Drawing Graphs to Speed Up Shortest-Path Computations*

D. Wagner and T. Willhalm[†]

Abstract

We consider the problem of (repeatedly) computing single-source single-target shortest paths in large, sparse graphs. Previous investigations have shown the practical usefulness of geometric speed-up techniques that guarantee the correctness of the result for shortest-path computations. However, such speed-up techniques utilize a layout of the graph which typically comes from geographic information. This paper examines the question how geometric speed-up techniques can be used in case there is no layout given. We present an extensive computational study analyzing the usefulness of methods from graph drawing as foundation for such techniques. It turns out that using appropriate layout algorithms, a significant speed-up can be achieved.

1 Introduction

Single-source single-target shortest-path computation is a fundamental algorithmic problem with manifold applications. Especially, the computation of shortest-path queries in large graphs is a common task in many application scenarios. We are particularly interested in a situation where the graph is fairly large, but sparse and an expensive preprocessing is feasible. This typically arises in routing systems where a central server operates on a static graph that is too large to admit storage of shortest paths between all pairs of vertices.

Without any preprocessing, Dijkstra's algorithm [5] is the fastest known algorithm for the general case of arbitrary non-negative edge lengths, taking $O(m + n \log n)$ worst-case time. In [16, 24, 25, 11] however, it has been shown that a considerable speed-up for the query time can be obtained for graphs with a given layout by using the according geometric information. The aim of this paper is to examine, if these techniques can be also utilized in the absence of a given layout. Applying methods from graph drawing, layouts are generated and used as foundation for geometric speed-up techniques.

In [4], a related question has been studied for the special case of a timetable information system. A scenario is considered where the geographic information typically contained in timetable data is incomplete. Our results are more general with respect to the graphs considered, as well as the layout algorithms explored. We experiment with real world graphs from different areas and with various generated graphs. In particular, in contrast to [4] no additional information is used that might support the layout algorithms (like movement of trains or coordinates of selected stations).

The main contribution of this paper consists in a computational study demonstrating that artificially produced layouts can indeed be used as basis for geometric speed-up techniques for shortest-path computations. For several of the graphs explored, significant speed-ups are achieved with appropriately generated layouts. Moreover surprisingly, for many of the tested instances where layouts based on geographic information were available, the generated layouts resulted in an even better speed-up.

After reviewing the basic definitions, Sect. 3 introduces the geometric speed-up techniques examined in our experiments. In Sect. 4, we describe the graph drawing algorithms considered. The experiments and results are shown in Sect. 5 before the conclusion in the last section.

2 Definitions

An *(undirected) graph* G is a pair (V, E), where V is a finite set of *nodes* and E is a set of *edges*, where an edge is an unordered pair $\{u, v\}$ of nodes $u, v \in V$. If the edges are ordered pairs (u, v) of nodes $u, v \in V$, we call the graph *directed*. Throughout this paper, the number of nodes $|V|$ is denoted by n and the number of edges $|E|$ is denoted by m. A graph can have up to $O(n^2)$ edges. We call a graph *sparse*, if $m = O(n)$, and we call a graph *large*, if one can only afford a memory consumption in $O(n)$. In particular, for large sparse graphs $O(n^2)$ space is not affordable.

A *path* in G is a sequence of nodes $(u_1, \ldots, u_k)$ such that $\{u_i, u_{i+1}\} \in E$ for all $1 \leq i < k$. If G is directed, a path must respect the direction of the

†Universität Karlsruhe, Fakultät für Informatik, Institut für Logik, Komplexität und Deduktionssysteme, D-76128 Karlsruhe

edges, i.e. $(u_i, u_{i+1}) \in E$ for all $1 \leq i < k$. Given edge weights or lengths $l : E \to \mathbb{R}_0^+$, the *length of a path* $P = (u_1, \ldots, u_k)$ is the sum of the lengths of its edges $l(P) := \sum_{1 \leq i < k} l(u_i, u_{i+1})$. A *shortest s-t-path* is a path of minimal length with $u_1 = s$ and $u_k = t$. A *layout* of a graph is a function $L : V \to \mathbb{R}^2$ that assigns each node a position in the Euclidean plane.

3 Speed-Up Techniques

In this section, we present the three geometric speed-up techniques for Dijkstra's algorithm that are considered to test the generated layouts. All of them return a shortest path regardless of the layout that is used. However, the running time heavily depends on the layout. In the worst case, the algorithm with "speed-up" technique will take longer than without it, because additional calculations are carried out.

