value per node for each SE subgraph within a given level. That is, when we process a node during a sweeping phase, we process all SE subgraph costs at the current level for that node before proceeding to the next node.

Since we only need to consider shortest paths that begin at $s$, travel through some nodes in $C$, and end at $t$, we may represent this search space as $R = \{R_s^{\uparrow}\} \cup \{\bigcup_{i=1}^{k} \bigcup_{j=1}^{|C_i|} \{R_{c_{i,j}}^{\downarrow} \cup R_{c_{i,j}}^{\uparrow}\}\} \cup \{R_t^{\downarrow}\}$. We begin by initializing $d(\langle s, \emptyset \rangle) = 0$, and for all other product nodes $\langle u, c \rangle \neq \langle s, \emptyset \rangle \in R \times \mathcal{P}(C)$, we initialize $d(\langle u, c \rangle) = \infty$. For $0 \leq i \leq k$, we process each level $G_i$ by keeping track of exactly $\binom{k}{i}$ costs for each node in $R$: one for each SE subgraph in level $G_i$. Before beginning each sweeping phase on a given level $G_i$, if $i > 0$ we must transfer costs from the previous search level $G_{i-1}$ via the established $E_2$ edges, according to our product graph definition. Specifically, for all $(\langle u, c \rangle, \langle v, c' \rangle) \in E_2 : |c'| = i$, we set $d(\langle v, c' \rangle) = min\{d(\langle v, c' \rangle), d(\langle u, c \rangle)\}$ to transfer the costs. We then proceed to perform an upsweep and then downsweep of $R_\phi$ (similar to before) taking care to relax all $\binom{k}{i}$ costs for each node in each sweeping phase.

**Lemma 32** *Upon completion of the upsweep and downsweep phases for level $G_i$, $d(\langle v, c \rangle) =$*

*$d(\langle s, \emptyset\rangle, \langle v, c\rangle)$ for all $\langle v, c\rangle \in R \times \mathcal{P}_i(C)$ such that $R_v^{\downarrow} \subseteq R$, where $\mathcal{P}_i(C)$ is the power set of $C$ containing only subsets of at most cardinality $i$.*

**Proof.** We prove this by induction on the level sequence $0 \leq i \leq k$. For the base case (where $i = 0$), we have that $\mathcal{P}_0(C) = \{\emptyset\}$ and thus we only need to show that $d(\langle v, \emptyset\rangle) = d(\langle s, \emptyset\rangle, \langle v, \emptyset\rangle)$ for all $\langle v, \emptyset\rangle \in R \times \{\emptyset\}$ such that $R_v^{\downarrow} \subseteq R$. This follows directly from Lemma 31, as this is equivalent to a single graph search on $R$, since we consider only product nodes within the same SE subgraph $G^{\{\emptyset\}} \simeq G'$ (i.e., $G^{\{\emptyset\}}$ is isomorphic to $G'$). For the induction step (where $i > 0$), our induction hypothesis assumes that this claim holds true for level $i - 1$. Therefore, it remains to show that the values $d(\langle v, c'\rangle)$ are correct for all $\langle v, c'\rangle \in R \times \{\mathcal{P}_i(C) \setminus \mathcal{P}_{i-1}(C)\}$ (i.e., only for the product nodes in level $G_i$) such that $R_v^{\downarrow} \subseteq R$. Since each level $G_i$ combines one or more independent SE subgraphs such that no path exists between them, and (consequently) we process the costs for each SE subgraph independently per level, then it suffices to consider only the costs within a single SE subgraph (i.e., the same logic holds for the rest, independently).

For any $c' \in \{\mathcal{P}_i(C) \setminus \mathcal{P}_{i-1}(C)\}$, let $G^{c'}$ be the SE subgraph under consideration. Let $X = \{\langle v, c'\rangle \mid \exists\, (\langle u, c\rangle, \langle v, c'\rangle) \in E_2\}$ be the set of product nodes in $G^{c'}$ which are the target of some $E_2$ edge. By definition of our product graph, any path (including a shortest path) from $\langle s, \emptyset\rangle$ to any node in $G^{c'}$ must pass through some product node in the set $X$. Therefore, if we can guarantee that $d(\langle v, c'\rangle) = d(\langle s, \emptyset\rangle, \langle v, c'\rangle)$ is correct for all $\langle v, c'\rangle \in X$ before we begin the sweeping search on $G^{c'}$, then it follows from logic similar to that presented in Lemma 31 that the sweeping search will correctly set the costs for all remaining product nodes $\langle v, c'\rangle \in R \times c'$ such that $R_v^{\downarrow} \subseteq R$.

Before sweeping $G^{c'}$, the LESS algorithm transfers costs along $E_2$ edges such that, for all $(\langle u, c\rangle, \langle v, c'\rangle) \in E_2$, $d(\langle v, c'\rangle) = min\{d(\langle v, c'\rangle), d(\langle u, c\rangle)\}$. Since $|c| = i - 1$

for all such edges, by definition, then it follows from our induction hypothesis that the values $d(\langle u, c\rangle) = d(\langle s, \emptyset\rangle, \langle u, c\rangle)$ (by definition of $E_2$, $u = c_{\ell,j}$ for some $\ell \in [1, k], j \in [1, |C_\ell|]$, and $R^{\downarrow}_{c_{\ell,j}} \subseteq R$). Then upon transfer completion, we must have that $d(\langle v, c'\rangle) = d(\langle s, \emptyset\rangle, \langle v, c'\rangle)$ is correct for these nodes as well (since we have taken the minimum over all adjacent $\langle u, c\rangle$ costs, and any path from $\langle s, \emptyset\rangle$ must pass through one such node). ■

**Corollary 33** *Upon completion of the upsweep and downsweep phases for level $G_k$, $d(\langle t, C\rangle) = d(\langle s, \emptyset\rangle, \langle t, C\rangle)$ represents the optimal GTSPP solution cost.*

**Theorem 34** *The LESS algorithm runs in $O(2^k(m' + nk))$ time, where $m' = |E \cup E'|$.*

**Proof.** Establishing the search space $R$ and its topological ordering $R_\phi$ can be done in time $O(m' + n)$ using a variant of depth-first search (which only searches upward in node ranking). The product graph has exactly $2^k n$ nodes and at most $2^k(m' + \frac{kg}{2}) \in O(2^k(m' + nk))$ edges. In the worst case, $R = V$ and we must sweep the entire product graph. As the sweep is linear in the size of the product graph, this gives a total of $O(2^k(m' + nk))$ time. ■

**Pruning.** While our current solution is correct, its relative performance is expected to be highly sensitive to the density of a given GTSPP query. This is because our runtimes for this algorithm are directly proportional to the size of our search space, $R$, which typically grows in size proportional to the density, $g$, of the GTSPP instance under consideration (i.e., more distinct locations make for larger unioned search spaces). This expected behavior suggests that, before the search, it could be beneficial for us to try and prune any locations which we determine cannot possibly belong to any optimal GTSPP solution, in order to minimize the overall search space size and resulting runtime.

To achieve this, we require a method for estimating the cost of a solution which contains a given category node. A straightforward and fast approach is to utilize a constant-time heuristic function, $h : V \times V \rightarrow \mathbb{R}_{\geq 0}$, which returns a non-negative estimate on the shortest-path cost between any two nodes. We only require that the heuristic function be *admissible*, such that $h(s,t) \leq d(s,t)$ for all $s,t \in V$ (i.e., it must always underestimate the shortest-path cost).

Given any admissible heuristic function, $h$, we must first establish an upper bound, $\mu$, on the optimal GTSPP solution cost $w(P^C_{s,t})$. Starting at node $x_0 = s$ and ending at node $x_{k+1} = t$, for $1 \leq i \leq k$, we apply a greedy nearest-neighbor strategy to select a node $x_i = \underset{\forall c_{i,j} \in C_i}{argmin}\{h(x_{i-1}, c_{i,j})\}$, giving us the resulting node sequence $\langle x_0, x_1, \ldots, x_k, x_{k+1}\rangle$. We then compute the value $\mu = \sum_{0 \leq i \leq k} d(x_i, x_{i+1})$ by performing $(k+1)$ fast, point-to-point shortest path queries in the CH search graph (using the traditional bidirectional Dijkstra search). Since, by definition, the path established by the node sequence $\langle x_0, x_1, \ldots, x_k, x_{k+1}\rangle$ is a *satisfying* path for the instance $\langle s, t, C\rangle$, then $\mu$ is also therefore a valid upper bound on $w(P^C_{s,t})$.

We now prune each category, $C_i$, as follows. For each $c_{i,j} \in C_i$, if $h(s, c_{i,j}) + h(c_{i,j}, t) > \mu$, we remove node $c_{i,j}$ from $C_i$. This is because the value $h(s,v) + h(v,t)$ is a valid lower bound on any GTSPP solution which contains node $v$. After pruning, we may then carry out the proposed LESS algorithm, as before.

## 6.5 Experiments

In this section, we present experiments highlighting the performance characteristics of our proposed GTSPP algorithms. Specifically, we examine the performance impacts of our two primary measures of interest regarding GTSPP complexity: category density ($g$) and the number of categories ($k$). For each experiment, we consider four product-graph search algorithms for comparison: unidirectional Dijkstra search (**U. Dijkstra**), bidirectional Dijkstra search (**B. Dijkstra**), CH-based Level-Sweeping Search (**LESS**), and LESS with Pruning (**P-LESS**).

For each experiment, we further consider two variants of GTSPP queries to model two possible extremes of locality: **non-local queries** in which $s \neq t$ are examined to model relatively long-distance routes and **local queries** in which $s = t$ are examined to model relatively short-distance routes.

### 6.5.1 Test Environment

All experiments were carried out on a 64-bit server machine running Linux CentOS 5.3 with 2 quad-core CPUs clocked at 2.53 GHz with 18 GB RAM (only one core was used per experiment). All programs were written in C++ and compiled using gcc version 4.1.2 with optimization level 3.

### 6.5.2 Test Dataset

All experiments were performed on the road network of North America[4], with $21,133,774$ nodes and $52,523,592$ edges. The weight function, $w$, is based on travel time (in minutes). This dataset was derived from NAVTEQ data products, under their permission. We have chosen the Pre-Computed Cluster Distances (PCD) heuristic function from [82] as our pruning function, $h$. In brief, PCD partitions the graph into $r$ partitions and computes an $r \times r$ cost matrix of the shortest path costs between the closest nodes from each pair of partitions. The heuristic function $h(u, v)$ returns the matrix value between the partitions of $u$ and $v$, which is a lower bound on $d(u, v)$. For our experiments, we have chosen $r = 10,000$. PCD preprocessing required 7 minutes using

[4]This includes only the US and Canada.

the CH search graph, resulting in an overhead of 23 bytes per node. CH preprocessing required 18 minutes, resulting in an overhead of 35 bytes per node.

### 6.5.3 Category Density Experiments

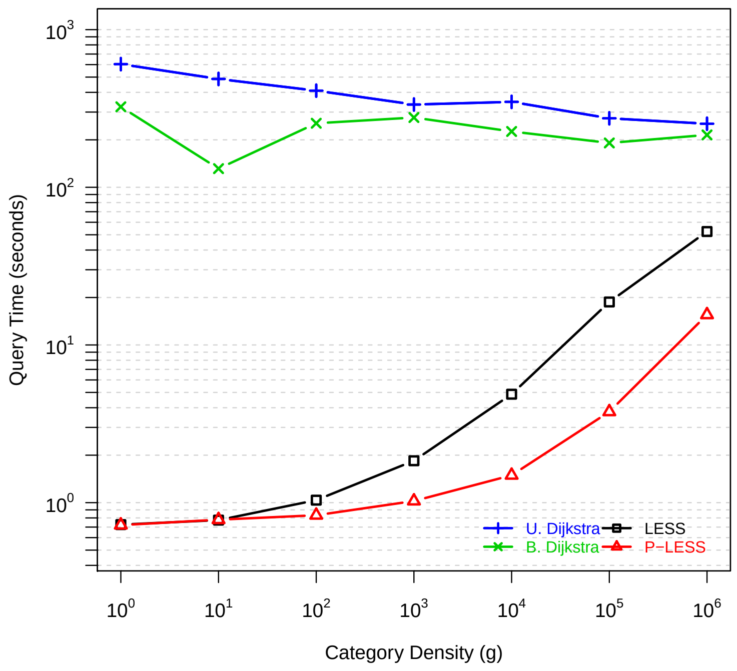
We begin by examining the impact of category density, $g$, on the performance of our proposed algorithms. In Fig. 6.5(a) and Fig. 6.5(b) we present the results of our four algorithms for both non-local and local queries, respectively.

For all experiments in this section, we fixed the category count at $k = 5$. For every $0 \leq i \leq 6$, we constructed 100 random query instances in which each of the 5 categories were populated with $g = 10^i$ nodes selected uniformly at random. All source and target nodes, $s$ and $t$, were additionally selected uniformly at random. The numbers presented in the figures represent average query times.

