# Engineering an Approximation Scheme for Traveling Salesman in Planar Graphs*

**Amariah Becker[1], Eli Fox-Epstein[2], Philip N. Klein[3], and David Meierfrankenfeld[4]**

1 **Department of Computer Science, Brown University, Providence, RI, USA**
becker@cs.brown.edu
2 **Department of Computer Science, Brown University, Providence, RI, USA**
ef@cs.brown.edu
3 **Department of Computer Science, Brown University, Providence, RI, USA**
klein@cs.brown.edu
4 **Department of Computer Science,Brown University, Providence, RI, USA**
nfelddav@cs.brown.edu

## Abstract

We present an implementation of a linear-time approximation scheme for the traveling salesman problem on planar graphs with edge weights. We observe that the theoretical algorithm involves constants that are too large for practical use. Our implementation, which is not subject to the theoretical algorithm's guarantee, can quickly find good tours in very large planar graphs.

**1998 ACM Subject Classification** F.2.2 Nonnumerical Algorithms and Problems

**Keywords and phrases** Traveling Salesman, Approximation Schemes, Planar Graph Algorithms, Algorithm Engineering

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2017.8

## 1 Introduction

> "Many (if not most) polynomial-time approximation schemes (PTASs) ... for the traveling salesman problem (TSP) and related problems suffer from gigantic constant factors." –Müller-Hannemann and Schirra [17]

**Overcoming gigantic constant factors.** Müller-Hannemann and Schirra's introduction on algorithm engineering [17] gives this example of how the daunting complexity of algorithms and outrageous "constants," byproducts of worst-case asymptotic analysis preferred by the theory community, can lead to seemingly unimplementable algorithms. Tazari and Müller-Hannemann [21] give the first implementation of an "inherently impractical" algorithm suffering from abysmal hidden constants: a PTAS for the Steiner Tree problem on planar graphs. Their surprising outcome was a very practical implementation, one that could get reasonably good solutions on larger instances than could be addressed with previous implementations. Our work on implementation of a seemingly theoretical approximation scheme is inspired by that outcome.

The framework underlying the Steiner-tree PTAS [4] was presented in a paper [15] that illustrated the framework by presenting a linear-time approximation scheme for the Traveling

* Research funded by NSF grant CCF-14-09520. Klein's research received additional support from the Radcliffe Institute of Advanced Study, Harvard University.


Salesman Problem (TSP) on the metric defined by a planar graph with edge-weights. The application of the framework to this problem is perhaps the simplest illustration of the framework, though the dynamic program at its heart is harder for TSP than for Steiner tree.

**Our implementation as a first step.** We describe an implementation of this TSP approximation scheme, adapted and engineered so that it is no longer guaranteed to find near-optimal tours but it runs quickly. Our implementation typically runs in less than a millisecond per vertex and provides significantly better tours than similarly fast heuristics on very large graphs. Due to the magnitude of the constants involved, it was not our objective to outperform existing implementations on small graphs. Instead, this proof-of-concept implementation targets very large graphs, tests the strategies for coping with the challenges of highly theoretical algorithms outlined by Tazari and Müller-Hannemann [21], and promotes implementation within the theory community, even for the most theoretical algorithms.

Moreover, this implementation is a first step toward implementing more complicated, related approximation schemes for other problems, including *Subset TSP* (a.k.a. *Steiner TSP*), in which the tour need only visit a given subset of the vertices. It is also a first step towards implementing a method that can cope with the nonplanarities and asymmetry of real road networks. Such progress is potentially valuable because there is potential to address other problems arising in road networks, such as ride-sharing, package-delivery routing, and public transportation layout. It remains to be seen whether these techniques can be adapted to address practical applications; in this work we add to the evidence provided by Tazari and Müller-Hannemann showing that the large constant factors in the theoretical algorithms do not represent a fundamental obstacle in this effort. It is in this sense that our implementation is a proof of concept.

**Alternative implementations.** The literature on solving TSP exactly, approximately, and heuristically in a variety of settings is too vast to thoroughly review. Much of the literature and implementations address primarily the *Euclidean case*, in which one seeks a tour through a Euclidean space that visits all given points and has minimum length. In particular, implementations (e.g. [3, 18]) that aim for both linear time and high-quality solutions are restricted to the Euclidean case. A more general problem is *metric TSP*, in which the distances between points is a metric. One source of metrics is edge-weighted undirected graphs; the *metric completion* of such a graph is the metric space over its vertices in which the $u$-to-$v$ distance is the length of a shortest $u$-to-$v$ path. Note that a tour in such a metric space corresponds to a closed walk in the graph that is allowed to travel through a vertex more than once. Thus the graph need not be Hamiltonian to admit a tour.

The leading TSP codes (Concorde [1] , `LKH-2` [14], and Google's `or-tools`) require that all vertex-to-vertex distances be available, i.e. all these distances are stored in a table or there is a procedure to compute any such distance given the two vertices. This works quite well for Euclidean instances because a point-to-point distance can be computed very quickly. For instances arising from edge-weighted graphs, this is a serious obstacle: for the graphs we want to address, the number of vertices is too large for an all-pairs-distance table. There is a publicly available implementation of a high-quality distance oracle (a data structure supporting fast vertex-to-vertex distance queries) that of Dibbelt et al. [9] (`RoutingKit`), based on the algorithm of Geisberger et al. [10]. In our experiments with this implementation, however, distance-finding is not fast enough to make the existing TSP codes competitive for very large graphs. Indeed, the running time of the Lin-Kernighan heuristic is reported [13] to increase as $n^{2.2}$, and in our experience with `RoutingKit`, the time per distance was 0.04 seconds.

Moreover, the primary motivation of our work is not to beat existing codes specifically addressing TSP but to illustrate the potential of an approach to optimization problems on planar graphs and to identify algorithmic techniques that could help make this approach useful in practice.

However, it is useful to have a baseline for tour quality in order to evaluate the effectiveness of our implementation on very large graphs. There is one other theoretical algorithm that runs in linear time and finds approximately optimal tours on edge-weighted planar graphs: the 2-approximation derived from a minimum-weight spanning tree. This algorithm finds a minimum-weight spanning tree and then obtains an Euler tour of the doubled tree. We found that the minimum-spanning-tree heuristic runs very fast on large instances but produces tours that are far worse than those produced by our implementation.

One natural enhancement to the minimum-spanning-tree algorithm is to "shortcut" the tour it produces: a segment of the tour that visits only vertices already visited can be replaced by a shortest path. However, this enhancement has two drawbacks. First, carrying out the enhancement on a very large graph requires substantially more time than our implementation, even when the distance oracle of Dibbelt et al. [9] is used to compute these shortest paths. Second, the quality of the tours produced on very large graphs derived from road maps is inferior to those found by our implementation. We outline these results in Section 4.2.

What about other approximation algorithms for TSP in graphs? Christofides' algorithm [7] gives a 1.5-approximation, but requires computing a minimum-weight $T$-join. We are aware of no code that can compute this without first computing all-pairs distance, though Barahona [2] gives a theoretical $O(n^{1.5} \log n)$ algorithm for planar graphs.