Our baseline algorithm is the bidirectional variant of *Dijkstra's algorithm with binary heaps*. For sparse graphs, the asymptotic running time for Dijkstra's algorithm with binary heaps is $O(n \log n)$. Moreover, for our application, *binary heaps* have been shown to be efficient in practice [22]. The *bidirectional search* simultaneously applies the "normal", or forward, variant of the algorithm, starting at the source node s, and a so-called reverse, or backward, variant of Dijkstra's algorithm, starting at the target node t with reversed edges $E^{rev} = \{(u, v) \mid (v, u) \in E\}$. The forward and the backward search process nodes depending on the size of their respective priority queues. The algorithm can be terminated when one node has been designated to be permanent by both the forward and the backward algorithm. As this speed-up technique can be applied without any preprocessing, it forms a reasonable baseline. The following geometric speed-up techniques can then be added individually or in combination to this bidirectional search.

3.1 Goal-Directed Search
Goal-directed search or A^* can be found in many text books (e.g., see [1, 21]). For a specific query from s to t, it modifies the edge lengths such that the search is driven towards the target t. More precisely, the weight of an edge $(u, v) \in E$ is changed to $l'(u, v) := l(u, v) + h(v) - h(u)$ with a so-called heuristic $h : V \to \mathbb{R}_0^+$. If, for each node $v \in V$, the heuristic $h(v)$ provides a lower bound for the length of a shortest v-t-paths, all modified edge length $l'(u, v)$ are non-negative. It is easy to show that in this case a path from s to t is a shortest s-t-path according to l, if and only if it is a shortest s-t-path according to l'.

Given a layout $L : V \to \mathbb{R}^2$, a lower bound for the distance from u to t can be determined as $h(u) = \frac{\|L(u) - L(t)\|}{v_{\max}}$ where $\|L(u) - L(t)\|$ denotes the Euclidean

distance of $L(u)$ and $L(t)$ and $v_{\max}$ is the maximum "velocity" $v_{\max} := \max\{\frac{\|L(u) - L(v)\|}{l(u,v)} \mid (u, v) \in E\}$. The maximum velocity can be computed in a preprocessing step by a linear scan over all edges.

3.2 Geometric Shortest-Path Containers
A *geometric shortest-path container* (see [24, 25]) is a geometric object for an edge (u, v) that contains all nodes $t \in V$ to which a shortest path starts with the edge (u, v). More precisely, we first determine, for each edge $(u, v) \in E$, the set $S(u, v)$ of all nodes $t \in V$ to which a shortest u-t-path starts with the edge (u, v). For a layout $L : V \to \mathbb{R}^2$, the geometric shortest-path container associated to (u, v) is the bounding box of $\{L(t) \mid t \in S(u, v)\}$, i.e. the smallest rectangle parallel to the axes that contains $\{L(t) \mid t \in S(u, v)\}$. We will denote this geometric shortest-path container by $C(u, v)$.

It is easy to verify that a shortest-path computation can be restricted to edges (u, v), where the target t is inside $C(u, v)$: If (u, v) is part of a shortest s-t-path, then its sub-path from u to t is also a shortest path. Therefore, t must be inside $C(u, v)$, because all nodes to which a shortest path starts with (u, v) are located inside $C(u, v)$. The restriction of the graph can be realized on-line during the shortest-path computation by excluding edges whose geometric shortest-path container does not contain the target node.

The geometric shortest-path containers can be computed beforehand by running a single-source all-target shortest-path computation for every node. The preprocessing for sparse graphs needs therefore $O(n^2 \log n)$ time and $O(n)$ space.

If you apply geometric shortest-path containers to bidirectional search, a second set C^{rev} of shortest-path containers is useful for the backward search. The "reversed" shortest-path containers C^{rev} can be determined by the same algorithm using the reversed edge set E^{rev}.

3.3 Reach-Based Routing
A fairly recent approach prunes the search space based on a centrality measure called *reach* [11]. Given a weighted graph $G = (V, E), l : E \to \mathbb{R}_0^+$ and a shortest s-t-path P, the *reach on the path* P of a node v is $r(v, P) := \min\{l(P_{sv}), l(P_{vt})\}$ where P_{sv} and P_{vt} denote the sub-paths of P from s to v and from v to t, respectively. The *reach* $r(v)$ of $v \in V$ is defined as the maximum reach for all shortest s-t-path in G containing v.