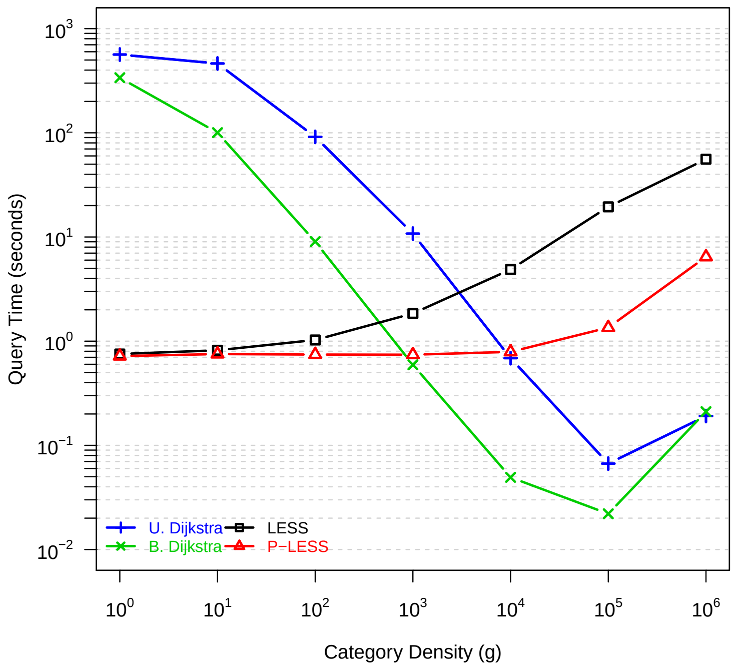
Starting with the non-local experiments of Fig. 6.5(a), we see that both Dijkstra variants have very high runtimes across all densities, but tend to improve as the density increases. B. Dijkstra is consistently faster than U. Dijkstra by a factor of 1.8, on average. However, it requires over 130 seconds, even in the best case.

In contrast, our advanced LESS algorithm can be seen to perform extremely well for low-density scenarios, but (as expected), begins to degrade as the density increases. Despite this degradation, it still outperforms the best Dijkstra algorithm by a factor of over two orders of magnitude, on average. The P-LESS approach reduces the runtimes even further, showing an additional 41% improvement, on average, over the unpruned LESS algorithm for these non-local queries, and requiring no more than 16 seconds in the worst case (for $g = 1,000,000$).

The story is slightly different, however, for the local query cases in Fig. 6.5(b).



(a) $s \neq t, k = 5$



(b) $s = t, k = 5$

Figure 6.5: Category Density Experiments for (a) non-local queries and (b) local queries.

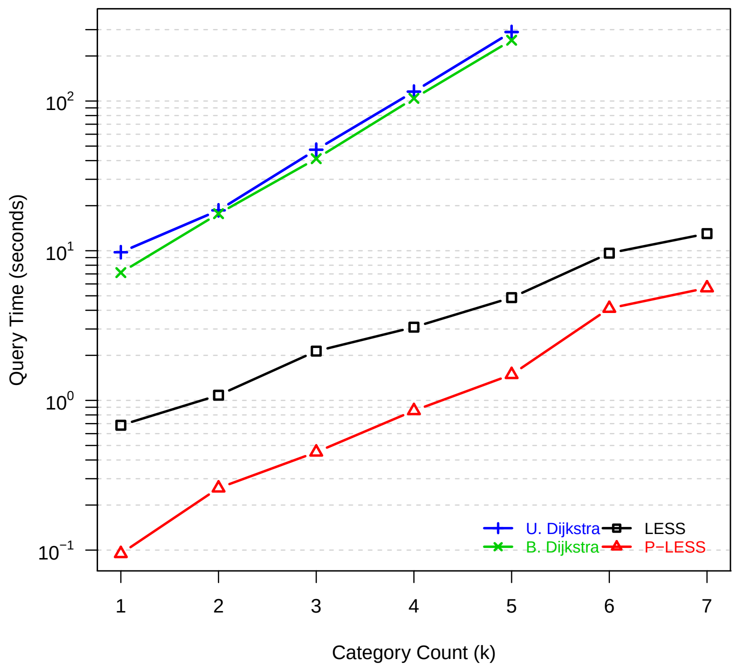
Here we see the Dijkstra algorithms show a much more significant overall improvement as the density increases (B. Dijkstra is less than 1 second for cases in which $g \geq 1,000$). This is due primarily to the fact that, as the density grows, so do the number of available zero-cost $E_2$ edges in the product graph. For such extremely-local cases (e.g., where $s = t$), this greatly benefits greedy search algorithms such as the Dijkstra variants which can then quickly transition along these increasingly-available $E_2$ edges to arrive at the nearby target in the final level. Alternatively, this scenario presents a slight disadvantage for both LESS-based strategies, which are not greedy by nature, but are instead forced to progress through the product graph search one level at a time, regardless of locality or density. Despite this, P-LESS is still the only algorithm which requires no more than 7 seconds across all of the densities tested here for such cases.

Furthermore, for such extremely local cases, we also see a marked improvement in our P-LESS approach over the unpruned LESS algorithm (e.g., pruning gives nearly an order of magnitude speed improvement for the highest-density scenario). This is anticipated from the fact that local cases are expected to have much smaller $\mu$ values, especially for high densities, leading to greater pruning.

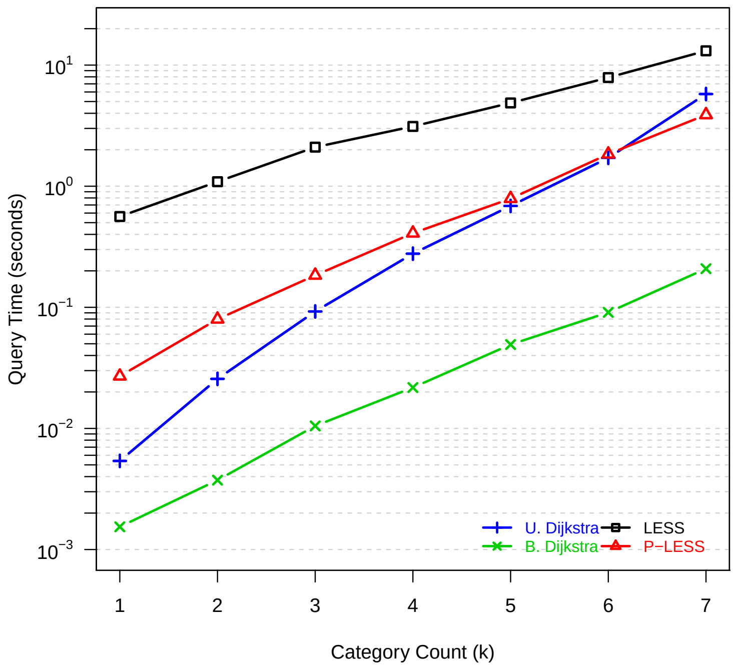
### 6.5.4 Category Count Experiments

Next we examine the impact of the number of categories, $k$, on the performance of our algorithms. In Fig. 6.6(a) and Fig. 6.6(b) we present the results for both non-local and local queries, respectively (showing average query times, as before).

For all experiments in this section, we fixed the category density at $g = 10,000$. For every $1 \leq i \leq 7$, we constructed 100 random query instances, each with $i$ categories populated with $g$ nodes selected uniformly at random. All source and target nodes, $s$



(a) $s \neq t, g = 10,000$



(b) $s = t, g = 10,000$

Figure 6.6: Category Count Experiments for (a) non-local queries and (b) local queries.

and $t$, were additionally selected uniformly at random.

Again, we start by reviewing the non-local query results in Fig. 6.6(a). As before, for non-local queries, the Dijkstra algorithms perform the worst overall, reaching nearly 300-second query times. Additionally, they are unable to even complete for cases with $k \geq 6$, due to memory exhaustion from such increasingly-large search spaces. Our LESS-based algorithms show similar improvements as before for such non-local queries, and appear to scale quite well across these experiments, with the P-LESS algorithm requiring no more than 6 seconds for even the largest number of categories at $k = 7$.

For the local query results in Fig. 6.6(b), we see similar improvements for the Dijkstra search algorithms as before, with the B. Dijkstra algorithm requiring only up to 0.2 seconds in the worst case. This gives over an order of magnitude improvement, on average, compared to the U. Dijkstra search. Comparatively, our LESS-based algorithms perform the worst overall for these local scenarios, suggesting a similar pattern to that from the previous experiments, in which the greedy algorithms perform best for local queries, but worst for non-local queries.

Across both sets of experiments, the P-LESS algorithm again provides significant improvements over LESS, although its relative speedups appear to degrade with increasing category counts (going from a speedup of over 7 for $k = 1$ down to just over 2 for $k = 7$ for non-local queries, and over 20 down to just over 3 for local queries). As our current heuristic estimates for pruning are most accurate for fewer categories, this suggests that a more accurate heuristic may lead to further improvements over queries with larger numbers of categories.

## 6.6 Conclusion

We have demonstrated a new algorithmic framework based on a unique product-graph formulation, which allows us to solve real-world, large-scale GTSPP instances using various graph search algorithms. Our proposed algorithms are able to efficiently solve such problem instances to optimality, typically in a matter of seconds. These algorithms may also be used interchangeably, as needed, based on their respective performance advantages across various problem sizes and scales of locality. Specifically, our results suggest that, for highly-local, very-dense queries, a greedy Dijkstra search in our proposed product graph may be sufficient and effective in practice (and does not require preprocessing). However, for more consistently-efficient performance over longer distances and for various problem sizes, our proposed LESS algorithm (with pruning) is justifiably better.

Several promising areas of further research include pursuing more-aggressive pruning strategies and incorporating goal-directed search techniques (e.g., $A^*$ search). We address these particular possibilities in the following chapter. Additionally, since each SE subgraph cost in a given level may be processed independently, this suggests that our level-sweeping search algorithm may further lend itself to strong parallelization, similar to that achieved in [34].

# Chapter 7

# Generalized Traveling Salesman Paths with Preprocessing

## 7.1 Introduction

For many travelers, running errands while “on the go” is a common occurrence for everyday trip planning. For example, a traveler sightseeing in a new city might wish to visit an ATM, a gift shop, and a coffee shop along the way to their destination. Other common errands include, e.g., getting groceries, refueling the car, dropping off mail, picking up dinner, etc. However, for most of these activities there may also be many valid options to choose from on a given trip (e.g., according to U.S. Census numbers, there are over 500,000 food-service locations and over 100,000 gas stations in the U.S. alone). Because the traveler often wishes to complete these errands in the least amount of overall travel distance or time on the way to their destination, determining which of each of these possible errand locations to visit and the order in which to visit them is a fundamental part of the trip planning process. Such trip planning problems arise frequently within the logistics industry as well. For example, many long-haul trucking scenarios require multiple days of traveling long distances, presenting a recurrent need for efficiently-planned stops for refueling, lodging, dining, etc. throughout the trip (each of which presents many options).

We formalize such problems as a variant of the Generalized Traveling Salesman Problem (GTSP), also commonly known as Errand Scheduling, Group TSP, Set TSP, One-of-a-Set TSP, Multiple-Choice TSP, and TSP with Neighborhoods. GTSP is a generalization of the well-known Traveling Salesman Problem (TSP) whereby a set of nodes (e.g., errand locations) are partitioned into groups, or categories (e.g., based on the type of service provided at each location), and the goal is to find a minimum-cost tour that visits exactly one node from each category. In general, GTSP is known to be NP-hard, by reduction from the standard TSP, in which each node merely defines its own category. GTSP variants having a fixed source location, $s$, and a fixed target location, $t$, such as the navigation problems we are interested in here, are referred to as Generalized Traveling Salesman *Path* Problems (GTSPP)[96]. As the more-general GTSP (with no fixed source or target) may be solved using any algorithm for the more-specific GTSPP[1], we will focus only on the GTSPP in the remainder of our discussion.

In this chapter, we present advanced algorithms for solving large-scale GTSPP queries on real-world road networks. In Section 7.2, we formalize GTSPP, including a brief discussion of related work. Section 7.3 details an $O(r)$-approximation algorithm for GTSPP, for $r \in \mathbb{N}$. Section 7.4 presents an alternative, and generally more-practical, $(1+\epsilon)$-approximation algorithm for GTSPP, $\forall \epsilon \in \mathbb{R}_{\geq 0}$. Section 7.5 presents experimental results for our $(1+\epsilon)$-approximation algorithm on the road network of North America. Section 7.6 concludes the chapter with further possible extensions to this problem.

[1]For each node $x$ in the category with the fewest nodes, solve the GTSPP with $s = t = x$, and choose the minimum-cost solution over all $x$ as the GTSP solution.

## 7.2 Generalized Traveling Salesman Path Problems

Let $G = (V, E, w)$ be a weighted, directed graph, with node set $V$, edge set $E \subseteq V \times V$, and edge weight function $w : E \rightarrow \mathbb{R}_{>0}$, such that $n = |V|$ and $m = |E|$. Let $P_{s,t} = \langle v_1, v_2, \ldots, v_q \rangle$ be any path in $G$ from $s = v_1 \in V$ to $t = v_q \in V$, such that, for $1 \leq i < q$, $(v_i, v_{i+1}) \in E$. The total weight, or cost, of any $P_{s,t}$ is $w(P_{s,t}) = \sum_{1 \leq i < q} w(v_i, v_{i+1})$. Let $P^*_{s,t}$ be any minimum-weight, or "shortest", path from $s$ to $t$. The shortest path "distance" is referenced as $d(s,t) = w(P^*_{s,t})$.

