Fast Computation of Continental-Sized Isochrones

Paolo Bolzoni[1], Sven Helmer[1], Oded Lachish[2]

[1] Faculty of Computer Science, Free University of Bozen-Bolzano, 39100 Bolzano, Italy
Email: {firstname.lastname}@unibz.it

[2] Dept. of Computer Science and Information Systems, Birkbeck, University of London, London WC1E 7HX, United Kingdom
Email: oded@dcs.bbk.ac.uk

Abstract

We propose an approach to speed up the computation of isochrones, which are maps showing the reachability of locations given a starting point and a time constraint. The core idea of our technique is to materialize large parts of an isochrone, demonstrating how this can be achieved for multi-modal transport networks in a scalable way. We illustrate the effectiveness of our method with the help of an experimental evaluation.

1. Introduction

Isochrone maps, which given a starting location show the reachability of places within a certain time span, have been around for more than a hundred years. Before the introduction of computer systems and the digitization of map data, creating these artifacts was a very time-consuming task. The easy availability of geographical information systems (GIS), such as PostGIS, and map data, such as OpenStreetMap, has sparked a renewed interest in isochrones. The applications of isochrone maps are manifold. For example, in urban and regional planning they can be used to determine the location of public services, such as hospitals, schools, police and fire stations, making sure that catchment areas cover an adequate part of the population. When planning new transport links, isochrones can help in identifying zones that are not well-connected. Establishing evacuation routes for emergency situations can be facilitated as well. In real estate applications potential buyers and tenants can check which accommodations are within easy reach of workplaces and schools. Finally, planning trips on the fly to quickly determine which places are reachable in an acceptable time frame is also made easier.

Currently, there are no isochrone algorithms that scale to very large and detailed maps. Our goal is to compute isochrones very efficiently, in the ideal case in real-time. While there are efficient algorithms for route planning, most of them are uni-modal, i.e., only one mode of transportation, e.g. by car, is used. Usually, people change their mode of transportation a few times when moving from one location to another, though. There are only a few sophisticated algorithms for multi-modal route planning, e.g. Bast *et al.* (2015), but they only compute shortest paths from point to point.

While a lot of the early work on algorithms for computing isochrones, and also some of the more recent work, assumes that the underlying network is fairly homogeneous, Gamper *et al.* (2012) specifically investigate isochrones for multi-modal transportation networks. These approaches are implemented on top of database systems utilizing geographical information system features. On the one hand, this makes it easier to implement the computation of isochrones, as some general-purpose database system functionality can be re-used. On the other hand, not even geographical information systems support or are optimized for the direct computation of isochrones, so there is still a lot of code outside of the database system that needs to be written.

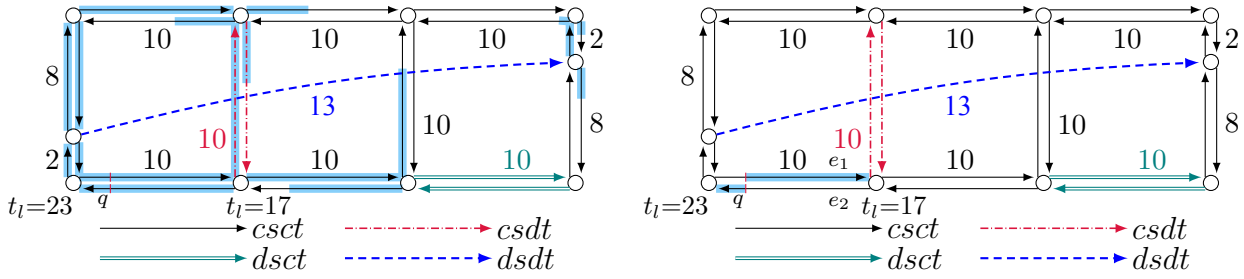
Our technique offers a highly scalable approach for computing isochrones on multi-modal transportation networks efficiently. In summary, we make the following contributions: we develop a data structure that precomputes and materializes a large part of an isochrone map; as a result, our algorithm can assemble large parts of the answer by sequentially scanning the data structure and in the case of a

single-mode scenario only a single sequential scan is required; we evaluate our technique experimentally using real-world data extracted from OpenStreetMap for Europe, showing that we can compute large isochrones quickly.