In a search for a shortest s-t-path P_{st}, a node $v \in V$ can be ignored, if (1) the distance $l(P_{sv})$ from s to v is larger than the reach of v and (2) the distance $l(P_{vt})$ from v to t is larger than the reach of v. While

performing Dijkstra's algorithm, the first condition is easy to check, since $l(P_{sv})$ is already known. The second condition is fulfilled if the reach is smaller than a lower bound of the distance from v to t. We used the same lower bound as for the goal-directed search. The reach can directly be used in the backward direction of the bidirectional search, too. In the backward search, $l(P_{vt})$ is already known whereas we have to use a lower bound instead of $l(P_{sv})$ for the first condition.

To compute the reach for all nodes, we perform a single-source all-target shortest-path computation for every node. With a modified depth first search on the shortest-path trees, it is easy to compute the reach of all nodes using the following insight: For two shortest paths P_{sx} and P_{sy} with a common node $v \in P_{sx}$ and $v \in P_{sy}$, we have $\max\{r(v, P_{sx}), r(v, P_{sy})\} = \min\{l(P_{sv}), \max\{l(P_{vx}), l(P_{vy})\}\}$. The preprocessing for sparse graphs needs therefore $O(n^2 \log n)$ time and $O(n)$ space. (We use exact values for the reach and not the bound provided by [11], since we admit an all-pairs shortest-path computation for shortest-path containers, too.)

4 Graph Drawing

In order to perform our computational study, we use the three common methods to draw graphs that have been shown to produce good results even for large graphs: multi-level force-directed layout, eigenvectors of the Laplacian matrix, and principal component analysis (PCA) of a high-dimensional embedding. Carefully implemented, all three graph-drawing algorithms show a near-linear running time in practice. The graphs were drawn as undirected graphs ignoring the direction of the edges, because preliminary results suggest that a drawing of the directed graph does not improve the quality of the result.

4.1 Multi-level Force-Directed Layout Eades presented in [6] the idea to draw a graph by simulating the edges of the graph as springs and the nodes of the graph as rings connecting these springs. The approach has been refined later by adding repelling forces to avoid small distances between nodes [7, 17]. While these algorithms produce appealing drawings, their running time is unacceptable for large graphs.

Three groups discovered independently [8, 26, 13] how a force-directed layout can be generated efficiently with a multi-level approach. For our implementation, we combined some of their methods: To create the next coarser graph in the multi-level hierarchy, edges of a maximal matching are contracted [26]. This works very well except for graphs that contain many star-like structures. (Then, only few edges are removed in the next coarser graph.) For such graphs, we therefore use an inclusion-maximal independent set as in [8].

Instead of forces according to Fruchterman and Reingold [7], we followed the approach of Kamada and Kawai [17], because these forces are better suited to represent distances in a graph. To avoid a computation and storage of all-pairs shortest paths, we restricted the forces to the nearest 30 nodes. Due to the multi-level embedding, distances are also preserved in a larger range.

Making steps in the direction of the forces is equivalent to minimizing the energy of the springs. Instead of a cooling schedule, we optimize the potential of the springs and determine the step length according to the Wolfe conditions (see, e.g., [23]). They guarantee a sufficient decrease of the potential function and prevent the algorithm to make too short steps without any "fine-tuning" of a cooling schedule. In contrast to the Newton's method, the second derivative is not needed. We remember the last step length to initialize the next step length calculation, which speeds up the algorithm considerably.

4.2 Spectral Layout This method uses eigenvectors of the Laplacian matrix to draw a graph. It has been introduced in [12], but can also be found in text books like, e.g., [9]. Its value for graph drawing has been renewed lately by [19].

The *Laplacian matrix* L of an undirected graph with adjacency matrix $A \in \{0,1\}^{n \times n}$ is defined as $D - A$, where D denotes the diagonal matrix containing the degree of the corresponding nodes on the diagonal. By construction, the Laplacian matrix is symmetric and positive semi-definite and therefore its eigenvalues are real and non-negative. Furthermore, there exists a basis of orthogonal eigenvectors. We will interpret an eigenvector of L as one dimension of coordinates. Thus, two eigenvectors provide a two-dimensional drawing. We are interested in eigenvectors $x \in \mathbb{R}^n$ with small eigenvalues λ minimizing the *energy* $\varepsilon(x) := (x^t L x)/x^t x = \frac{1}{\|x\|^2} \sum_{\{u,v\} \in E} (x_i - x_j)^2 = \lambda$ to place connected nodes close together. It is easy to verify that the all one vector $\mathbb{1} \in \mathbb{R}^n$ is an eigenvector with eigenvalue 0. As this eigenvector is useless for a drawing, orthogonal eigenvectors x and y for the smallest eigenvalues strictly larger than zero are used to create the layout.