A category set, $C = \{C_1, C_2, \ldots, C_k\}$, defines a set of generalized node subsets within the graph where, for $1 \leq i \leq k$, $C_i = \{c_{i,1}, c_{i,2}, \ldots, c_{i,|C_i|}\} \subseteq V$ represents a distinct category of locations, each of which provides the same general type of service (e.g., the set of gas station locations). Note that $V \setminus (C_1 \cup C_2 \cup \ldots \cup C_k) \neq \emptyset$, in general (i.e., not every graph node has to belong to a category). The primary structural measures of a category set are the *category count*, $k = |C|$, representing the total number of distinct categories, and the *category density*, $g = \underset{1 \leq i \leq k}{max} \{|C_i|\}$, representing the maximum number of optional locations per category. Thus, the total number of category locations $||C|| = \sum_{1 \leq i \leq k} |C_i|$ is upper-bounded by $O(kg)$.

A path, $P_{s,t}$, is said to be a *satisfying path* for a category set $C$ iff, for $1 \leq i \leq k$, $P_{s,t} \cap C_i \neq \emptyset$ (i.e., $P_{s,t}$ contains at least one node from each category). Therefore, for any GTSPP instance $I = \langle G, s, t, C \rangle$, for some $s, t \in V(G)$ and category set $C$ (defined on $G$), we seek to compute a *minimum-weight satisfying path*, referenced as $P^C_{s,t}$ (see Fig. 7.1 for an example).

For a given category set, $C$, the GTSPP product graph, introduced in [96], is defined as $G_C = (V \times \mathcal{P}(C), E_1 \cup E_2)$, with product nodes $\langle u, c \rangle$ such that $u \in V$ and

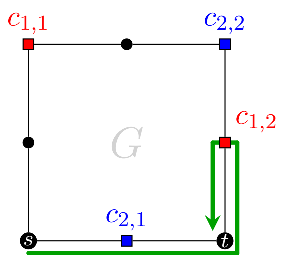


Figure 7.1: Example GTSPP instance (taken from [96]) for a graph $G$ (with unit-cost edge weights), $s, t \in V$, and category set $C = \{C_1, C_2\}$. The minimum-weight satisfying path is shown in green.

$c \in \mathcal{P}(C)^2$, where $E_1 = \{(\langle u, c\rangle, \langle v, c'\rangle) \mid c = c' \wedge (u, v) \in E\}$ and $E_2 = \{(\langle u, c\rangle, \langle v, c'\rangle) \mid u = v \wedge c' \setminus c = C_i \in C \wedge v \in C_i\}$.

The $E_1$ edges are structured to represent a unique copy of each original edge from $G$ for every subset of $C$. For all $(\langle u, c\rangle, \langle v, c'\rangle) \in E_1$, we define $w(\langle u, c\rangle, \langle v, c'\rangle) = w(u, v)$. The $E_2$ edges are structured to represent the accumulation of a new category by inclusion of a specific node from within that category. For all $(\langle u, c\rangle, \langle v, c'\rangle) \in E_2$, we define $w(\langle u, c\rangle, \langle v, c'\rangle) = 0$. Any path from $\langle s, \emptyset\rangle$ to $\langle t, C\rangle$ in $G_C$ therefore represents a valid satisfying path in the original graph, based on a specific accumulation sequence of category nodes from each category. As shown in [96], the shortest path from $\langle s, \emptyset\rangle$ to $\langle t, C\rangle$ in $G_C$ thus represents an equivalent-cost, optimal solution for the GTSPP query in the original graph $G$. See Fig. 7.2 for an example product graph, based on the GTSPP instance from Fig. 7.1.

As noted in [96], the full product graph need not be explicitly constructed to perform any related GTSPP searches, but may instead be implicitly constructed, as needed, during the search. We further denote the *product subgraph* induced by any subset of nodes $V' \subseteq V$ as $G[V']_C \equiv G_C[V' \times \mathcal{P}(C)]$.

[2]$\mathcal{P}(C)$ represents the *power set* of $C$.

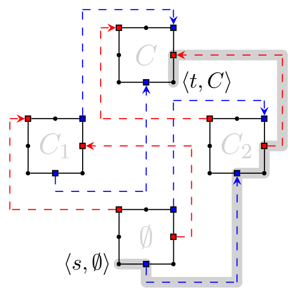


Figure 7.2: Example product graph of graph $G$ (from Fig. 7.1) for category set $C = \{C_1, C_2\}$. $E_1$ edges are shown as solid edges whereas $E_2$ edges are shown as dashed edges. The shortest path from $\langle s, \emptyset\rangle$ to $\langle t, C\rangle$ is shown in grey.

### 7.2.1 Related Work

GTSP was initially examined in [63, 110] and has since been approached via various algorithmic strategies. Dynamic programming formulations for GTSP have been explored in [63, 100, 110]. Integer- and linear-programming techniques for GTSP are presented in [46, 73, 74, 87]. Algorithms for transforming GTSP instances into standard TSP instances with $O(kg)$ nodes are given in [24, 40, 80]. Heuristics have been proposed for this problem type in [30, 109]. Approximation algorithms for GTSP have also been considered in [11, 44, 58, 79, 81, 97, 108]. In general, however, (even metric) GTSP cannot be approximated to within any constant factor in polynomial time unless P = NP [99].

In the field of parameterized complexity theory [41, 48], a problem is said to be *fixed parameter tractable* (FPT) if there exists an algorithm for solving the problem in time $f(k)n^{O(1)}$ for some arbitrary function $f$ (independent of the problem size, $n$) with respect to some problem-specific parameter $k$ (e.g., solution size, structural properties such as treewidth, or, as in our case, the number of distinct categories). Such algorithms help confine the true combinatorial explosion of the problem to the parameter $k$, which is hopefully small (and thus, manageable) in practice (even if $n$ is very large).

GTSPP may be solved to optimality using FPT algorithms with $f(k) \in O^*(k!)$[3] [92, 97], requiring only polynomial space. Exact FPT algorithms with $f(k) \in O^*(2^k)$ have also been considered in [96], by way of using the GTSPP product graph introduced earlier. Such algorithmic approaches turn out to be quite practical for many real-world GTSPP scenarios in personal navigation, as the number of optional locations to consider may be quite large (e.g., $g = 100,000$), but the number of "errands" (categories) for a given trip are typically quite small (e.g., $k = 3$).

### 7.2.2 Our Contributions

In the following sections, we examine new approximation algorithms which allow us to circumvent the constant-factor inapproximability barrier of GTSP [99] by considering efficient FPT algorithms. We additionally improve upon the previous exact algorithms from [96] to achieve optimal GTSPP solutions on real-world road networks in near real-time (i.e., sub-second times).

## 7.3 An $O(r)$-Approximation Algorithm for GTSPP

Our first approximation algorithm is a simple "divide-and-approximate" strategy for obtaining constant-factor approximations. Specifically, for any user-defined integer parameter $1 \leq r \leq k$, we show how to compute an $O(r)$-approximate solution in time $O^*(2^{k/r})$. The approximation algorithm workflow is outlined below (including an example given in Fig. 7.3, for illustration):

1. Given an integer $r \in [1, k]$, **partition $C$ into $r$ disjoint category subsets**

[3]The $O^*$-notation omits any polynomial factors. Only small polynomials (e.g., similar to those in [96]) are omitted here.

$C^{(1)}, \ldots, C^{(r)}$ such that each subset has at most $\lceil k/r \rceil$ categories each (some subsets may have fewer categories than others, due to uneven partitioning).

2. **Solve each sub-problem** using an exact GTSPP algorithm (e.g., [96]).

3. **Combine the resulting paths** by chaining the sub-solutions together as follows. Let $c^i_\alpha$ and $c^i_\omega$ be the first and last category nodes visited along the solution for sub-problem $C^{(i)}$, respectively. For $1 \leq i < r$, remove the subpath $P' = \langle c^i_\omega, \ldots, t \rangle$ from the solution for $C^{(i)}$, remove the subpath $P'' = \langle s, \ldots, c^{i+1}_\alpha \rangle$ from the solution for $C^{(i+1)}$, and replace them with the shortest path from $c^i_\omega$ to $c^{i+1}_\alpha$. This "connecting" process leads to a single, satisfying-path solution for the original problem instance (e.g., see Fig. 7.3).

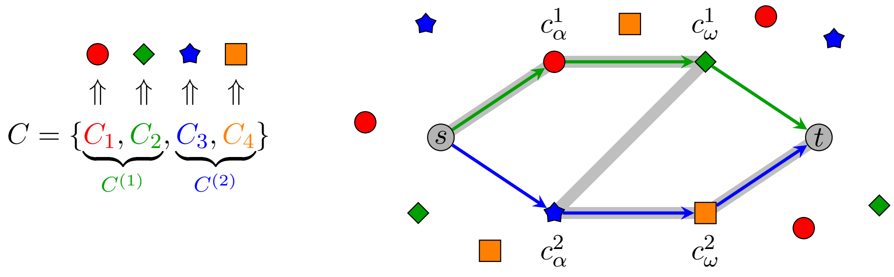


Figure 7.3: An example (Euclidean) GTSPP instance with $k = 4$ and $r = 2$. The $C^{(1)}$ sub-problem solution is shown in green, the $C^{(2)}$ sub-problem solution is shown in blue, and the final connected solution path is shown in grey.

**Theorem 35** *On undirected graphs (or directed graphs with bounded asymmetry*[4]*), the above algorithm computes an $O(r)$-approximate solution in time $O^*(2^{k/r})$.*

**Proof.** Steps 1 and 3 require polynomial time in the size of the category sets and the underlying graph, respectively. Step 2 requires that we solve $r$ smaller GTSPP instances

[4]Graphs $G = (V, E)$ such that $\forall s, t \in V$, $d(s, t) \leq c \cdot d(t, s)$, for some constant $c$.

with $\leq \lceil k/r \rceil$ categories each, requiring a total of $r \cdot O^*(2^{k/r})$ time in total (e.g., if using the FPT algorithms from [96]), thus proving our suggested total time bound of $O^*(2^{k/r})$.

To prove the $O(r)$-approximation bound requires that we first examine an alternate, more primitive form of concatenation of the sub-problem solution paths. Namely, suppose instead of the chaining rule presented in Step 3, that we combine all $r$ sub-problem solution paths by simply inserting between each consecutive path the shortest path from $t$ (i.e., the end of one sub-solution) back to $s$ (i.e., the beginning of the next sub-solution) to connect them. This gives a satisfying path, $\tilde{P}^C_{s,t}$, for the original (full) problem, and gives the following bounds (the inequalities below are only valid for undirected graphs but may easily be modified for directed graphs of bounded asymmetry):

$$\begin{aligned}
w(\tilde{P}^C_{s,t}) &= w(P^{C^{(1)}}_{s,t}) + d(t,s) + w(P^{C^{(2)}}_{s,t}) + \ldots + \\
&\quad\; d(t,s) + w(P^{C^{(r)}}_{s,t}) \\
&= \sum_{1 \leq i \leq r} w(P^{C^{(i)}}_{s,t}) + (r-1) \cdot d(t,s) \\
&\leq r \cdot w(P^C_{s,t}) + (r-1) \cdot d(s,t) \\
&\leq (2r-1) \cdot w(P^C_{s,t})
\end{aligned}$$

Since the solution path formed by the original chaining rule (in Step 3) is merely an appropriately-shortcutted instance of $\tilde{P}^C_{s,t}$, then by the triangle-inequality, the approximation bounds of $O(r) \cdot w(P^C_{s,t})$ are maintained. ■

Note that this allows for a very nice and highly-flexible tradeoff between computation time and approximation. For example, using this approach, we can derive several useful high-level approximation forms:

1. It is known that metric GTSPP cannot be efficiently approximated to within a constant factor unless P = NP [99]. However, using this reduction scheme allows us

to compute constant-factor approximate solutions in only moderately-exponential FPT time; e.g., $r = 2$ gives an $O(1)$-approximation in time $O^*(1.415^k)$, $r = 3$ in time $O^*(1.260^k)$, $r = 4$ in time $O^*(1.190^k)$, etc.

2. For additional parameters $\alpha > 0$ and $\beta \geq 0$, we can also compute $\Theta(\frac{k}{\alpha log^{\beta} n})$-approximate solutions in either quasi-polynomial time (for $\beta > 1$), or in polynomial time (for $\beta \leq 1$), improving upon the previous polynomial-time, $O(k)$-approximation algorithms from [79, 97] by up to a logarithmic factor. Additionally, this approach can be shown to subsume the Minimum-Distance (MD) approximation algorithm from [79], as MD can be seen as a worst-case special instance of our proposed approximation strategy for $\alpha = 1$, $\beta = 0$, and thus, $r = k$.