In independent work, Xia et al. [22] reported on experiments with a graph-based method for the *Steiner TSP*, finding a tour visiting a given subset of vertices. We briefly discuss their approach in Section 1.2 since it uses a term defined in that section.

**Approximating the Held-Karp lower bound.** How then can we evaluate the quality of the tours obtained by our implementation? We have implemented an algorithm to compute an approximately optimal Held-Karp lower bound. The algorithm uses the packing-covering framework (see, e.g., [23], also known as multiplicative-weights update) and a dynamic algorithm for minimum-weight cuts that is based on a min-cut algorithm [5] for planar graphs. The approximate solutions we obtain for Held-Karp are enough to show that, on large graphs derived from road maps, our implementation finds tours of good quality.

**Summary of our contributions.** Our main contributions are as follows:

- We have implemented a theoretically linear-time approximation scheme for TSP in edge-weighted planar graphs.
- Our implementation incorporates data structures and heuristics that turn a highly theoretical algorithm into a procedure that shows promise as a practical tool.
- For appropriate settings of the parameters (going beyond what theoretical analysis allows), our implementation processes a graph in less than a millisecond per vertex and produces a tour that is empirically within 5% to 15% of optimal.
- We have also developed a procedure for deriving lower bounds on TSP in planar graphs. This enables us to measure the quality of the tours produced by our implementation.

A live demonstration of our implementation applied to graphs derived from road maps is available at `http://tsp.cs.brown.edu`.

## 1.1 Overview

After reviewing some fundamentals, we give a brief overview of the algorithm we implement: the linear-time approximation scheme for TSP in planar graphs [15]. We observe that setting the parameters according to theoretical analysis would lead to impractical runtimes. We show that, even with tighter analysis of the constants hidden in the running time analysis, an implementation with theoretical guarantees would require tremendous computation power. For example, achieving a 1.5-approximation might require in excess of $2^{144}n$ comparisons. Next, we describe the design choices of our implementation in detail, and discuss extensive experimental results. Finally, we compare this implementation with Tazari and Müller-Hannemann's.

## 1.2 Preliminaries

We assume familiarity with TSP, graph algorithms, dynamic programming on branch decompositions, and approximation algorithms, including PTASs.

**Planar graphs.** Throughout, all graphs considered are planar and embedded. The *dual* of a planar graph $G$ is the graph whose vertices are the faces of $G$, with edges between faces which share a boundary edge. The *radial* of a planar graph $G$ is the bipartite graph whose vertices are the union of the vertices of $G$ and the faces of $G$, with edges between each incident vertex-face pair.

**Dynamic programming and branch decompositions.** A *branch decomposition* of a graph is a rooted binary tree and a bijection between leaves of the tree and edges of the graph. Each edge $e$ of the tree defines a *cluster* of graph edges, namely those edges corresponding to the tree leaves whose leaf-to-root path contains $e$. The *boundary* of a cluster is the set of all vertices with at least one incident edge within the cluster and one not within the cluster. The *width* of a branch decomposition is the maximum cardinality of any cluster's boundary. The *branchwidth* of a graph is the minimum width of any branch decomposition of the graph. By considering only interactions on the boundary, branch decompositions are amenable to dynamic programming.

A *sphere-cut decomposition* is a branch decomposition where, for each cluster, a Jordan curve intersects no edges and exactly the boundary vertices. This induces a cyclic order to the boundary. In a planar graph whose radial graph has radius $k$, a sphere-cut decomposition of width at most $k + 1$ can be found in linear time using Tamaki's heuristic [20].

In independent work, Xia et al. [22] very recently reported on experiments on solving the Steiner traveling salesman problem, finding a tour that visits a given subset of vertices (*terminals*). They started with a graph derived from road maps and pruned away vertices and edges not on shortest terminal-to-terminal paths. They then found an optimal sphere-cut decomposition (i.e. one whose width is the branchwidth of the pruned graph), and then used the decomposition to find an optimal tour. The algorithm [24] for finding an optimal sphere-cut decomposition, based on the algorithm of Seymour and Thomas [19], requires $O(n^3)$ time. Given a graph and a sphere-cut decomposition of width $w$, the algorithm of Xia et al. takes $O(7^{2w}n^2)$ time to compute the optimal tour.

Xia et al. reported on experiments finding an optimal tour using the sphere-cut decomposition and finding an approximately optimal tour using Christofides' 1.5-approximation algorithm. However, they addressed much smaller graphs; the largest graph they considered had 300 vertices and 363 edges.

**TSP.** Let $\mathsf{OPT}(G)$ be the minimum cost of a TSP tour of graph $G$ and $\mathsf{MST}(G)$ be the cost of a minimum spanning tree of $G$. It is well-known that $\mathsf{MST}(G) < \mathsf{OPT}(G) \leq 2\,\mathsf{MST}(G)$, giving a trivial 2-approximation algorithm. Christofides' algorithm [7] gives a 1.5-approximation, but requires computing a minimum-weight perfect matching, for which no nearly linear-time algorithm is known on planar graphs.

## 2 PTAS for TSP

We implement a slight variation of the Klein's PTAS [15]. In this section, we briefly summarize the algorithm. As input, we have a planar graph $G_0 = (V(G_0), E(G_0))$ equipped with an embedding and edge cost function $c(\cdot)$. We additionally have a precision parameter $\varepsilon \in (0, 1]$. The algorithm consists of four steps, described without the compromises necessary for fast runtimes:

1. **Cost reduction**: find an edge subgraph $G_1 \subseteq G_0$ of total cost at most $c(G_1) = (1 + 2/\varepsilon_1)\mathsf{MST}(G_0)$, for some constant $\varepsilon_1$ depending only on $\varepsilon$ such that $\mathsf{OPT}(G_0) \leq \mathsf{OPT}(G_1) \leq (1 + \varepsilon_1)\mathsf{OPT}(G_0)$. The subroutine Spanner provided by Klein [15] satisfies these requirements and is practical.
2. **Slab decomposition**: split the graph into a collection of subgraphs called *slabs*, each of which has branchwidth at most $2/\varepsilon_2 + 3$ such that each vertex appears in at least one slab, and the sum of the costs of optimal TSP tours on the slabs is at most $\mathsf{OPT}(G_1) + 2\varepsilon_2 c(G_1)$, where $\varepsilon_2$ is a constant depending only on $\varepsilon$.
3. **Dynamic programming**: for each slab, build a branch decomposition and solve TSP exactly on it. For each cluster in the decomposition, we build a table of *configurations* and their corresponding costs: all relevant interactions between the interior and the exterior of the cluster. An upper bound on the number of configurations per cluster of width $k$ is $M(k) = 3^{2k}$. Solving TSP exactly on a graph with a branch decomposition of width $k$, takes $O(3^{4k}n)$ time, since the DP performs pairwise compatibility checks of configurations from sibling clusters. (A tighter upper bound on the number of configurations per cluster is $M(k) = \sum_{i=0}^{k} C_{k-i}\binom{2k}{2i}$ where $C_j$ is the $j$th Catalan number.)
4. **Combining**: return the union of the exact solutions on the slabs.