The eigenvectors can be computed with power iteration of the matrix $2\Delta I - L$, where Δ denotes the maximum degree of a node in V and $I \in \mathbb{R}^{n \times n}$ is the identity matrix. A vector $x \in \mathbb{R}^n$ is an eigenvector of L with eigenvalue λ iff x is an eigenvector of $2\Delta I - L$ with eigenvalue $2\Delta - \lambda$. For a sparse graph, the matrices L and $2\Delta I - L$ are sparse as well, so the

matrix multiplication can be implemented efficiently using adjacency lists.

We incorporated the edge weights in this approach by replacing the adjacency matrix A by a weighted adjacency matrix $W = (w_{ij})_{ij}$. Since an edge weight $l(v_i, v_j)$ represents the length of an edge, we set $w_{ij} = \frac{1}{l(v_i, v_j))}$, if $\{v_i, v_j\} \in E$, and $w_{ij} = 0$ otherwise. The diagonal matrix D uses the weighted degree in this case. (Using the weighted Laplacian substantially improved our results.)

4.3 High-Dimensional Embedding The key idea of this method [14] is to draw the graph in a very high-dimensional space (usually $d = 50$) and then project the graph to 2D using principal component analysis. For each dimension, a node $s \in V$ is selected. The coordinate of a node $v \in V$ for this dimension is then defined as the distance of v from s, i.e. the length of a shortest s-v-path in G. As our graphs are weighted, the shortest paths were calculated using these edge weights. (The breadth first search in [14] is simply replaced by Dijkstra's algorithm.) Using these distances as coordinates results in a layout in $\mathbb{R}^d$.

In order to project the graph into a plane, principal component analysis (PCA) is used. It determines the linear combination of coordinates that produces the largest variance. First, the coordinates x_i are centered in every dimension $i = 1, \ldots, d$, such that $\sum_{k=1,\ldots,n} x_{ik} = 0$. Then, the (empirical) covariance matrix $S \in \mathbb{R}^d$ is computed with $s_{ij} = \langle x_i, x_j \rangle$. The final 2D coordinates are a linear combination of $x_1, \ldots, x_d$. The eigenvectors u and v of S with the largest and second largest eigenvalues form the weights of the linear combination: $\sum_{i=0,\ldots,d} u_i x_i$ and $\sum_{i=0,\ldots,d} v_i x_i$.

5 Experiments

5.1 Experimental Setup For our experiments, we used graphs of six different types and 10 graphs of every type. Three of the types are real data that stem from an application (Streets, Railway, and AS) while the three other types are randomly generated graphs. Furthermore, three of the types provide already a layout (Streets, Railway, and Planar), which we can use for a comparison. In detail, the six types of graphs are:

Street Networks. The street networks in our test data are derived from street maps of US cities and their surroundings. As bends are realized with piecewise straight lines, these graphs are very sparse and fairly large. Unfortunately, our data does not contain information about the street classification, so all edges are weighted by slightly dis-

	Street		Rail		AS	
1	1429	3034	409	1215	14492	29948
2	2948	7128	698	1669	14689	30336
3	15868	33380	1650	4311	14899	30712
4	20036	41476	2239	5948	15078	31347
5	24106	53826	2348	7856	15268	33743
6	35802	78646	4553	14829	15415	34716
7	38823	79988	6848	18542	15578	35041
8	44439	96994	10795	29328	15763	33000
9	44878	90930	12070	33728	15899	33585
10	78947	171410	14335	39887	16037	34283

Table 1: Number of nodes and edges for real-world graphs.

	Planar		Small World		Preferential	
1	2000	10000	2000	11974	2000	12000
2	4000	20000	4000	23984	4000	24000
3	5999	30000	6000	35960	6000	36000
4	7999	40000	8000	47974	8000	48000
5	9998	50000	10000	59978	10000	60000
6	11999	60000	12000	71974	12000	72000
7	13998	70000	14000	83976	14000	84000
8	15999	80000	16000	95974	16000	96000
9	17998	90000	18000	107984	18000	108000
10	19999	100000	20000	119970	20000	120000

Table 2: Number of nodes and edges for generated graphs.

turbed Euclidean lengths.