While theoretically interesting due to its asymptotic runtime improvements, this approach unfortunately can still be quite poor in practice, given the nature of its construction. Specifically, for long-distance queries (where $s$ and $t$ are far apart), the "connecting" paths needed to connect sub-solutions into one valid solution for the original problem may result in too much detour for practical cases (since we keep having to "go back and forth" to connect each sub-problem solution). This can result in solutions whose total path costs become quite close to the worst-case $O(r)$-approximation bound in practice (a bound which is also provably tight). Likewise, the approximation bounds only hold for either undirected graphs or for graphs with bounded asymmetry, so at least some level of symmetry is required in the graph. Therefore, we wish to find a more practically-relevant approximation algorithm for this problem. In the remainder of this chapter, we focus our efforts on establishing such a practical approximation algorithm.

## 7.4 A $(1+\epsilon)$-Approximation Algorithm for GTSPP

In this section, we extend the previous work from [96] to produce a more efficient, and adjustable, $(1+\epsilon)$-approximation algorithm for GTSPP (for all $\epsilon \geq 0$). The work from [96] incorporates concepts from the graph pre-processing technique known as Contraction Hierarchies (CH) [53]. CH pre-processing ranks the graph nodes, $\phi : V \rightarrow \{1, \ldots, n\}$, according to some (heuristic) measure of importance and then *shortcuts* the nodes in increasing rank order. Shortcutting a node, $v$, considers each pair of incoming and outgoing edges, $(u, v)$ and $(v, x)$, from and to higher-ranking nodes, respectively (i.e., $\phi(v) < min\{\phi(u), \phi(x)\}$). If the path $\langle u, v, x\rangle$ is a unique shortest path, then a shortcut edge $(u, x)$ is added with weight $w(u, v)+w(v, x)$ to preserve shortest path costs in the higher-ranking subgraph $G[\{z \in V \mid \phi(z) > \phi(v)\}]$. The result is a new graph $G' = (V, E \cup E', w)$, where $E'$ represents the newly-constructed shortcut edges. We define the "upward" graph as $G^{\uparrow} = (V, E^{\uparrow}, w)$ where $E^{\uparrow} = \{(u, v) \in E \cup E' \mid \phi(u) < \phi(v)\}$ and the "downward" graph as $G^{\downarrow} = (V, E^{\downarrow}, w)$ where $E^{\downarrow} = \{(u, v) \in E \cup E' \mid \phi(u) > \phi(v)\}$. Let $R_s^{\uparrow} = \{v \in V \mid \exists P_{s,v} \subseteq G^{\uparrow}\}$ be the set of all nodes reachable from $s$ in the upward graph and $R_t^{\downarrow} = \{v \in V \mid \exists P_{v,t} \subseteq G^{\downarrow}\}$ be the set of all nodes from which $t$ is reachable in the downward graph.

As shown in [96], hierarchical GTSPP search algorithms using a pre-processed CH graph need only focus their search on the set of all "upward-reachable" nodes from $s$, $t$, and $C$. More specifically, according to the structural properties of the CH graph, between any pair of nodes for which there is a path, after pre-processing there must exist a *weakly-bitonic* shortest path $P = \langle v_1, \ldots, v_q\rangle$, such that $\exists\, v_i \in P$, where $\phi(v_1) < \ldots < \phi(v_{i-1}) < \phi(v_i)$ (i.e., node rank strictly increases up to node $v_i$) and $\phi(v_i) > \phi(v_{i+1}) > \ldots > \phi(v_q)$ (i.e., node rank strictly decreases after node $v_i$). Note that $v_i = v_1$ or $v_i = v_q$

may be true, hence the term *weakly* bitonic. Any such weakly-bitonic paths between $s$, $t$, and members of $C$ are, by definition, confined to the set $R = R_s^{\uparrow} \cup R_C \cup R_t^{\downarrow}$, where $R_C = \bigcup_{i=1}^{k} \bigcup_{j=1}^{|C_i|} \{R_{c_{i,j}}^{\downarrow} \cup R_{c_{i,j}}^{\uparrow}\}$ (see Fig. 7.4). Therefore, it suffices to explore only the induced product subgraph $G[R]_C$ for correctness. The proposed Level-Sweeping Search (LESS) algorithm from [96] was thus designed around a non-greedy, iterative "sweeping" process, intended to exploit this and other structural properties of the product graph.

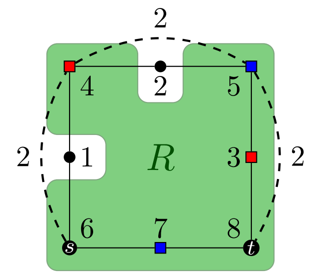


Figure 7.4: Example graph (based on Fig. 7.1) after CH processing. Each node, $v$, is labeled with $\phi(v)$. Shortcut edges are shown as dashed edges, labeled with their weight. The set $R$ is shown in green.

We introduce here several improvements to the hierarchical GTSPP search algorithm from [96]:

1. The *non-greedy* approach of the hierarchical algorithm from [96] forces the search to examine *every* node in $G[R]_C$, regardless of query locality. To further reduce the overhead of the search algorithm, we instead consider a *greedy, goal-directed* A$^*$ search [62] in $G[R]_C$ so that we may (ideally) explore only those nodes which are necessary to ensure correctness (see Section 7.4.1).

2. For many GTSPP applications, the set of optional categories may be fixed for the lifetime of the application (e.g., many common category definitions for navigation,

such as gas station locations, are fairly static and may thus be pre-defined[5]). Assuming a fixed category set $C$ (in which queries may only utilize subsets of categories from $C$), we propose to offload the computation of the set $R_C$, as well as data required for enhancing our A$^*$ search, into a separate pre-processing phase to speedup subsequent queries (see Section 7.4.2).

3. For high-density problems, $G[R]_C$ may still be too large for practical search algorithms to explore in real-time. To alleviate this issue, we present an adjustable search-space pruning strategy which allows us to efficiently approximate the optimal solution to within any arbitrary constant factor (see Section 7.4.3).

### 7.4.1 A$^*$ Search in the Product Graph

A$^*$ search [62] is a well-known graph search algorithm which assigns each reached node, $v$, a value $f(v) = d(v) + h(v)$, where $d(v)$ represents the best-known path cost from $s$ to $v$, and $h(v)$, called the *heuristic function*, represents an estimate on the shortest-path cost from $v$ to $t$. Initially, for all $v \in V$, $d(v) = \infty$. A$^*$ begins by assigning $d(s) = 0$ and inserting $s$ into a set $F$ (the "fringe" of the search). At each iteration, the search removes from $F$ a node, $u$, with minimum $f$-value and *expands* $u$ as follows. For all $e = (u, v) \in E$, if $d(v) > d(u) + w(e)$, the algorithm sets $d(v) = d(u) + w(e)$ and $F = F \cup \{v\}$. When node $t$ is removed from $F$, the value $d(t)$ is returned. A$^*$ will return the correct shortest-path cost if the function $h$ is *admissible*: $\forall v \in V$, $h(v) \leq d(v,t)$ (i.e., $h$ never overestimates the true shortest-path cost). A$^*$ will expand each node at most once in the search if the function $h$ is *consistent*: $\forall (u, v) \in E$, $h(u) \leq w(u, v) + h(v)$. Bidirectional A$^*$ search [91] extends this approach by performing two simultaneous A$^*$

[5]Categories may be made more specific, as necessary; e.g., consider only gas stations of a certain brand.

searches: one forward search from $s$ (towards goal node $t$) and one backward[6] search from $t$ (towards goal node $s$). We shall distinguish all $f$, $d$, $h$, and $F$ values by the source node for each respective search (e.g., $h_s$ for the search from $s$ and $h_t$ for the search from $t$). The bidirectional search alternates expanding each search direction, keeping track of $\gamma = min\{d_s(v) + d_t(v)\}$ for any node $v$ expanded by both search directions. The search may terminate once $\gamma \leq max\{min\{f_s(v) \mid v \in F_s\}, min\{f_t(v) \mid v \in F_t\}\}$, ensuring $\gamma = d(s,t)$ [91].

To apply A* search within our GTSPP product graph, we must therefore establish heuristic functions $h_{\langle s,\emptyset\rangle}$ and $h_{\langle t,C\rangle}$ for the forward and backward search, respectively. For each node $v \in V$ and each category $C_i \in C$, we first define the values $\pi_i(v) = min\{d(v, c_{i,j}) \mid c_{i,j} \in C_i\}$ (i.e., the cost to reach the nearest member of $C_i$) and $\pi'_i(v) = min\{d(c_{i,j}, v) \mid c_{i,j} \in C_i\}$ (i.e., for the reverse case). We demonstrate how to efficiently pre-compute the values for each function $\pi_i$ and $\pi'_i$ in Section 7.4.2. Using these definitions, we establish our heuristic functions as:

- $h_{\langle s,\emptyset\rangle}(\langle v, c\rangle) = max\{d(v,t), max\{\pi_i(v) + \pi'_i(t) \mid C_i \in C \setminus c\}\}$
- $h_{\langle t,C\rangle}(\langle v, c\rangle) = max\{d(s,v), max\{\pi_i(s) + \pi'_i(v) \mid C_i \in c\}\}$

We discuss how to efficiently obtain the shortest path costs $d(v,t)$ and $d(s,v)$ for each reached node $v$, as used in our heuristic function definitions above, later in Section 7.4.3.

**Theorem 36** *Heuristic functions $h_{\langle s,\emptyset\rangle}$ and $h_{\langle t,C\rangle}$ are admissible and consistent.*

**Proof.** For brevity, we shall prove this claim only for the forward search heuristic $h_{\langle s,\emptyset\rangle}$. A symmetric argument can be made for the backward search heuristic $h_{\langle t,C\rangle}$.

[6]A backward search is a standard (i.e., forward) search in the *reverse* graph $\bar{G} = (V, \bar{E})$, where $\bar{E} = \{(v,u) \mid (u,v) \in E\}$.

First, we address admissibility. For a product node $\langle v, c \rangle$, any optimal solution which contains node $v$ must travel a cost of at least the shortest path cost between $v$ and the target node $t$. Therefore, the value $d(v, t)$ is trivially admissible. Additionally, for any remaining path in the product graph from the product node $\langle v, c \rangle$ to the target product node $\langle t, C \rangle$, the set $C \setminus c$ represents the set of categories left to be visited along the path. For each category $C_i \in C \setminus c$, this means that such a path (when re-interpreted within the original graph) must eventually visit some node $c_{i,j} \in C_i$ and later terminate at node $t$, incurring a cost of at least $d(v, c_{i,j}) + d(c_{i,j}, t) \geq \pi_i(v) + \pi'_i(t)$, by definition. Since the maximum between multiple admissible heuristic values is itself also admissible, the proof of admissibility is complete.

Regarding the consistency of the function, we must consider each individual edge $(\langle u, c \rangle, \langle v, c' \rangle)$ in the product graph. Each edge belongs to either the set of $E_1$ edges or $E_2$ edges (as defined in Section 7.2). If $(\langle u, c \rangle, \langle v, c' \rangle) \in E_1$, then $(u, v) \in E \cup E'$ and we have that $d(u, t) \leq w(u, v) + d(v, t)$ by the triangle inequality (and thus the left-half of the heuristic is consistent). Additionally, we have that $c = c'$, and thus only need to ensure that $\pi_i(u) \leq w(u, v) + \pi_i(v)$ for each $C_i \in C \setminus c$, which can easily be proven by contradiction: if $\pi_i(u) > w(u, v) + \pi_i(v)$, then $\pi_i(u)$ is incorrect (a contradiction), since this would indicate we can reach a member of $C_i$ more quickly than $\pi_i(u)$ suggests by going through node $v$. Therefore, both halves of the heuristic function are consistent. Since taking the maximum of multiple consistent function values is also consistent, then this case is complete.

Alternatively, if $(\langle u, c \rangle, \langle v, c' \rangle) \in E_2$, then $u = v$, so $d(u, t) = d(v, t)$ is trivially consistent and $\pi_i(u) = \pi_i(v)$ is also consistent for all $C_i \in C \setminus c'$. Note that $c' \setminus c = C_j \in C$ (i.e., $c'$ contains one additional category), but since $u \in C_j$, then $\pi_j(u) = 0$ and thus

$\pi_j(u) + \pi'_j(t) \leq d(u,t) = d(v,t) \leq h_{\langle s,\emptyset\rangle}(\langle v,c'\rangle)$, by definition, so this additional value cannot increase the heuristic estimate for $\langle u,c\rangle$ over $\langle v,c'\rangle$. This completes the proof of consistency. ∎

### 7.4.2 Pre-Processing the Category Set

We present here several simple, but effective category-related pre-processing approaches, under the assumption that the category set of interest is fixed.

Since computing the set $R_C \subseteq R$ per query can be relatively time-consuming (especially for very dense categories), we instead pre-compute the set $R_C$ to help speedup subsequent queries in $G[R]_C$. Note that this does not require the pre-computation of $R_s^{\uparrow}$ and $R_t^{\downarrow}$ to fully complete the set $R$ (as defined in Section 7.4), as these depend on source and target locations for each query and can be quickly computed at query time (in sub-millisecond time). We can pre-compute the set $R_C$ in linear time by adding all category nodes in $C$ to a queue, and performing a breadth-first search (BFS) in $G'$, exploring only edges leading to or coming from higher-ranking nodes (effectively, an undirected BFS leading to higher nodes). The set of all nodes reached by this "upward" BFS is equivalent to $R_C$, by definition.