The final output has total cost at most

$$\begin{aligned}\mathsf{OPT}(G_1) + 2\varepsilon_2 c(G_1) &\leq (1 + \varepsilon_1)\mathsf{OPT}(G_0) + 2\varepsilon_2(1 + 2/\varepsilon_1)\mathsf{MST}(G_0) \\ &< (1 + \varepsilon_1)\mathsf{OPT}(G_0) + 2\varepsilon_2\mathsf{OPT}(G_0) + 4\varepsilon_2/\varepsilon_1\mathsf{OPT}(G_0) \\ &= (1 + \varepsilon_1 + 2\varepsilon_2 + 4\varepsilon_2/\varepsilon_1)\mathsf{OPT}(G_0).\end{aligned}$$

## 3 Engineering Considerations

We benefit from the pessimism of worst-case analysis in several places. Before discussing how we engineer the algorithm, we note that the worst-case analysis is overly pessimistic. First, $\mathsf{MST}(G_0)$ frequently costs significantly less than $\mathsf{OPT}(G_0)$.

Second, frequently the branchwidth is less than the theoretical upper bound. Third, the total cost of slab boundaries is typically much less than $2\varepsilon_2 c(G_1)$. Furthermore, some edges in slab boundaries also belong to optimal solutions. Finally, the structure of the sphere-cut decompositions we use ensures that only rarely are two clusters merged in a way that requires considering a number of configuration pairs that is at all close to the theoretical upper limit.

Next we summarize our implementation. As part of the input, it takes parameters $\varepsilon_1$ and $\varepsilon_2$ separately, as their effects in practice differ from the theoretical guarantees.

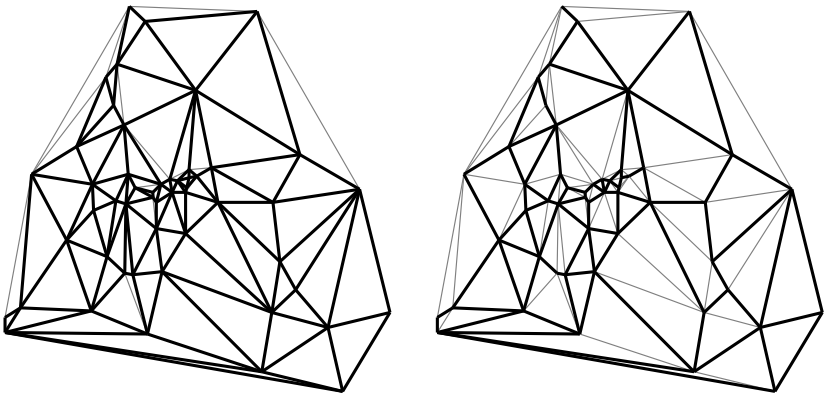


**Figure 1** Delaunay triangulation of TSPLIB's `berlin52`: bold edges indicate a spanner with $\varepsilon_1 = 0.1$ (left) and $\varepsilon_1 = 0.5$ (right).

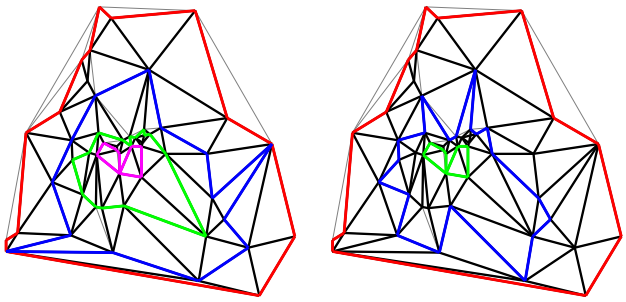
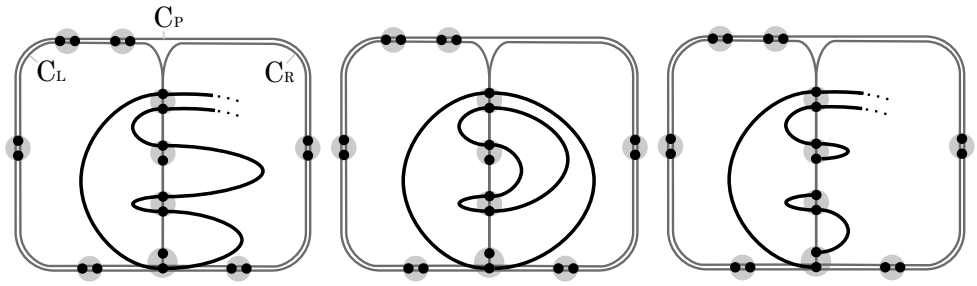


**Figure 2** `berlin52`: spanner in black ($\varepsilon_1 = 0.1$) and slab boundaries for $\varepsilon_2 = 1/3$ (left) and $\varepsilon_2 = 1/4$ (right).

**Cost reduction.** The implementation finds graph $G_1$ from $G_0$ as in [15]. On a planar graph, this can be done in linear time [6, 16]. In practice, though, we use Kruskal's algorithm and observe that the time spent building a minimum spanning tree is usually less than 0.001% of the total runtime under reasonable choices of $\varepsilon_2$. Refer to Figure 1 for examples of two spanners on a small graph.

**Slab decomposition.** The implementation performs a breadth-first search of the dual graph. This partitions the dual edges into two types: those with endpoints on the same level of the search tree (type A) and those with endpoints on different levels (type B). Each edge is assigned a level by the minimum level of its endpoints. Level interval $(i, j)$ consists of all type A edges with level in $[i + 1, j]$ and all type B edges with level in $[i, j]$. The type B edges of level $i$ form the *upper seam* and the type B edges of level $j$ form the *lower seam*. The edges in a level interval can induce a slab. The slabs used will have the property that the only edges shared between slabs are seams and the only vertices shared between slabs are incident to seam edges. Refer to Figure 2.

**Dynamic program.** Essentially all of the runtime is spent in the dynamic program. Because of this, most of the complexity in the implementation focuses on efficiently finding pairs of compatible configurations and merging compatible pairs into a parent configuration, and it is here that the choice of engineering techniques have the greatest impact.




**(a)** compatible **(b)** incompatible **(c)** incompatible

**Figure 3** Prefix configuration pairs for merging child clusters $C_L$ and $C_R$ into parent cluster $C_P$: gray circles represent boundary vertices, black circles represent portals, and black lines represent tour segments. Starting with the uppermost shared vertex and going down, the left prefix configuration [(,(,),-,(,),-,)] is (a) compatible with the right prefix configuration [(,(,(,-,),(,-,)], (b) incompatible with the right prefix configuration [(,(,(,-,),),-,)] because the inner prefix cycle indicates a disconnected tour, and (c) incompatible with the right prefix configuration [(,(,(,),-,(,),-] because the crossings do not align.