2. Problem Definition

Let us briefly define the multi-modal transportation network we use (it is similar to the one used by Booth *et al.* (2009)). Vertices represent noteworthy points in the street network like crossings or bus stops, arcs represent connections between these points. Given an arc e we define $d(e)$ as its destination, and $s(e)$ as its source. Moreover, we need a set M of transportation modes that are associated with arcs. Each mode $m \in M$ has a label (e.g. pedestrian, car, bus, train) and a description whether the mode is discrete or continuous in space (ds or cs) and discrete or continuous in time (dt or ct). An example for a continuous space and time ($csct$) mode is a pedestrian network. Any point can be reached at any time. Public transport systems, such as trains and buses, are discrete in space and time ($dsdt$), as they only run at specific times and can only be boarded or left at certain locations, e.g. stations and bus stops. Examples for a mode that is discrete in space and continuous in time ($dsct$) are escalators and tunnels: they operate continuously, but a person cannot get off before reaching the end. Finally, a mode continuous in space and discrete in time ($csdt$) are roads closed at specific times. Due to the multiple different modes, we actually have a multiset of edges, i.e., parallel arcs are possible (e.g. by walking or taking the bus between two stops).

In order to compute an isochrone *Iso*, we need a starting location q, a time threshold t_{max}, and a starting time t^s (needed for the departure times of ds arcs). The isochrone for a query includes all parts of the network that can be reached from q within t_{max}. The answer to q very likely consists of partial arcs around the boundary of the isochrone (see Figure 1(a) for a query q, located in the lower left corner, with $t_{max} = 25$; please also note the different transportation modes, in fact the bus on the $dsdt$ arc is intended to depart at $t^s + 9$ and it is impossible to cross the discrete arcs in the bottom right). Furthermore, the location q does not have to be a vertex: it can lie on continuous space (cs) arcs and then has to be converted to queries starting from the endpoints of the arcs. In Figure 1(b) q lies on the arcs e_1 and e_2, both of which have a weight of 10. Moreover, we know that q has a distance of 8 to $d(e_1)$ and a distance of 2 to $d(e_2)$, which means that at $d(e_1)$ we have 17 minutes left and at $d(e_2)$ 23 minutes.



(a) Example **(b) Starting queries**

Figure 1: Isochrones

3. Our Approach

The basic idea of our approach is to precompute and materialize a large part of isochrones, namely those along the pedestrian network (or any other large $csct$ network). For every node $x \in V$ we create

a list L_x of triplets $((e, \delta(x, s(e)), w_e)$, where $e \in A$ (A being a multiset of directed edges, or arcs), w_e is the weight of the arc e, and $\delta(u, v)$ computes the minimal distance between $u \in V$ and $v \in V$. Basically, we compute the distance δ from node x to the starting node $s(e)$ of every arc e. This also includes starting nodes of arcs that are not part of the $csct$ network. However, to compute $\delta(x, s(e))$ we only use $csct$ arcs and the triplets in a list are sorted in increasing distance of $s(e)$ from x. This data structure allows us to do a very fast computation of the $csct$ components of an isochrone: we just need to sequentially scan L_x. Every time we encounter a non-$csct$ arc, though, we have to trigger a new subquery. In our case we use the algorithm by Johnson (1977)[1] applied to the pedestrian network computing the all-pairs shortest paths to determine the values for δ.

However, applying our technique in a straightforward way does not scale: in the worst case we need storage space quadratic in the number of nodes. Therefore, we partition the graph and apply our technique to every partition (connecting the individual partitions in the process). We employ METIS by Karypis and Kumar (1999), a fast and readily available state-of-the-art algorithm for partitioning graphs; we use the kMETIS variant. We configured it to create partitions minimizing the number of cut arcs under the constraints of producing contiguous partitions and balancing them (in our case the allowed difference in size is at most 3%).

Querying works in the following way. We store the queries to be processed in a priority queue Q, which is initialized with the starting point q (or two starting points, in case q lies on an arc), the starting time t^s, and t_{max}. Q sorts the queries by partition identifier (in order of appearance), breaking ties via the duration of a query in descending order. In other words, if the partition of the starting node v_i of a query q_i that is added to Q is already in Q, then q_i is grouped with these queries (ordering the queries in this group by the remaining time t_i^l). Otherwise, q_i is appended to the end of the queue. Grouping queries by partitions means we can keep the processing localized, not jumping back and forth between different parts of the graph. Ordering queries by duration allows us to do effective pruning. If we have already run a query from a starting node v_i with more time remaining, then there is no reason to run one from the same point with less time remaining. The former query will always cover a greater area.

While the query in front of the queue is still in the same partition as the previously executed one, we process it, remove it from the queue, add new (partial) arcs to the isochrone, and enqueue any new queries resulting from the processing of the current one (no new queries will be scheduled if the distance of the nodes in the materialized list becomes too great). If the next query to be processed accesses a different partition, we switch to a new partition. Once Q runs empty, the algorithm terminates.