To supplement our A* search, we additionally pre-compute, for each node $v \in V$ and each category $C_i \in C$, the values $\pi_i(v)$ and $\pi'_i(v)$ (as defined in Section 7.4.1), as well as the values $\rho_i(v)$ such that $d(v, \rho_i(v)) = \pi_i(v)$ (i.e., $\rho_i(v)$ is the closest member of $C_i$ from $v$). Computing the values $\pi_i$ and $\rho_i$ for each category $C_i$ can be done via a simple *Voronoi-like* variant of a Dijkstra [39] search in $G'$: insert all category nodes from $C_i$ into a priority queue with a cost of zero, and carry out a backward Dijkstra [39] search from the initialized priority queue. The shortest-path cost to each node $v$ represents the value $\pi_i(v)$ and the source category node along the shortest path to $v$

represents the value $\rho_i(v)$. Values for $\pi'_i$ may be similarly computed using a forward Voronoi Dijkstra search from $C_i$. This approach takes $O(k(m + nlogn))$ time in total and requires $O(kn)$ storage space.

### 7.4.3 Δ-Corridors

As discussed previously, hierarchical algorithms need only search in the product subgraph $G[R]_C$. However, for high-density problems (expected in practice), $G[R]_C$ may still be too large for practical search algorithms to explore in real-time (as evidenced by experiments in [96]). To further alleviate this issue, we present here the concept of Δ-Corridors, based on the following intuitive notion.

For most well-defined queries in road networks, it is unlikely that a good solution path will deviate too far from the most direct, shortest path between the source and destination (e.g., if looking to refuel between LA and San Diego, it makes little sense to consider detours to gas stations in New York). One practical approach is therefore to focus the search effort instead only along some narrow corridor of interest surrounding the shortest path between the source and target locations.

This area is not only the most likely to yield an optimal detour path, but also helps significantly limit the search space for much faster queries. For any fixed $s, t \in V$, we define this so-called Δ-Corridor as $V_\Delta = \{v \in V \mid d(s, v) + d(v, t) \leq \Delta\}$ (i.e., the set of all nodes, $v$, such that the cost of the shortest detour path from $s$ to $t$ via $v$ is bounded by $\Delta$). Utilizing this concept for solving GTSPP thus involves limiting the search only to the induced subgraph $G[R \cap V_\Delta]_C \subseteq G[R]_C$, for some appropriately-defined value of $\Delta$ (see Fig. 7.5 and 7.6). We discuss how to efficiently compute $V_\Delta$ below and how to effectively select the $\Delta$ value itself in Section 7.4.4.

**Lemma 37** *$\forall v \in V_\Delta$, $\{P^*_{s,v} \cup P^*_{v,t}\} \subseteq V_\Delta$.*

**Proof.** Consider any node $v' \neq v \in P^*_{s,v}$ (a similar argument holds for the shortest path $P^*_{v,t}$). This gives us $\Delta \geq d(s,v) + d(v,t) = d(s,v') + d(v',v) + d(v,t) \geq d(s,v') + d(v',t)$, by the triangle-inequality, and thus $v' \in V_\Delta$. ■

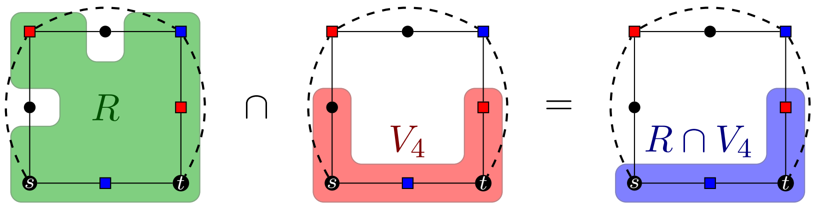


Figure 7.5: Example graph showing $R \cap V_\Delta$ for $\Delta = 4$ (based on Fig. 7.1 and Fig. 7.4).

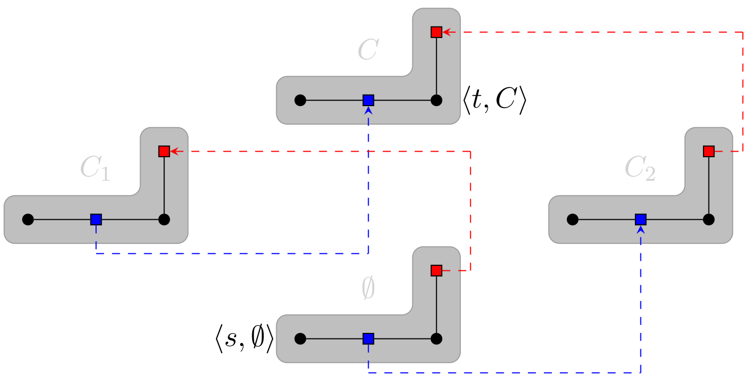


Figure 7.6: Example product graph $G[R \cap V_4]_C$ (based on Fig. 7.2 and Fig. 7.5).

**Efficiently Computing $V_\Delta$.** To compute $V_\Delta$ we begin by carrying out two $\Delta$-bounded Dijkstra [39] searches in the CH graph: one forward search from $s$ in $G^\uparrow$ and one backward search from $t$ in $G^\downarrow$. Let $d_s(v)$ and $d_t(v)$ represent the best path costs found for any reached node, $v$, by the forward and backward search, respectively. Nodes unreached by the forward (backward) search have $d_s(v) = \infty$ ($d_t(v) = \infty$). After this initial upward search phase, we begin a downward search phase by setting $Q = \{v \in R^\uparrow_s \cap R^\downarrow_t \mid d_s(v) + d_t(v) \leq \Delta\}$ and $V_\Delta = \emptyset$. While $Q \neq \emptyset$, we remove the highest-ranking node $v$ from $Q$, set $V_\Delta = V_\Delta \cup \{v\}$, and "expand downward" from $v$ as follows. For all

$(v, x) \in E \cup E'$ such that $\phi(v) > \phi(x)$, set $d_s(x) = min\{d_s(x), d_s(v) + w(v, x)\}$, and, if $x \notin Q$ and $d_s(x) + d_t(x) \leq \Delta$, set $Q = Q \cup \{x\}$. For all $(u, v) \in E \cup E'$ such that $\phi(u) < \phi(v)$, set $d_t(u) = min\{d_t(u), w(u, v) + d_t(v)\}$ and, if $u \notin Q$ and $d_s(u) + d_t(u) \leq \Delta$, set $Q = Q \cup \{u\}$.

**Theorem 38** *Upon completion of the above algorithm, we have $V_\Delta = \{v \in V \mid d(s, v) + d(v, t) \leq \Delta\}$ (i.e., $V_\Delta$ is correct), and $\forall\, v \in V_\Delta$, $d_s(v) = d(s, v)$ and $d_t(v) = d(v, t)$.*

**Proof.** Consider the set of all nodes which must belong to the $\Delta$-Corridor, by definition, arranged in sequence as $\{v_1, v_2, \ldots, v_z\}$, such that, for $1 \leq i < z$, $\phi(v_i) > \phi(v_{i+1})$ (i.e., arranged in decreasing-rank order). We must establish that *all, and only,* those nodes from this sequence will belong to the set $V_\Delta$ upon completion of the algorithm.

To establish that *only* nodes belonging to the $\Delta$-Corridor are properly included in $V_\Delta$, it suffices to observe that the values $d_s$ and $d_t$ maintain valid upper bounds on the shortest-path costs from $s$ and to $t$, respectively, throughout the search. Since a node $v$ will only be added to $Q$ (and thus, later to $V_\Delta$) once $\Delta \geq d_s(v) + d_t(v) \geq d(s, v) + d(v, t)$, then it is clear that only nodes truly belonging to the $\Delta$-Corridor will be included.

To establish that *all* nodes belonging to the $\Delta$-Corridor are properly included in $V_\Delta$, we shall prove this by induction on the decreasing-rank sequence $1 \leq i \leq z$. First, consider the base case, where $i = 1$ (i.e., $v_i = v_1$ is the highest-ranking node which belongs in the $\Delta$-Corridor).

We claim that $v_1$ must also be the highest-ranking node on the shortest paths from $s$ to $v_1$ and from $v_1$ to $t$. For otherwise, if there were a higher-ranking node $x$ on either the shortest $s$-$v_1$ or $v_1$-$t$ paths, then, by Lemma 37, $x$ must also belong to the corridor, contradicting the fact that $v_1$ is the highest-ranking corridor node.

According to the structural principles of CH graphs, if a node, $v$, is the highest-ranking node on a shortest $s$-$v$ (or $v$-$t$) path, then $d_s(v) = d(s, v)$ (or $d_t(v) = d(v, t)$) is correct after performing a Dijkstra search in $G^{\uparrow}$ (or $G^{\downarrow}$) [53]. Thus, we have that $d_s(v_1) = d(s, v_1)$ and $d_t(v_1) = d(v_1, t)$ must also be correct after the first (upward) search phase. This ensures that $v_1 \in Q = \{v \in R_s^{\uparrow} \cap R_t^{\downarrow} \mid d_s(v) + d_t(v) \leq \Delta\}$ and thus, $v_1$ will be the first to be removed from $Q$ and added to $V_\Delta$ in the second (downward) search phase, at which point its shortest-path costs are already correctly established.

For the induction step, where $i > 1$, our induction hypothesis assumes that, for all $1 \leq j < i$, $v_j \in V_\Delta$, and both $d_s(v_j) = d(s, v_j)$ and $d_t(v_j) = d(v_j, t)$ are correct by the time $v_j$ is added to $V_\Delta$. If $v_i$ is the highest-ranking node on both the shortest path from $s$ to $v_i$ and the shortest path from $v_i$ to $t$, then its cost-correctness and inclusion in $Q$ (and later, $V_\Delta$) again follow via the same logic as above. If this is not the case, consider the scenario in which $v_i$ is not the highest-ranking node on the shortest path from $s$. By our earlier definition of weakly-bitonic paths (see Section 7.4), there must exist a node $u_i$ such that $\phi(u_i) > \phi(v_i)$ and $(u_i, v_i)$ is the last edge along a weakly-bitonic shortest path from $s$ to $v_i$ (see Fig. 7.7). By Lemma 37, and our induction hypothesis, we have that $u_i \in V_\Delta$ and $d_s(u_i) = d(s, u_i)$ by the time $u_i$ is added to $V_\Delta$. Thus, when $u_i$ gets added to $V_\Delta$, edge $(u_i, v_i)$ is relaxed, correctly establishing $d_s(v_i) = d(s, u_i) + w(u_i, v_i) = d(s, v_i)$. A symmetric argument holds to ensure that $d_t(v_i) = d(v_i, t)$ is correctly established if $v_i$ is not the highest-ranking node on the shortest path from $v_i$ to $t$. Thus, $v_i$ will eventually be added to $Q$, and later to $V_\Delta$, at which point $d_s(v_i) = d(s, v_i)$ and $d_t(v_i) = d(v_i, t)$ must be correct, as we will have already processed all higher-ranking corridor nodes by this time. ■

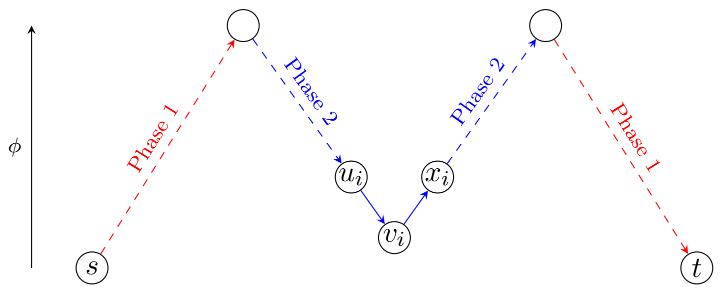


Figure 7.7: Conceptual diagram of the weakly-bitonic shortest paths computed forward from $s$ to $v_i$ and backward from $t$ to $v_i$ for some $v_i \in V_\Delta$ during the 2-Phased $\Delta$-Corridor Search. Phase 1 represents the portions of the weakly-bitonic paths computed during the first (upward) search phase. Phase 2 represents the portions of the weakly-bitonic paths computed during the second (downward) search phase.

As a corollary, since our A* search will only be exploring a product subgraph $G[R \cap V_\Delta]_C$ induced by a subset of corridor nodes, then we may use the values $d_s(v)$ and $d_t(v)$ in place of the values $d(s, v)$ and $d(v, t)$, respectively, as required by our heuristic function definitions from Section 7.4.1. Note that we may further augment this algorithm to explore only nodes in $R$, as we require only $R \cap V_\Delta$.

### 7.4.4 Putting It All Together

We may now outline the complete $(1 + \epsilon)$-approximation algorithm workflow, as shown below:

1. **Establish an upper bound**, $\mu$, on the optimal solution cost (discussed below).

2. **Establish $V_\Delta$ for** $\Delta = \mu/(1 + \epsilon)$ (see Section 7.4.3).