Recall, each non-root, non-leaf cluster in a sphere-cut decomposition has a sibling and a parent. Furthermore, since each cluster is bounded by a Jordan curve, a natural cyclic order is assigned to the cluster's boundary vertices.

Since a TSP tour can enter or exit a cluster at most twice per vertex (subsequent crossings can be uncrossed), we split each boundary vertex into two *portals*, representing these potential connections. An involution is stored mapping portals to portals: each portal is associated with another (or to itself, when there is no entrance/exit at the portal). This involution is stored in two different ways, depending on the context: either as small integers representing the portal number or using nested parentheses (really, an array of enum objects) where matching parentheses map to one another (since TSP tours in planar graphs can be uncrossed).

The *prefix* of a cluster's portals is the interval of portals common to both children in the cyclic order induced by the cluster's bounding Jordan curve. Whether two child-cluster configurations are *compatible* can be determined mostly by comparing the section of the configurations corresponding to the prefix portals. Cycles formed between prefix portals not shared with the parent cluster indicate incompatible configurations because the final tour must be connected. Additionally, the child-cluster configurations must agree on the presence of a crossing at a prefix portal. That is, if a portal is mapped to itself on one side, it is mapped to itself on the other.[1] Refer to Figure 3 for examples of prefix compatibility.

In practice, computing the entire dynamic programming table is prohibitively expensive: merging two clusters with boundary size just 5 theoretically requires considering over 20 times more pairs of configurations than there are vertices in the largest road network publicly available for testing; in order for the algorithm to "act" like it is linear time, clusters must discard some configurations. Tazari and Müeller-Hannemann face a similar problem with large dynamic programming tables and impose a cap on the size of their dynamic programming table. We follow this strategy and limit each cluster to hold the $\lambda$ best configurations found, plus one corresponding to the MST-based 2-approximation of the original graph to ensure that there is always a solution.

[1] For technical reasons, vertices common to the parent and both children pose additional complication.

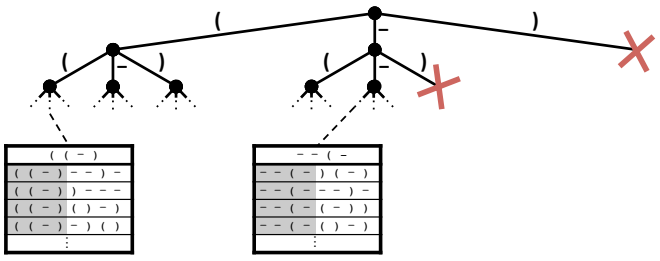


**Figure 4** Each leaf of the trie corresponds to the prefix configuration generated by the root-to-leaf path. Configurations are grouped by prefix and stored (sorted by cost) at these leaves. Two such leaves are shown. The crossed-off trie branches represent invalid prefix configurations.

Rather than generating all pairs of compatible configurations and selecting the $\lambda$ best, we generate them in order of increasing cost as follows. The configurations for each cluster are partitioned such that if a configuration is compatible with one configuration in a part, it is compatible with each other configuration in the part. Partitioning by equivalence classes on prefixes suffices. These parts can be efficiently stored as lists sorted by non-decreasing cost at the leaves of a trie. Each root-to-leaf path is the prefix in the nested-parenthesis representation common to all the configs stored at the leaf (refer to Figure 4). Once compatible pairs of leaves are found by traversing the tries for the child configurations in tandem, pairs of pointers to the lists' first elements are inserted into a min-heap keyed by the sum of the costs of the pointed-to elements. To get the next cheapest configuration, one pops the heap. Then, the appropriate pointer is incremented and the pair is re-inserted into the heap.

Note that just popping the heap $\lambda$ times is insufficient, as some of these pairs might yield the same parent configurations. Instead, the heap is popped and parents are formed until $\lambda$ have been collected. The actual formation of parent configurations from a compatible pair of child configurations is delayed until necessary, as this transformation turns out to be a bottleneck.

**Post processing.** As a post-processing heuristic, we draw inspiration from [8]: some number of tours, determined by an input parameter, are produced by running the full algorithm with several different slab decompositions. We then make a new graph from the union of the edges used in these tours and run the PTAS on this (unlike [8], which would just run the dynamic program on it).

Tours output by the PTAS typically are very suboptimal around slab boundaries. Recursively re-solving on the graph induced by the set of edges occurring in the tour is effective. The maximum permitted recursion depth is another parameter of the implementation.

## 4 Experimental Results

Our implementation consists of about 5000 lines of `C++11` with no external dependencies. We use the `g++` compiler, version 5.2.0, on the Debian 8 operating system.

We use two types of graphs for testing:

- Road networks from OpenStreetMap [12] of the major American cities of Tulsa, Dallas, Los Angeles, Rochester, and Chicago. Crossing edges are planarized by introducing a new vertex at the crossing point.
- Synthetic instances: grids with each degree-4 face randomly triangulated and random small integer costs assigned to each edge.

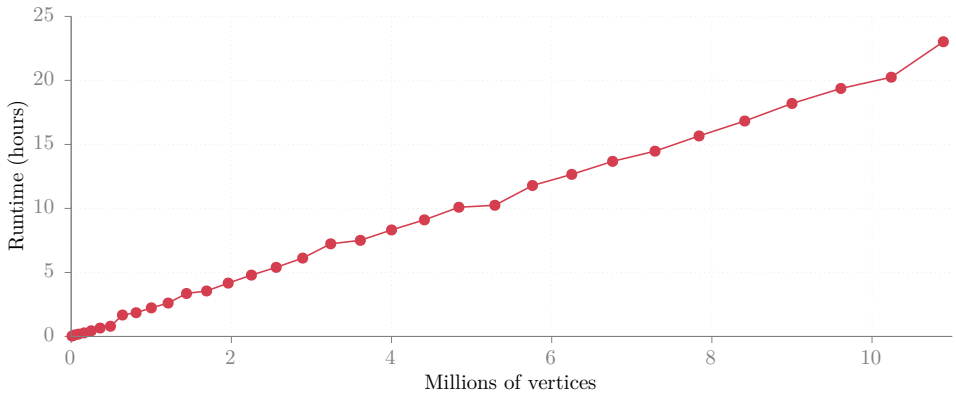



**Figure 5** Runtime is linear in the size of the graph.

**Table 1** Comparison of the performance of four heuristics on road networks.

| Graph | # Vertices | LB | 2MST | | Shortcut 2MST | | Fast PTAS | | Slow PTAS | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | val/LB | ms/v | val/LB | ms/v | val/LB | ms/v | val/LB | ms/v |
| `rochester` | 19488 | 100746068 | 1.41 | <0.01 | 1.31 | 0.06 | 1.11 | 0.15 | 1.03 | 4.33 |
| `tulsa` | 68335 | 65840000 | 1.45 | <0.01 | 1.34 | 0.22 | 1.11 | 0.23 | 1.04 | 3.41 |
| `dallas` | 403393 | 36332200 | 1.57 | <0.01 | 1.45 | 2.17 | 1.34 | 0.24 | 1.15 | 4.05 |
| `chicago` | 1032016 | 31782700 | 1.49 | <0.01 | 1.38 | 5.90 | 1.32 | 0.48 | 1.09 | 5.83 |
| `losangeles` | 1135323 | 53903389 | 1.44 | <0.01 | 1.35 | 6.83 | 1.25 | 0.34 | 1.09 | 2.24 |

## 4.1 Linear Runtime

Our implementation exhibits linear runtime. The figure below shows the running time of the algorithm, with an arbitrary, realistic choice of parameters, on a series of synthetic square grids as described above.