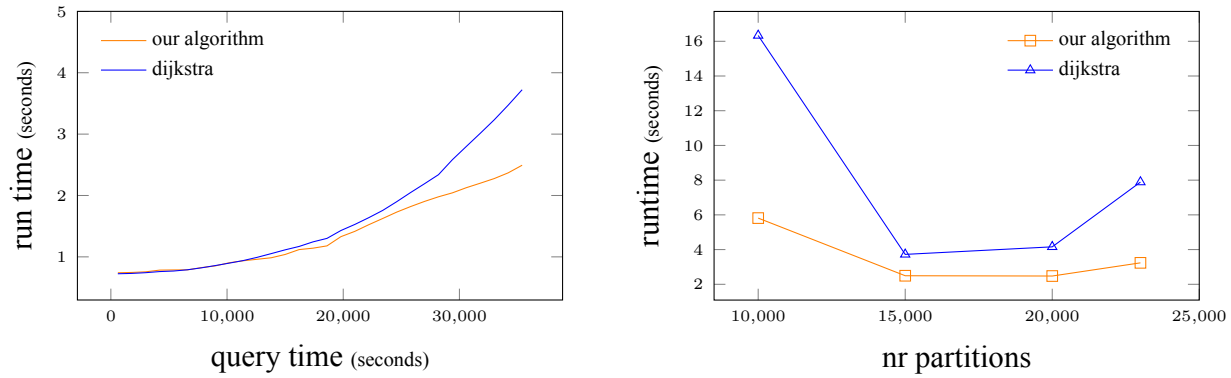
Processing a triplet also depends on what kind of arc we are facing: $csct$, $csdt$, $dsct$, or $dsdt$. The simplest case is $csct$, as this means that we are staying within the materialized network. We merely have to figure out which part of the arc we cover. We calculate the fraction l_e of the arc we can reach, given the time that is left after arriving at $s(e)$ and the (partial) arc l_e is added to the isochrone. If the arc e crosses into another partition, we have to schedule a new query, which is added to Q. The other types of arcs are not much more difficult to handle. $csdt$ arcs, which are closed at certain times, may introduce waiting times, but apart from this are treated in the same way as $csct$ arcs. The discrete space arcs ($dsct$ and $dsdt$) can only be traversed fully or not at all.

4. Experimental Evaluation

We evaluated our algorithm on a PC with an Intel i7-4800MQ CPU running at 2.7 GHz, 24 GBytes RAM, and a 120GB solid state disk. The dataset, taken from OpenStreetMap, is a map of continental Italy comprising 3.7 million vertices and 9.7 million edges.

Figure 2(a) shows a comparison of our technique to running Dijkstra's algorithm to find an isochrone

[1] For sparse networks, Johnson's algorithm is more efficient than Floyd-Warshall.



(a) Run time comparison **(b) Effect of partitioning**

Figure 2: Experiments

using 20,000 partitions. We make a couple of observations. Gamper *et al.* (2012) implemented Dijkstra's algorithm by loading the whole network into main memory, resulting in a suboptimal performance. For an isochrone of ten hours Dijkstra took 100 seconds and their algorithm MINEX even longer than that. We adapted Dijkstra to profit from the graph partitioning as well, leading to a much more competitive approach. Nevertheless, our algorithm still outperforms it for large isochrones, making it much more scalable. We also investigated the effect of the number of partitions on the performance (see Figure 2(b) for ten hour queries). As it turns out, there is a sweet spot: a small number of large partitions means we load too much unused data during query processing, a large number of small partitions means we lose time due to too many I/O operations. We suppose that the optimal number of partitions is related to the number of (urban) agglomerations, but at the moment we determine this number experimentally.

5. Conclusion and Future Work

We show how to implement the computation of isochrones in a scalable way, i.e., we can determine large isochrones on OpenStreetMap graphs very quickly (e.g. a ten hour isochrone for a map of Italy takes on average less than three seconds). We manage to do so by precomputing and materializing parts of the solution. For future work, we see optimization potential in parallelizing the algorithm and developing more intelligent partition loading strategies. Additionally, we want to work on efficiently rendering the full contours of an isochrone.

References

Bast, H., Delling, D., Goldberg, A. V., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D., and Werneck, R. F. (2015). Route planning in transportation networks. *Computing Research Repository (CoRR)*, **abs/1504.05140**.

Booth, J., Sistla, P., Wolfson, O., and Cruz, I. F. (2009). A data model for trip planning in multimodal transportation systems. In *Proc. of the 12th Int. Conf. on Extending Database Technology (EDBT'09)*, pages 994–1005, Saint Petersburg, Russia.

Gamper, J., Böhlen, M. H., and Innerebner, M. (2012). Scalable computation of isochrones with network expiration. In *Proc. of 24th Int. Conf. Scientific and Statistical Database Management (SSDBM'12)*, pages 526–543, Chania, Crete, Greece.

Johnson, D. B. (1977). Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, **24**(1), 1–13.

Karypis, G. and Kumar, V. (1999). A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, **20**(1), 359–392.