3. **Perform Bidirectional A* Search** in $G[R \cap V_{\mu/(1+\epsilon)}]_C$ (see Section 7.4.1).

4. **Return** $min\{\mu, w(P^*)\}$, where $P^*$ is the path found in Step 3 (above).

**Theorem 39** *The above algorithm computes a* $(1+\epsilon)$*-approximate solution in time* $O^*(2^k)$.

**Proof.** The time bound holds from the fact that we are using consistent heuristic functions to search a subgraph of $G_C$, whose full size is bounded by $O^*(2^k)$[96]. Regarding the approximation bound, if $\mu \leq (1+\epsilon) \cdot w(P^C_{s,t})$, then the proof is trivially complete, since we return $min\{\mu, w(P^*)\}$. Therefore, assume instead that $\mu > (1+\epsilon) \cdot w(P^C_{s,t})$. For this case, we shall prove an even stronger claim that $w(P^*) = w(P^C_{s,t})$ (i.e., we find an *optimal* solution). Consider any node $v \in P^C_{s,t}$ on an optimal solution path. We have $d(s,v) + d(v,t) \leq w(P^C_{s,t}) < \mu/(1+\epsilon)$ (this last inequality, by the assumption above), and thus $v \in V_{\mu/(1+\epsilon)}$, by definition. Therefore, $P^C_{s,t} \subseteq V_{\mu/(1+\epsilon)}$ and $(P^C_{s,t} \cap R)$ will be fully explored by the A$^*$ search (Step 3). Since only nodes in $(P^C_{s,t} \cap R)$ need be explored for correctness when using a CH graph [96], and our heuristic functions are admissible, then the search will find an optimal solution path $P^*$ such that $w(P^*) = w(P^C_{s,t})$. ■

**Efficiently Computing $\mu$.** While any strategy for computing the upper bound $\mu$ will suffice for correctness of our algorithm, we present here a greedy, nearest-neighbor heuristic for computing $\mu$ which takes advantage of the pre-processing from Section 7.4.2. Let $\tilde{C} = C$ represent the remaining categories left to visit. Starting at node $x_0 = s$, for $1 \leq i \leq k$, we select a category $C_j \in \tilde{C}$ such that $\pi_j(x_{i-1}) \leq \pi_\ell(x_{i-1})$ for all $C_\ell \in \tilde{C}$ and set $x_i = \rho_j(x_{i-1})$, $\mu_i = \pi_j(x_{i-1})$, and $\tilde{C} = \tilde{C} \setminus C_j$. We thus obtain $\mu = (\sum_{1 \leq i \leq k} \mu_i) + d(x_k, t)$ as a valid upper bound, requiring only a single (fast) point-to-point shortest path computation in the CH graph (see [53]) to evaluate $d(x_k, t)$ (all $\mu_i$ values are simple lookups of pre-processed values).

## 7.5 Experiments

In this section, we present experiments examining the impact of density ($g$) and category count ($k$) on the performance of our $(1+\epsilon)$-approximation algorithm.

### 7.5.1 Test Environment

All experiments were carried out on a 64-bit server machine running Linux CentOS 5.3 with 2 quad-core CPUs clocked at 2.53 GHz with 18 GB RAM (only one core was used per experiment). All programs were written in C++ and compiled using gcc version 4.1.2 with optimization level 3.

### 7.5.2 Test Dataset

All experiments were performed on the road network of North America[7], with $21,133,774$ nodes and $52,523,592$ edges using travel time (in minutes) as the weight function, $w$. This dataset was derived from NAVTEQ data, under their permission.

### 7.5.3 Category Density Experiments

We first examine how our algorithm scales across various category densities. For all experiments discussed here, we fixed the category count at $k = 5$ and selected nodes $s \neq t$ independently at random.

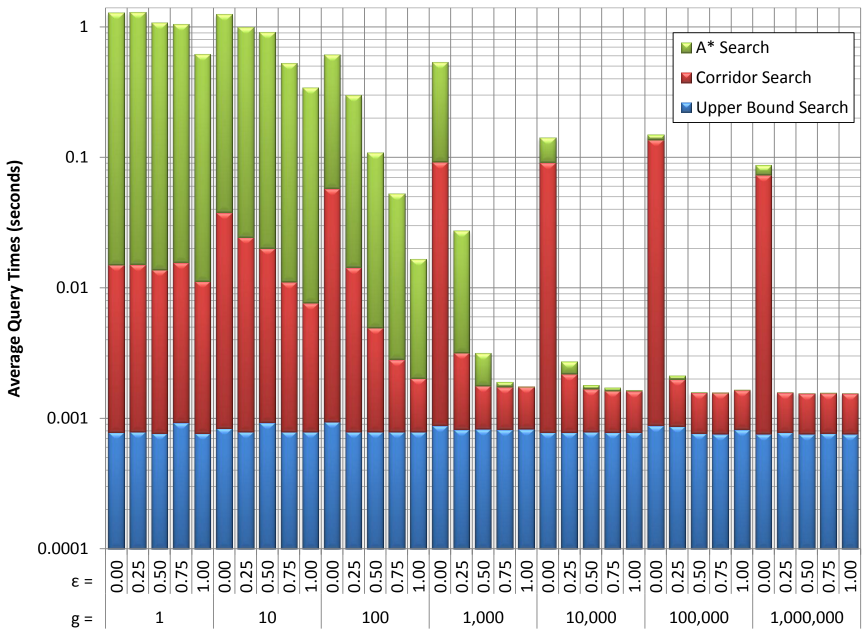
For every $0 \leq i \leq 6$, we constructed 100 random query instances in which each of the $k = 5$ categories were populated with $g = 10^i$ nodes selected uniformly at random. For each density value tested, we additionally tested the impact of the $\epsilon$ value on our solutions. For $0 \leq j \leq 4$, we ran each query with $\epsilon = 0.25j$. Results are presented in Fig. 7.8, showing average query times on the top (note the logarithmic $y$-axis) and

[7]This includes only the US and Canada.

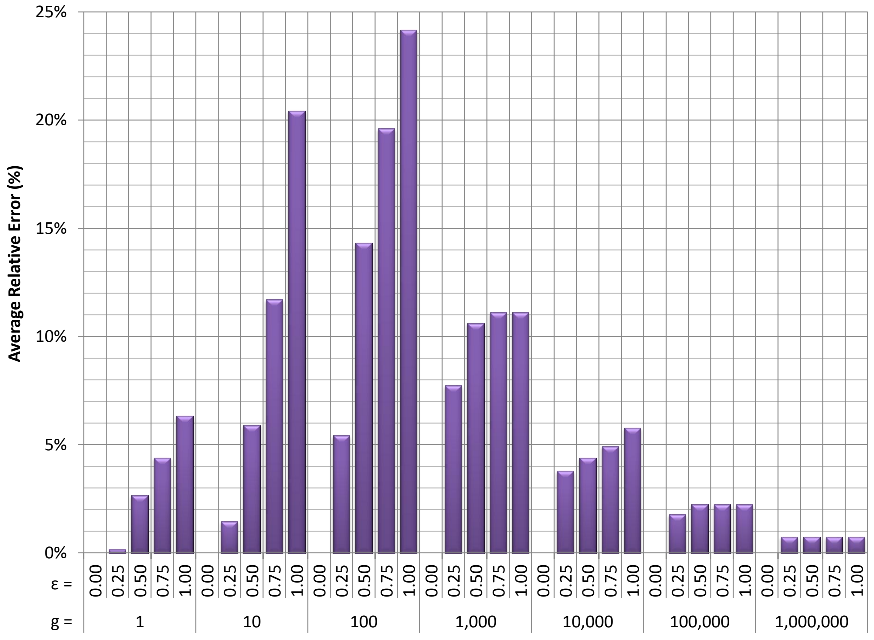
relative errors (w.r.t. optimal) on the bottom.

Overall, we see the total query times decrease from 1 second down to just milliseconds (ms) as both the densities and $\epsilon$ values increase. We note that even modest increases in $\epsilon$ begin to result in generally more-significant speed improvements as the overall density increases (higher densities lead to smaller $\mu$ bounds, which, combined with higher $\epsilon$, significantly helps the pruning process). The query times are further broken down based on the time contributed by each of the 3 main steps of our algorithm from Section 7.4.4. The upper bound construction phase is consistently the fastest ($< 1$ ms), whereas the A$^*$ search overhead quickly diminishes and becomes dominated by the corridor search for increasing density and $\epsilon$ (note that corridor search times for optimal solutions generally increase with density, as we only search nodes in $R$, and $|R|$ also increases with density). For all approximate solutions, our algorithm achieves $< 25\%$ relative error (though typically much better), on average.

To get a sense of how this new algorithm's performance compares to the previous-best runtimes achieved by the non-greedy LESS algorithm from [96], we present the relative speedups over the LESS algorithm for each density value tested here in Table 7.1. Note that, since the LESS algorithm is optimal, we present the comparison against only our own optimal solutions from these experiments (i.e., cases where $\epsilon = 0$). For extremely low densities of, e.g., 1-10 locations per category, we see that our new hybrid algorithm is actually slower, but still achieves satisfactory runtimes of around 1 second (while the LESS algorithm runs in sub-second time for these instances). However, as the density increases, our relative performance begins to show drastic improvements, resulting in speedups of up to 2 orders of magnitude faster than LESS for the highest-density problems tested (for which LESS was previously taking up to 16 seconds to solve).



(a) Average Runtimes



(b) Average Relative Errors

Figure 7.8: Density experiments showing average runtimes (top) and average relative errors (bottom).

Table 7.1: Speedup of our new algorithm with respect to category density, when compared to the previous-best GTSPP algorithm from [96].

| **Category Density** ($g$) | Speedup over LESS[96] |
|---|---|
| $10^0$ | 0.562 |
| $10^1$ | 0.624 |
| $10^2$ | 1.359 |
| $10^3$ | 1.915 |
| $10^4$ | 10.557 |
| $10^5$ | 25.293 |
| $10^6$ | 178.603 |

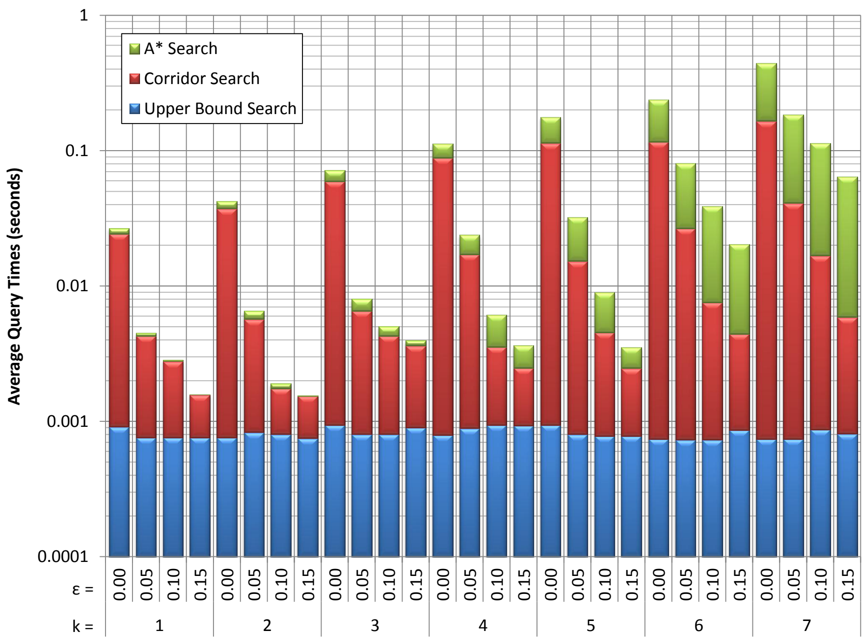
### 7.5.4 Category Count Experiments

We similarly examine how our algorithm scales for various category counts. For all experiments discussed here, we fixed the category density at $g = 10,000$ and selected $s \neq t$ independently at random.

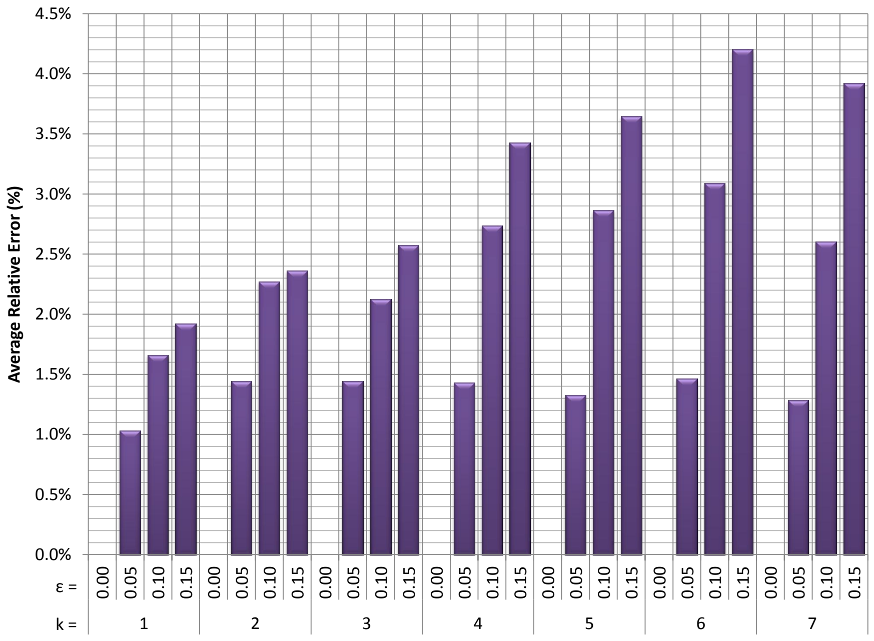
For every $1 \leq i \leq 7$, we constructed 100 random query instances of $k = i$ categories, populated uniformly at random. To further examine the impact of $\epsilon$ on our solutions, for $0 \leq j \leq 3$, we ran each query with $\epsilon = 0.05j$ (a sufficiently-smaller range than before). Results are presented in Fig. 7.9, with average query times on the top (again note the logarithmic $y$-axis) and relative errors on the bottom.