Over 99.99% of the time on large instances is spent in the dynamic program; a plot of the runtime breakdown would be uninteresting.

## 4.2 Quality

There are two aspects of evaluating the quality of the tours returned by our algorithm: how close to optimal the tours are and how our solutions compare to other implementations. To address the former, we compute lower bounds on tour lengths, as described in the next section. For large graphs however, the latter point poses a problem. As discussed in the introduction, leading TSP implementations require all-pairs distances, which is infeasible for very large non-Euclidean instances. We compare the performance of our implementation with two different MST-based heuristics:

- The **2MST** heuristic doubles the edges of the minimum-spanning-tree.
- The **Shortcut 2MST** heuristic follows the tour of the 2MST heuristic but takes shortcuts to avoid unnecessarily re-visiting vertices.
- **Fast PTAS** is our implementation with a quicker-running set of parameters
- **Slow PTAS** is our implementation with a slower-running set of parameters

The ratios of tour lengths to lower bounds, given in Table 1 and depicted in Figure 6, provide upper bounds for solution error. We additionally report the runtime in milliseconds

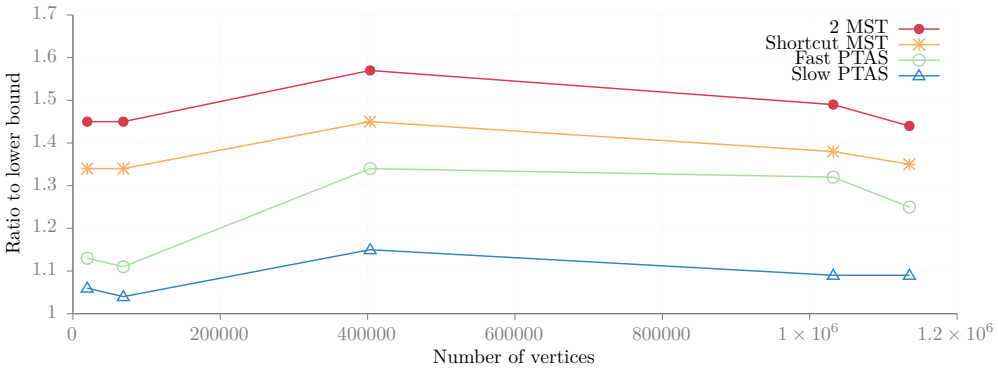



**Figure 6** Visual comparison of heuristic quality on road networks.

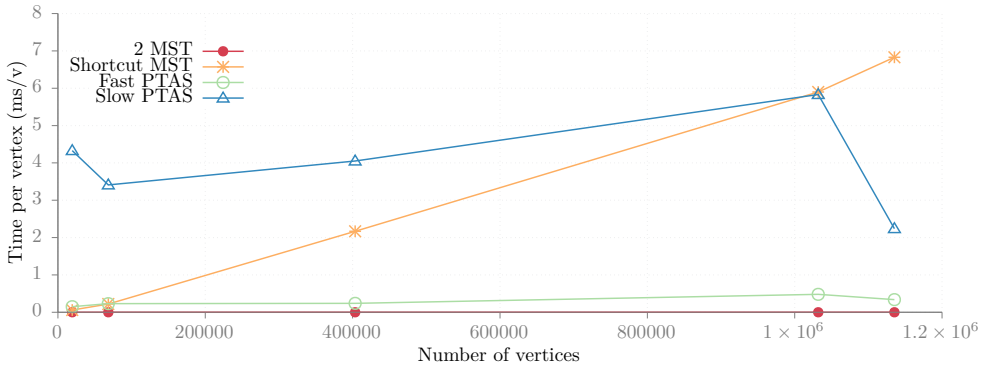



**Figure 7** Visual comparison of heuristic runtime per vertex on road networks.

per vertex (refer to Figure 7). The 2MST heuristic runs extremely quickly even on very large graphs but provides a poor approximation. The Shortcut 2MST heuristic slightly outperforms the basic 2MST heuristic but takes much longer to find (the running time is superlinear in graph size). Our Fast PTAS tours are found very quickly and show a substantial improvement over 2MST, and our Slow PTAS tours are close to optimal.

## 4.3 The Effects of Parameters on Performance

To explore the effects of various parameters on runtime and tour cost, we ran a parameter sweep across six graphs and a variety of settings of each of four parameters: **slab height** ($1/\varepsilon_2$), number of **configurations** ($\lambda$), number of **re-solves**, and number of **tour unions**. We examined each parameter separately to identify trends in the effects on runtime and tour cost. In particular we wanted to identify parameter settings that exhibited a promising cost-runtime tradeoff. Figures 8, 9, 10, and 11 show the parameter effects on performance for the Los Angeles road network.

Figure 8 shows the effect of the **slab height** parameter. Recall that in Step 2 of the approximation scheme (Section 2, see also Section 3), the graph is decomposed into slabs. Each slab consists of a consecutive sequence of levels of a breadth-first search. The *slab height* is the number of levels comprising each slab.