For our query times, we are able to achieve optimal solutions (i.e., $\epsilon = 0$) in times ranging from tens of milliseconds (for $k = 1$) up to only less than half of a second (for $k = 7$). However, even for our most-approximate solutions ($\epsilon = 0.15$), we achieve solutions with $< 5\%$ error in only $< 65$ ms, on average, for all category counts tested.

Of the three separate stages of this algorithm, the A$^*$ search phase shows the most overall relative degradation with respect to increasing category counts. However, this is expected, as this is the only stage of the algorithm which is exponentially-impacted



(a) Average Runtimes



(b) Average Relative Errors

Figure 7.9: Count experiments showing average runtimes (top) and average relative errors (bottom).

Table 7.2: Speedup of our new algorithm with respect to category count, when compared to the previous-best GTSPP algorithm from [96].

| **Category Count ($k$)** | Speedup over LESS[96] |
|---|---|
| 1 | 3.539 |
| 2 | 6.131 |
| 3 | 6.285 |
| 4 | 7.569 |
| 5 | 8.382 |
| 6 | 17.345 |
| 7 | 12.758 |

by the increasing value of $k$ (the other two stages require only polynomial time).

Our speedups relative to the LESS algorithm from [96] are also presented for these experiments in Table 7.2 (again, only optimal solutions with $\epsilon = 0$ were compared). Similar to our previous comparisons with LESS, we see nearly an order of magnitude speed improvement, on average, across all of the various category counts tested.

## 7.6 Conclusion

We have presented both exact and approximate solutions for solving large-scale GTSPP instances in real-world road networks. Our results achieve near real-time solutions for nationwide queries, using only modest pre-processing effort, and result in orders of magnitude speed improvements over the previous-best algorithm from [96] for solving GTSPP queries in road networks. As future work, we conclude with 2 possible extensions to the ideas presented here.

### 7.6.1 Category Subset Selection

The pre-processing of category sets (proposed in Section 7.4.2) currently assumes that *every* location within a given category is valid for consideration within a given GTSPP query which includes that category. However, this formulation may still be considered somewhat inflexible by some practitioners. For example, if we have a single category to represent *all* gas station locations (regardless of brand, etc.), a traveler may still have specific preferences and/or requirements as to which gas stations they would like to consider as valid for a solution; e.g., they might wish to visit only gas stations of a certain brand, only gas stations which also have a car wash or sell diesel, etc. One could easily utilize our existing solution by simply establishing a separate category for each of these possible distinct subsets. However, this may result in an arbitrarily-large number of highly-specific categories during pre-processing, and there is also risk that we simply can not know in advance how users might actually specify their unique preferences for certain category subsets (as their needs are likely to change on a trip-by-trip basis).

Despite all of this, we may still utilize our currently proposed solution framework, in which we pre-process only broadly-defined, high-level categories (such as the set of *all* gas stations), while still allowing users to arbitrarily select from within each category at query time as they see fit. It is easy to prove[8] that the general algorithm proposed in Section 7.4.4 remains correct under such *category subset selection*, as long as (i) the implicitly searched product graph is properly established only for the valid category subsets (which is straightforward to achieve) and (ii) the calculation of the upper bound, $\mu$, properly honors the specific subset selection for each involved category (i.e., remains a valid upper bound). Therefore, one may simply augment Step 1 from

[8]The proof relies on the fact that $R_C$ and our previously-established heuristic functions remain valid/admissible (although weaker) even under subset selection and can therefore be used exactly as before.

our proposed algorithm (Section 7.4.4) to construct a new upper bound honoring the specific subset definitions chosen from each category at query time, allowing for an even more flexible approach in practice.

### 7.6.2 Generalized Orienteering Problems

The Generalized Orienteering Problem (GOP) is a variation on the previous idea of GTSPP in which we seek to maximize the number of unique categories visited along some path from source $s$ to target $t$ within a given cost budget $B$ (i.e., the resulting path cannot cost more than $B$). This problem has similar applications to GTSPP in the domain of personalized location-based services as well. For example, when shopping to buy a new home, a typical consideration is the type of categorical locations (e.g., grocery stores, banks) that can be reached from each home location within some bounded time (e.g., 15 minutes). This helps the buyer to rate each home location based on its overall utility for running common errands. However, classical versions of this problem only consider the travel time to each individual category separately from one another. This could in fact result in much higher travel times than originally anticipated, if, for example, we wish to visit two categorical locations in one trip, but one category is located 15 minutes to the east of the home and one 15 minutes to the west (making for a round trip $\geq 60$ minutes). A more appropriate approach might be to instead consider the types of categories that can be reached within only a *single* round trip from each home, in the given budget. This is exactly what GOP will compute.

In keeping with the approach of computing shortest paths within the established GTSPP product graph, a straightforward enhancement for solving GOP would therefore be to simply choose the largest-cardinality subset $C' \subseteq C$, such that we have $d(\langle s, \emptyset \rangle, \langle t, C' \rangle) \leq B$ (cardinality ties could be broken by preferring those paths with minimum cost). Many of the techniques presented here and in [96] used to solve GT-SPP can thus be easily extended to solve the GOP as well.

Using this new problem formulation, we can effectively view the high-level problem of Generalized Route Planning as having dual versions: a *quota-driven* version (GTSPP) in which we must minimize the cost of a path visiting some fixed quota of categories and a *budget-driven* version (GOP) in which we must maximize the number of categories visited according to some fixed budget on the path cost.

## Part IV

# Conclusion

# Chapter 8

# Conclusion

In this dissertation, we have considered various constraint scenarios for route planning in road networks in order to effectively and efficiently support a more personalized overall navigation experience for the individual traveler. Thus far, we have focused separately on two primary types of personalized constraints, including avoidance constraints enforced by so-called edge restrictions, as discussed in Part II, and detour (or preference) constraints based on categorical points of interest, as discussed in Part III.

Within each of these parts, we progressively engineered various combinations of both hierarchical and goal-directed search techniques. Overall, our hybrid combinations of these two general approaches achieved the best performance by effectively combining the most useful aspects of each approach. Such hybrid algorithms allowed us to achieve optimal (or near-optimal) solutions for each of our various constraint scenarios across continent-wide problem instances, typically in only a matter of milliseconds, on average.

**Future Work.** While each particular topic has been addressed separately, both for simplicity as well as self-containment, we should note that there is also a natural synergy between our results for these two separate constraint types, as they may be easily integrated to work together as a single, all-in-one route-planning solution. This should be readily apparent, as the engineered results of Part III work so well in large part due to some general structural properties of the CH index, which are also maintained by our own extended CH preprocessing techniques from Part II, as well. Combining our results from these two respective sections into a single, integrated approach would therefore ultimately allow us to solve even more practical combinations of such constraint scenarios for personalized navigation (e.g., find the best detour through a given category set, which also avoids undesirable roads).

Aside from the prospect of more tightly integrating the various parts of this dissertation, there are many additional interesting directions to consider for future work. The most obvious and straightforward of these would be to further extend the results of this work to also incorporate more of the various other practical concepts currently being explored for route planning functionality, including (but not limited to): incorporating turn restrictions, time-dependent route planning, multi-criteria route planning, finding multiple alternative routes, multi-modal route planning (with transit schedules), etc.

However, apart from these extensions, perhaps the most promising (but least-straightforward) of these new directions would be to consider further research into the discovery of new and efficient, fixed-parameter tractable algorithms for other complex (i.e., NP-hard) route planning problems within the context of road networks (such as was discovered for our own GTSP problem instances). For example, are there other parameterizations (e.g., highway dimension [8], etc.) for NP-hard problems within the context of road networks that can similarly make such a surprisingly-practical impact? Applying such concepts of parameterized complexity within the broad domain of route planning could potentially result in some of the most significant achievements thus far.

# Bibliography

[1] *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*, volume 22. 2004.

[2] *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*. SIAM, 2005.

[3] *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*. SIAM, 2007.

[4] *Seventh Cologne Twente Workshop on Graphs and Combinatorial Optimization, Gargano, Italy, 13-15 May, 2008*. University of Milan, 2008.

[5] *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 2009.

[6] *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*. IEEE, 2011.

[7] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. In Pardalos and Rebennack [89], pages 230–241.

[8] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In Charikar [31], pages 782–793.

[9] Carme Àlvarez and Maria J. Serna, editors. *Experimental Algorithms, 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006, Proceedings*, volume 4007 of *Lecture Notes in Computer Science*. Springer, 2006.

[10] Lars Arge, Giuseppe F. Italiano, and Robert Sedgewick, editors. *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics, New Orleans, LA, USA, January 10, 2004*. SIAM, 2004.

[11] Esther M. Arkin and Refael Hassin. Approximation Algorithms for the Geometric Covering Salesman Problem. *Discrete Applied Mathematics*, 55(3):197–218, 1994.

[12] Stefan Arnborg and Lars Ivansson, editors. *Algorithm Theory - SWAT '98, 6th Scandinavian Workshop on Algorithm Theory, Stockholm, Sweden, July, 8-10, 1998, Proceedings*, volume 1432 of *Lecture Notes in Computer Science*. Springer, 1998.

[13] Yossi Azar and Thomas Erlebach, editors. *Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings*, volume 4168 of *Lecture Notes in Computer Science*. Springer, 2006.

[14] Christopher L. Barrett, Keith Bisset, Martin Holzer, Goran Konjevod, Madhav V. Marathe, and Dorothea Wagner. Label-Constrained Shortest-Path Algorithms: An Experimental Evaluation Using Transportation Networks. Technical report, NDSSL, Virginia Tech., 2007.

[15] Christopher L. Barrett, Keith Bisset, Martin Holzer, Goran Konjevod, Madhav V. Marathe, and Dorothea Wagner. Engineering Label-Constrained Shortest-Path Algorithms. In Fleischer and Xu [47], pages 27–37.

[16] Christopher L. Barrett, Keith Bisset, Riko Jacob, Goran Konjevod, and Madhav V. Marathe. Classical and Contemporary Shortest Path Problems in Road Networks: Implementation and Experimental Analysis of the TRANSIMS Router. In Möhring and Raman [85], pages 126–138.

[17] Christopher L. Barrett, Riko Jacob, and Madhav V. Marathe. Formal Language Constrained Path Problems. In Arnborg and Ivansson [12], pages 234–245.

[18] Christopher L. Barrett, Riko Jacob, and Madhav V. Marathe. Formal-Language-Constrained Path Problems. *SIAM J. Comput.*, 30(3):809–837, 2000.

[19] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In Transit to Constant Time Shortest-Path Queries in Road Networks. In *ALENEX* [3].

[20] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, April 2007.

[21] Giuseppe Di Battista and Uri Zwick, editors. *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*, volume 2832 of *Lecture Notes in Computer Science*. Springer, 2003.

[22] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. In Munro and Wagner [86], pages 13–26.

[23] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. In McGeoch [83], pages 303–318.

[24] Arash Behzad and Mohammad Modarres. A New Efficient Transformation of Generalized Traveling Salesman Problem into Traveling Salesman Problem. In *Proceedings of the 15th International Conference of Systems Engineering (ICSE)*, pages 6–8, 2002.

[25] David Ben-arieh, Gregory Gutin, M. Penn, Anders Yeo, and Alexey Zverovitch. Transformations of Generalized ATSP into ATSP. *Operations Research Letters*, 31(3):357–365, 2003.

[26] Guy E. Blelloch and Dan Halperin, editors. *Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments, ALENEX 2010, Austin, Texas, USA, January 16, 2010*. SIAM, 2010.

[27] Daniel Borrajo, Ariel Felner, Richard E. Korf, Maxim Likhachev, Carlos Linares López, Wheeler Ruml, and Nathan R. Sturtevant, editors. *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS 2012, Niagara Falls, Ontario, Canada, July 19-21, 2012*. AAAI Press, 2012.

[28] Gerth Stølting Brodal and Stefano Leonardi, editors. *Algorithms - ESA 2005, 13th Annual European Symposium, Palma de Mallorca, Spain, October 3-6, 2005, Proceedings*, volume 3669 of *Lecture Notes in Computer Science*. Springer, 2005.

[29] Valentina Cacchiani, Albert Einstein Fernandes Muritiba, Marcos Negreiros, and Paolo Toth. A Multi-Start Heuristic Algorithm for the Generalized Traveling Salesman Problem. In *CTW* [4], pages 136–138.