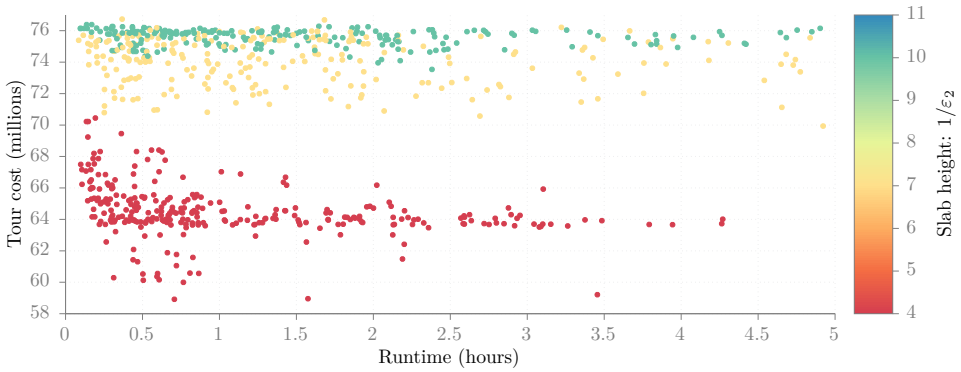



**Figure 8** Performance by slab size for Los Angeles road network.

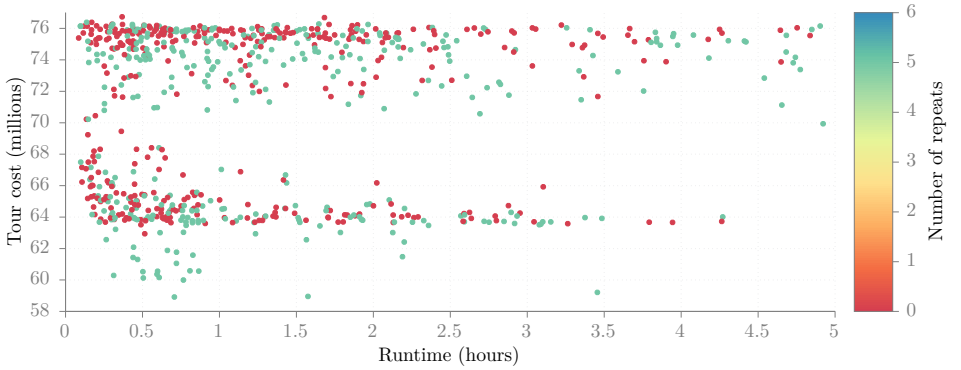



**Figure 9** Performance by number of tour re-solves for Los Angeles road network.

Figures 9 and 10 shows the effect of the **re-solves** and **tour unions** parameters respectively. Recall (see *Post Processing* in Section 3) that the implementation finds several tours using different slab decompositions (all with the same height), takes the union, and repeats on the union. The number of *tour re-solves* is the number of times this happens. The *number of tour unions* refers to the size of the union taken in each repeat.

Finally, Figure 11 shows the effect of the number of **configurations** parameter. Recall that in the dynamic program, for each cluster of the branch decomposition, the implementation limits the size of the table of configurations stored for that cluster. The limit is $\lambda$, the *number of configurations*.

The number of retained configurations, $\lambda$, appears to have only a weak association to tour quality, but very fast runtimes require small $\lambda$ values. Recall that the algorithm returns a tour composed of the union of slab tours which is often very suboptimal at the slab *seams*; re-solving on the graph induced by edges of the initial resulting tour (and iterating several times) can greatly improve tour quality with minimal increase in runtime. Similarly, taking the union of several solutions and re-solving on the resulting graph comprised of the union of the tours also improves tour quality. In both of these post-processing strategies the branchwidth of the graph used to re-solve TSP is typically much smaller than that of the original graph, which both substantially changes the slab decomposition and decreases the runtime.

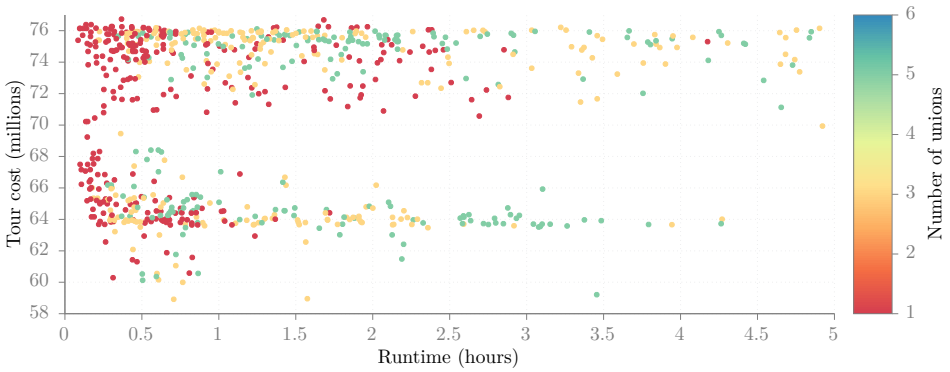



**Figure 10** Performance by number of tour unions for Los Angeles road network.

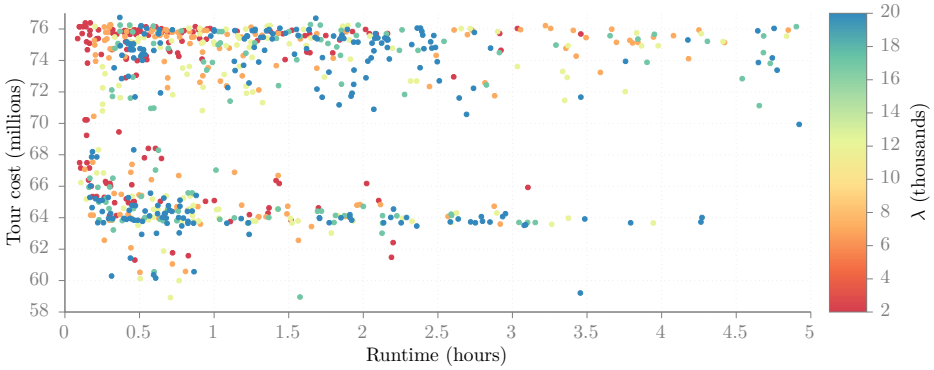



**Figure 11** Performance by number of configurations for Los Angeles road network.

Interestingly and unexpectedly, larger slab heights (smaller $\varepsilon_2$) produce *worse* solutions. We attribute this to all values of $\lambda$ being too small: the configurations kept in any cluster are only a tiny subset of potential configurations, increasing the odds of missing an important one.

Overall, we see that some parameters (such as number of repeats and unions) have clear benefits to tour quality whereas other parameters have more complicated and intricate effects and potential dependencies.

## 5 Computing a Lower Bound on the Traveling-Salesman Tour

We needed a way to evaluate the quality of the tours found by our code. Other implementations (e.g. Concorde and LKH) include subroutines for computing lower bounds but none supported finding a lower bound on a graph with many vertices where distances between vertices take more than a few hundred nanoseconds to compute.

## 5.1 The Mathematical Program

We wrote a procedure to find an approximately optimal solution to the dual of the linear program (LP) that optimizes over the subtour elimination polytope. This LP is:

$$\min\ c \cdot x\ :\ x \geq 0,\ \sum\{x_e\ :\ e \in \delta(S)\} \geq 2 \text{ for every nontrivial subset } S \subsetneq V$$

where there is a variable $x_e$ for each edge and a constraint for each nontrivial *cut* in the graph. A nontrivial cut is the set of edges between the two parts of a bipartition of the vertices. For a subset $S$ of vertices, $\delta(S)$ is the set of edges between $S$ and $V - S$.

The above LP is a relaxation of TSP: any tour induces an LP solution of the same value. Therefore, the value of the LP is at most the value of the best tour.

Our procedure computes a solution to the dual of the above LP, namely

$$\max\ \vec{2} \cdot y\ :\ y \geq 0,\ \sum\{y_S\ :\ e \in \delta(S)\} \leq c_e \text{ for every edge } e.$$

This LP has a variable $y_S$ assigning a weight to every cut $\delta(S)$. For each edge $e$, the total weight of cuts containing $e$ is required to be at most the cost of $e$. The goal is to maximize twice the sum of the cut edges. This is called a *packing of cuts*. By LP duality, the value of this LP equals the value of the LP with the subtour elimination constraints.