[30] Valentina Cacchiani, Albert Einstein Fernandes Muritiba, Marcos Negreiros, and Paolo Toth. A Multistart Heuristic for the Equality Generalized Traveling Salesman Problem. *Networks*, 57(3):231–239, 2011.

[31] Moses Charikar, editor. *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*. SIAM, 2010.

[32] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and Distance Queries Via 2-Hop Labels. In Eppstein [45], pages 937–946.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[34] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-Accelerated Shortest Path Trees. In *IPDPS* [6], pages 921–931.

[35] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In Lerner et al. [78], pages 117–139.

[36] Daniel Delling and Dorothea Wagner. Landmark-Based Routing in Dynamic Graphs. In Demetrescu [37], pages 52–65.

[37] Camil Demetrescu, editor. *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*, volume 4525 of *Lecture Notes in Computer Science*. Springer, 2007.

[38] Camil Demetrescu, Robert Sedgewick, and Roberto Tamassia, editors. *Proceedings of the Seventh Workshop on Algorithm Engineering and Experiments and the Second Workshop on Analytic Algorithmics and Combinatorics, ALENEX /ANALCO 2005, Vancouver, BC, Canada, 22 January 2005*. SIAM, 2005.

[39] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[40] Vladimir Dimitrijevic and Zoran Saric. An Efficient Transformation of the Generalized Traveling Salesman Problem into the Traveling Salesman Problem on Digraphs. *Inf. Sci.*, 102(1-4):105–110, 1997.

[41] Rod G. Downey and M. R. Fellows. *Parameterized Complexity (Monographs in Computer Science).* Springer, November 1998.

[42] Andreas W. M. Dress, Yinfeng Xu, and Binhai Zhu, editors. *Combinatorial Optimization and Applications, First International Conference, COCOA 2007, Xi'an, China, August 14-16, 2007, Proceedings*, volume 4616 of *Lecture Notes in Computer Science*. Springer, 2007.

[43] Moshe Dror and Mohamed Haouari. Generalized Steiner Problems and Other Variants. *J. Comb. Optim.*, 4(4):415–436, 2000.

[44] Adrian Dumitrescu and Joseph S. B. Mitchell. Approximation Algorithms for TSP with Neighborhoods in the Plane. In Kosaraju [70], pages 38–46.

[45] David Eppstein, editor. *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA.* ACM/SIAM, 2002.

[46] Matteo Fischetti, Juan Jose Salazar Gonzalez, and Paolo Toth. A Branch-And-Cut Algorithm for the Symmetric Generalized Traveling Salesman Problem. *Operations Research*, 45(3):378–394, 1997.

[47] Rudolf Fleischer and Jinhui Xu, editors. *Algorithmic Aspects in Information and Management, 4th International Conference, AAIM 2008, Shanghai, China, June 23-25, 2008. Proceedings*, volume 5034 of *Lecture Notes in Computer Science*. Springer, 2008.

[48] J. Flum and M. Grohe. *Parameterized Complexity Theory (Texts in Theoretical Computer Science. An EATCS Series).* Springer, March 2006.

[49] Michael L. Fredman and Robert Endre Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. ACM*, 34(3):596–615, 1987.

[50] Robert Geisberger. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. Master's thesis, Institut fur Theoretische Informatik Universitat Karlsruhe, 2008.

[51] Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. Route Planning with Flexible Objective Functions. In Blelloch and Halperin [26], pages 124–137.

[52] Robert Geisberger, Michael N. Rice, Peter Sanders, and Vassilis J. Tsotras. Route Planning with Flexible Edge Restrictions. *ACM Journal of Experimental Algorithmics*, 17(1), 2012.

[53] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In McGeoch [83], pages 319–333.

[54] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *SODA* [2], pages 156–165.

[55] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In Raman and Stallmann [93], pages 129–143.

[56] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Better Landmarks Within Reach. In Demetrescu [37], pages 38–51.

[57] Andrew V. Goldberg and Renato F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In Demetrescu et al. [38], pages 26–40.

[58] Joachim Gudmundsson and Christos Levcopoulos. A Fast Approximation Algorithm for TSP with Neighborhoods and Red-Blue Separation. In *COCOON*, pages 473–482, 1999.

[59] Gregory Gutin and Daniel Karapetyan. Generalized Traveling Salesman Problem Reduction Algorithms. *Algorithmic Operations Research*, 4(2):144–154, 2009.

[60] Ronald J. Gutman. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In Arge et al. [10], pages 100–111.

[61] Dan Halperin and Kurt Mehlhorn, editors. *Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings*, volume 5193 of *Lecture Notes in Computer Science*. Springer, 2008.

[62] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[63] A. L. Henry-Labordere. The Record Balancing Problem: A Dynamic Programming Solution of a Generalized Traveling Salesman Problem. *RIRO*, B-2:43–49, 1969.

[64] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. [5], pages 41–72.

[65] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering Multilevel Overlay Graphs for Shortest-Path Queries. *ACM Journal of Experimental Algorithmics*, 13, 2009.

[66] T. Ikeda, Min-Yao Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh. A Fast Algorithm for Finding Better Routes by AI Search Techniques. In VNIS-94 [111], pages 291–296.

[67] Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou, editors. *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. IEEE Computer Society, 2013.

[68] Ralf Klasing, editor. *Experimental Algorithms - 11th International Symposium, SEA 2012, Bordeaux, France, June 7-9, 2012. Proceedings*, volume 7276 of *Lecture Notes in Computer Science*. Springer, 2012.

[69] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing Many-to-Many Shortest Paths Using Highway Hierarchies. In *ALENEX* [3].

[70] S. Rao Kosaraju, editor. *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA*. ACM/SIAM, 2001.

[71] James B. H. Kwa. BS*: An Admissible Bidirectional Staged Heuristic Search Algorithm. *Artif. Intell.*, 38(1):95–109, 1989.

[72] Gilbert Laporte, Adravan Asef-Vaziri, and Chelliah Sriskandarajah. Some Applications of the Generalized Travelling Salesman Problem. *The Journal of the Operational Research Society*, 47(12):1461–1467, 1996.

[73] Gilbert Laporte, Helene Mercure, and Yves Nobert. Generalized Travelling Salesman Problem Through n Sets of Nodes: The Asymmetrical Case. *Discrete Applied Mathematics*, 18(2):185–197, 1987.

[74] Gilbert Laporte and Yves Nobert. Generalized Travelling Salesman Problem Through n Sets of Nodes: An Integer Programming Approach. *INFOR*, 21(1):61–75, 1983.

[75] Gilbert Laporte and U. Palekar. Some Applications of the Clustered Travelling Salesman Problem. *The Journal of the Operational Research Society*, 53(9):972–976, 2002.

[76] Gilbert Laporte and F. Semet. Computational Evaluation of a Transformation Procedure for the Symmetric Generalized Traveling Salesman Problem. *INFOR*, 37(2):114–120, 1999.

[77] Ulrich Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung* [1], pages 219–230.

[78] Jürgen Lerner, Dorothea Wagner, and Katharina Anna Zweig, editors. *Algorithmics of Large and Complex Networks - Design, Analysis, and Simulation [DFG priority program 1126]*, volume 5515 of *Lecture Notes in Computer Science*. Springer, 2009.

[79] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. On Trip Planning Queries in Spatial Databases. In Medeiros et al. [84], pages 273–290.

[80] Yao-Nan Lien, Y. W. Eva Ma, and Benjamin W. Wah. Transformation of the Generalized Traveling-Salesman Problem Into the Standard Traveling-Salesman Problem. *Inf. Sci.*, 74(1-2):177–189, 1993.

[81] Cristian S. Mata and Joseph S. B. Mitchell. Approximation Algorithms for Geometric Tour and Network Design Problems (Extended Abstract). In *Symposium on Computational Geometry*, pages 360–369, 1995.

[82] Jens Maue, Peter Sanders, and Domagoj Matijevic. Goal Directed Shortest Path Queries Using Precomputed Cluster Distances. In Àlvarez and Serna [9], pages 316–327.

[83] Catherine C. McGeoch, editor. *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, volume 5038 of *Lecture Notes in Computer Science*. Springer, 2008.

[84] Claudia Bauzer Medeiros, Max J. Egenhofer, and Elisa Bertino, editors. *Advances in Spatial and Temporal Databases, 9th International Symposium, SSTD 2005, Angra dos Reis, Brazil, August 22-24, 2005, Proceedings*, volume 3633 of *Lecture Notes in Computer Science*. Springer, 2005.

[85] Rolf H. Möhring and Rajeev Raman, editors. *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*, volume 2461 of *Lecture Notes in Computer Science*. Springer, 2002.

[86] J. Ian Munro and Dorothea Wagner, editors. *Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments, ALENEX 2008, San Francisco, California, USA, January 19, 2008*. SIAM, 2008.

[87] Charles E. Noon and James C. Bean. A Lagrangian Based Approach for the Asymmetric Generalized Traveling Salesman Problem. *Operations Research*, 39(4):623–632, 1991.

[88] Charles E. Noon and James C. Bean. An Efficient Transformation of the Generalized Traveling Salesman Problem. *INFOR*, 31(1):39–44, 1993.

[89] Panos M. Pardalos and Steffen Rebennack, editors. *Experimental Algorithms - 10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proceedings*, volume 6630 of *Lecture Notes in Computer Science*. Springer, 2011.

[90] Wim Pijls and Henk Post. A New Bidirectional Search Algorithm with Shortened Postprocessing. *European Journal of Operational Research*, 198(2):363–369, 2009.

[91] Ira Pohl. *Bidirectional Heuristic Search in Path Problems*. PhD thesis, Stanford University, 1969.

[92] Petrica C. Pop, Corina Pop Sitar, Ioana Zelina, and Ioana Tascu. Exact Algorithms for Generalized Combinatorial Optimization Problems. In Dress et al. [42], pages 154–162.

[93] Rajeev Raman and Matthias F. Stallmann, editors. *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments, ALENEX 2006, Miami, Florida, USA, January 21, 2006*. SIAM, 2006.

[94] Michael N. Rice and Vassilis J. Tsotras. Graph Indexing of Road Networks for Shortest Path Queries with Label Restrictions. *PVLDB*, 4(2):69–80, 2010.

[95] Michael N. Rice and Vassilis J. Tsotras. Bidirectional A* Search with Additive Approximation Bounds. In Borrajo et al. [27].

[96] Michael N. Rice and Vassilis J. Tsotras. Exact Graph Search Algorithms for Generalized Traveling Salesman Path Problems. In Klasing [68], pages 344–355.

[97] Michael N. Rice and Vassilis J. Tsotras. Engineering Generalized Shortest Path Queries. In Jensen et al. [67], pages 949–960.

[98] Michael N. Rice and Vassilis J. Tsotras. Parameterized Algorithms for Generalized Traveling Salesman Problems in Road Networks. In *ACM SIGSPATIAL*, 2013.

[99] Shmuel Safra and Oded Schwartz. On the Complexity of Approximating TSP with Neighborhoods and Related Problems. In Battista and Zwick [21], pages 446–458.

[100] J. P. Saksena. Mathematical Model of Scheduling Clients Through Welfare Agencies. *CORS Journal*, 8:185–200, 1970.

[101] Peter Sanders and Dominik Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In Brodal and Leonardi [28], pages 568–579.

[102] Peter Sanders and Dominik Schultes. Engineering Highway Hierarchies. In Azar and Erlebach [13], pages 804–816.

[103] Peter Sanders and Dominik Schultes. Engineering Fast Route Planning Algorithms. In Demetrescu [37], pages 23–36.

[104] Peter Sanders, Dominik Schultes, and Christian Vetter. Mobile Route Planning. In Halperin and Mehlhorn [61], pages 732–743.

[105] Dominik Schultes and Peter Sanders. Dynamic Highway-Node Routing. In Demetrescu [37], pages 66–79.

[106] Mehdi Sharifzadeh, Mohammad R. Kolahdouzan, and Cyrus Shahabi. The Optimal Sequenced Route Query. *The VLDB Journal*, 17(4):765–787, 2008.

[107] Mehdi Sharifzadeh and Cyrus Shahabi. Processing Optimal Sequenced Route Queries Using Voronoi Diagrams. *GeoInformatica*, 12(4):411–433, 2008.

[108] P. Slavik. The Errand Scheduling Problem. Technical Report 97-2, Department of Computer Science, SUNY, Buffalo NY, March 1997.

[109] Lawrence V. Snyder and Mark S. Daskin. A Random-Key Genetic Algorithm for the Generalized Traveling Salesman Problem. *European Journal of Operational Research*, 174(1):38–53, 2006.

[110] S. S. Srivastava, S. Kumar, R. C. Garg, and P. Sen. Generalized Travelling Salesman Problem Through n Sets of Nodes. *CORS Journal*, 7:97–101, 1969.

[111] *Proceedings of the Vehicle Navigation and Information Systems Conference (VNSI'94)*. ACM Press, 1994.

[112] Dorothea Wagner and Thomas Willhalm. Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs. In Battista and Zwick [21], pages 776–787.