## 5.2 Approximation Scheme

Our procedure approximates the value of the packing LP using an approximation scheme of Young [23] for solving fairly general mathematical programs (*packing/covering*) via solving a sequence of simpler mathematical programs. In this application of the method, in each iteration the procedure must find a cut whose weight is less than a threshold. The weights are adjusted in each iteration. In particular, in each iteration the procedure increases the weights of edges in the cut just selected, and adjusts the threshold. The number of iterations grows as $O(\epsilon^{-2} m \log m)$ where $m$ is the number of edges. Each iteration of the implementation takes a step that is larger than that prescribed by theory; the implementation uses binary search to find the largest step size that preserves the algorithm's invariant.

The main work in each iteration is to find a cut of weight less than the threshold. There is a near-linear-time algorithm [5] for min-weight cut in planar graphs. The algorithm uses shortest-path separators, divide-and-conquer, and an $O(n \log n)$ algorithm for minimum $st$-cut in a planar graph. We implemented this algorithm but using it to implement an iteration is far too slow for our purposes. We therefore used it as the basis for a dynamic min-cut algorithm.

## 5.3 Dynamic Algorithm for Min-Weight Cut in Planar Graphs

The divide-and-conquer algorithm forms a balanced binary tree, a recursive-decomposition tree: each internal node has an associated min $st$-cut instance on a subgraph, and each leaf has an associated global min-cut instance. The dynamic algorithm maintains a priority queue of solutions to these instances, ordered according to the weights of the solutions. However, the algorithm does not automatically update the solutions or the priority queue when edge-weights increase.

When the LP algorithm requests a cut of weight less than a threshold, the dynamic algorithm examines the cut in the priority queue whose key is smallest, and computes the true weight of the cut (i.e. with respect to current edge-weights). If the true weight is less than the threshold, the dynamic algorithm returns it; if not, the algorithm puts the

corresponding instance in a queue of instances to reprocess, and moves on to the next cut in the priority queue. Once the cuts in the priority queue are exhausted and no cut of weight less than the threshold has been found, the algorithm turns to the queue of instances to reprocess; it selects the smallest of these instances and recomputes the corresponding cut. If that cut's weight is still not less than the threshold, the algorithm goes to the next larger instance, and so on. If this queue is exhausted, the algorithm starts from scratch, recomputing shortest-path separators and the recursive-decomposition tree.

### 5.4 Experiments

As predicted by theory, the runtime of the lower bound procedure depends quadratically on the inverse of the precision parameter $\epsilon$. Also as predicted by theory, the number of iterations grows as $O(n \log n)$. The runtime appears to scale slightly superlinearly with the size of the graph, illustrating the empirical effectiveness of our dynamic min-cut algorithm.

The data are shown in the Appendix A.

## 6 Discussion

We implemented a PTAS for TSP on planar graphs designed to handle instances with millions of vertices in a reasonable amount of time. Our analysis demonstrates that, despite the significant hurdles presented by massive constants obscured in asymptotic notation, with a bit of engineering, highly theoretical algorithms can become practical. Data used for the road-map experiments will be made available at `http://tsp.cs.brown.edu`.

**Comparison with Steiner tree.** Like [21], we found that the maximum table size, $\lambda$, has a major effect on runtime and quality. Implementing heuristics for pruning tables more effectively seems like a viable strategy toward engineering a better, faster implementation.

In the "Conclusion and Outlook" section [21], Tazari and Müller-Hannemann write that it "would be very interesting to see how the PTAS performs for [TSP] and especially, [sic] if one can drastically reduce the required table sizes using well-known lower and upper bounds for the TSP." Unfortunately, it seems that the additional complexity of configurations in the dynamic program for TSP (compared to those in Steiner tree) means that tables are *not* able to be significantly smaller.

In Steiner tree, the configuration for a cluster boundary can be represented completely by a non-crossing partition of the vertices of the boundary; this bounds the number of configurations for a cluster of width $k$ at $C_k$, which is significantly smaller than the $\sum_{i=0}^{k} C_{k-i}\binom{2k}{2i}$ we encounter. For example, with a boundary size of 5, there are at most 42 Steiner tree configurations and over 2,000 TSP configurations. Furthermore, actually combining configurations or determining if they are compatible is significantly more complicated.

**Potential improvements.** Our tours on the larger graphs might be better relative to the optimum than we have reported: running the lower-bound code for longer will improve those lower bounds.

The dynamic program can be easily parallelized. Considerable speedup might therefore be achieved on multi-core processors by using parallel processing.

Racing several techniques to solve the slabs and taking whichever finishes first could provide a significant speedup; although the slabs can contain $O(n)$ vertices, frequently they are small and could be solved more quickly with other techniques, e.g. local search. Simply post-processing the tour using local search is likely to significantly improve the tour length.

Upper and lower bounds could be used to prune away entire subtrees of the trie storing configurations. A deeper understanding of which branch decompositions lead to good solutions given a small $\lambda$ value might improve quality significantly. Additionally, different algorithms may yield lower-width branch decompositions; experimenting with these may be fruitful.

Integrating our lower-bound computations into the dynamic program could prove fruitful by determining which clusters have near-optimal-cost configurations and which need more time invested. (Currently the lower-bound code is too slow to serve in that way.)

Tamaki's heuristic [20] can be generalized [11] to handle planar hypergraphs. It might be possible to use this to handle the localized nonplanarities that arise in road maps.

**Adaptation to related problems.** As discussed earlier, this is a first step in providing a robust implementation capable of handling a suite of related problems, such as Steiner TSP. To adapt to these other problems, one might need a more involved spanner step, as well as to modify the configuration compatibility-checking code. Most of the engineering techniques applied here will translate, often without modification, to related problems.

## References


1 D. Applegate, R. Bixby, V. Chvatal, and W. Cook. Concorde TSP solver, 2006.

2 F. Barahona. Planar multicommodity flows, max cut, and the chinese postman problem. In *Polyhedral Combinatorics, Proceedings of a DIMACS Workshop, Morristown, New Jersey, USA, June 12-16, 1989*, pages 189–202, 1990.

3 J. J. Bartholdi and L. K. Platzman. Heuristics based on spacefilling curves for combinatorial problems in euclidean space. *Management Science*, 34(3):291–305, 1988.

4 G. Borradaile, P. N. Klein, and C. Mathieu. An $O(n \log n)$ approximation scheme for steiner tree in planar graphs. *ACM Trans. Algorithms*, 5(3):31:1–31:31, 2009.

5 P. Chalermsook, J. Fakcharoenphol, and D. Nanongkai. A deterministic near-linear time algorithm for finding minimum cuts in planar graphs. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*, pages 828–829, 2004.

6 D. Cheriton and R. E. Tarjan. Finding minimum spanning trees. *SIAM Journal on Computing*, 5(4):724–742, 1976.

7 N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon University, 1976.

8 W. Cook and P. Seymour. Tour merging via branch-decomposition. *INFORMS Journal on Computing*, 15(3):233–248, 2003.

9 J. Dibbelt, B. Strasser, and D. Wagner. Customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 21(1):1.5:1–1.5:49, 2016.

10 R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.

11 Q. P. Gu and H. Tamaki. Improved bounds on the planar branchwidth with respect to the largest grid minor size. *Algorithmica*, 64(3):416–453, 2012.

12 M. Haklay and P. Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, 2008.

13 K. Helsgaun. An effective implementation of the lin-kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

14 K. Helsgaun. General k-opt submoves for the Lin–Kernighan TSP heuristic. *Mathematical Programming Computation*, 1(2-3):119–163, 2009.

15 P. N. Klein. A linear-time approximation scheme for TSP in undirected planar graphs with edge-weights. *SIAM Journal on Computing*, 37(6):1926–1952, 2008.

16 T. Matsui. The minimum spanning tree problem on a planar graph. *Discrete Applied Mathematics*, 58(1):91–94, 1995.

17 M. Müller-Hannemann and S. Schirra, editors. *Algorithm engineering: bridging the gap between algorithm theory and practice*, volume LNCS 5971. Springer, 2010.

18 G. Reinelt. Fast heuristics for large geometric traveling salesman problems. *ORSA Journal on Computing*, 4(3):206–217, 199.

19 P. Seymour and R. Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.

20 H. Tamaki. A linear time heuristic for the branch-decomposition of planar graphs. In *Algorithms – ESA 2003*, volume 2832 of *Lecture Notes in Computer Science*, pages 765–775. Springer, 2003.

21 S. Tazari and M. Müller-Hannemann. Dealing with large hidden constants: Engineering a planar Steiner tree PTAS. *Journal of Experimental Algorithmics (JEA)*, 16:3–6, 2011.

22 Y. Xia, M. Zhu, Q. Gu, L. Zhang, and X. Li. Toward solving the steiner travelling salesman problem on urban road maps using the branch decomposition of graphs. *Information Sciences*, 374:164–178, 2016.

23 N. E. Young. Sequential and parallel algorithms for mixed packing and covering. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 538–546, 2001.

24 M. Zhu. Computational study on branch decompositions of planar graphs. Master's thesis, School of Computing Science, Simon Fraser University, 2013.


## A Experiments with lower-bound procedure

As predicted by theory, the number of iterations grows as $O(n \log n)$. Here the curve is $50\ n\ \log n$:

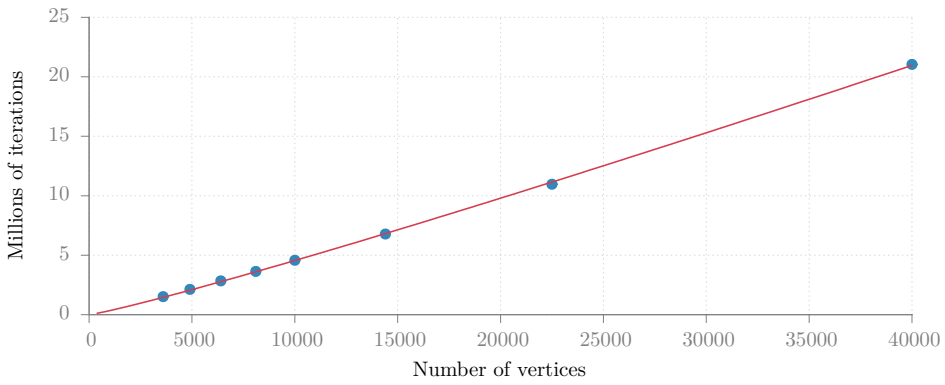



**Figure 12** Lower-bound code: Number of interations grows as $50\ n\ \log\ n$ for graph size $n$.

The runtime in minutes of several runs with different values of $\varepsilon$ on a road network of Rochester, NY is illustrated, along with the curve $0.008/\varepsilon^2 - 0.18$.

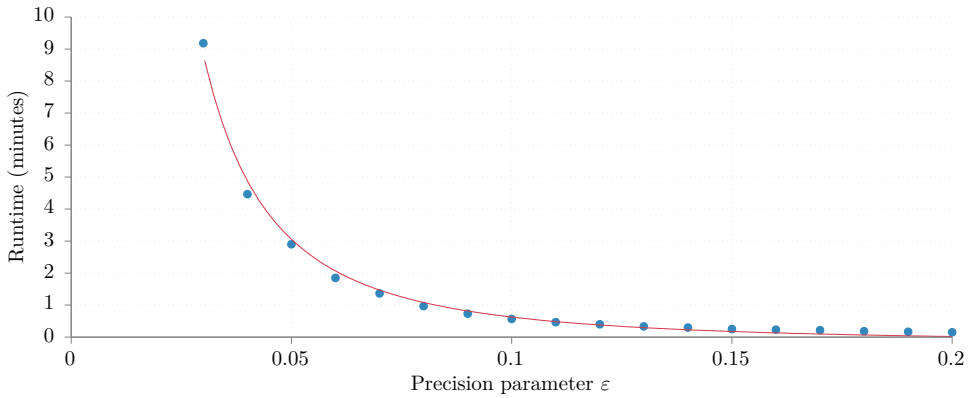



**Figure 13** Lower-bound code: Runtime shrinks as $0.008/\varepsilon^2 - 0.18$ for precision parameter $\varepsilon$.

As visualized below, the runtime of the lower bound code appears to scale slightly superlinearly with the size of the graph. This shows the empirical effectiveness of our dynamic-min-cut algorithm. This plot shows runtime on a variety of synthetic grids with $\varepsilon = 0.05$.

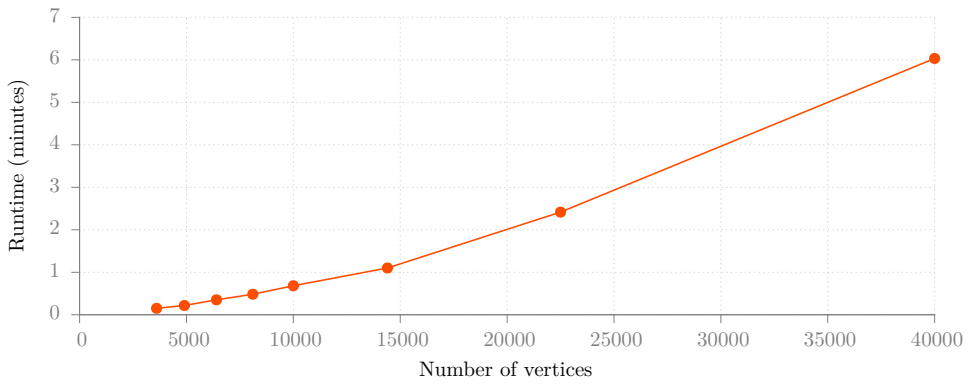



**Figure 14** Lower-bound code: Runtime as a function of graph size.