Railway Graphs. A node in a railway network is a station or stop. There exists an edge between two nodes, if there is a non-stop connection serving the respective stations. The edges are weighted according to their average travel time. A layout of this graph is provided by the geographic coordinates of the stations (although the edge weights are not derived from this layout).

Autonomous Systems. These graphs represent the autonomous systems topology of the Internet. An Autonomous System (AS) is a collection of routers which are under one administrative domain (e.g. UUNET, AT&T, or DFN). They present a unified face to the rest of the world in terms of accessibility and routing policies. Every node in an AS-graph represents one Autonomous System, and two nodes are connected if there is at least one physical link between the two corresponding Autonomous Systems.

The AS-graphs are generated regularly by the *Oregon Route Views Project* using traces. If $t(e)$

denotes the number of traces that crossed an edge e, we regard $\frac{1}{t(e)}$ as the edge length. Therefore, good connections that are passed by a lot of traces are regarded as short edges.

Random Planar Graphs. We generated a family of random planar graphs. For given n and m, we first select n points uniformly at random in the unit square. Then, we determine a Delaunay triangulation of these points and delete edges randomly until there are only m edges left.[1] The edge weights are the Euclidean distances in the layout that is provided by the construction.

Small Worlds. In [27], a graph model has been introduced to simulate self-organizing networks that show a small average path length (like, e.g., networks of acquaintances, the power network in the Western US, or the collaboration graph of movie actors). Starting from a regular ring lattice with n nodes and d edges per node, an edge is "rewired" with a given probability p. For our test sets we used $d = 3$ and $p = 0.1$. Edge weights are uniformly distributed over $[0, 1]$.

Graphs with Preference. A model for random graphs that produces a power law distribution of the node degrees has been presented in [2]. Nodes are added consecutively to the graph, while an incident edge of a new node is added with a certain probability. To produce the power law distribution of the node degrees, the main idea is that the probability of a new edge to connect to a node is higher, if this node has a high degree. For our test sets, we generated graphs with an expected degree of 3. Edges weights are uniformly distributed over $[0, 1]$.

From all graphs, we used only the maximal connected component for our calculations to assert connectedness. (In all cases, the maximum connected component contained almost all if not all nodes.) The number of nodes and edges are listed in Table 1 for real-world graphs and Table 2 for generated graphs.

The graphs were drawn with the three methods described in Sect. 4 with our own (prototypic) implementation of the algorithms. The running time varied from minutes to hours. However, tuning the graph-drawing algorithms for speed is not the focus of this paper and may be improved for a productive system.

For each graph and layout, we processed shortest-path queries using bidirectional search and all eight combinations of (1) goal-directed search, (2) geometric shortest-path containers, and (3) reach-based routing. We determined the average number of nodes that were inserted in the priority queue as well as the average CPU time used per query. Observe that in contrast to the second number, the first number does not depend on the type of the priority queue, implementation, compiler, system, or processor. The actual *speed-up* for a speed-up technique, graph and layout is then defined as the ratio of the respective values without speed-up technique and with speed-up technique.

The algorithms have been implemented in C++ using LEDA 4.5 (see [22]). In particular, we used the graph and binary heap data structures, vector and matrix classes and the algorithms for computing maximum matchings and Delaunay triangulations from LEDA. The programs were compiled with GCC 3.3 and the experiments were performed on an AMD Opteron with 2.2 GHz running Linux 2.4. For a unified processing, all graphs have been converted to GraphML [3].

5.2 Experimental Results The results are depicted in Fig. 1–14. Each figure summarizes the result for a combination of speed-up techniques that is added to bidirectional search. For each type of graphs, the speed-ups are shown with standard box-and-whisker plots: The box ends at the hinges (very similar to quartiles) and the horizontal line is at the statistical median. Apart from outliers, the so-called "whisker" covers the rest of the values. A data point is considered an outlier if it is more than 1.5 times the length of the box away from the box. The outliers are plotted as small circles. Since there is no layout given for some graphs (AS graphs, small world graphs, and graphs with preferential attachment), the speed-up for "given layout" for these graphs is missing in all figures.

The high-dimensional layout is obviously well suited for goal-directed search (Fig. 1). It is the best layout for small-world and street graphs and also very good for random planar graphs, although the given layout is unbeatable in this case. Note, that for street graphs the speed-up for the high-dimensional layout is higher than for the given layout. However, the overhead for goal-directed search is so large that in many cases no speed-up in terms of CPU time is achieved (Fig. 2).

The situation is completely different for shortest-path containers (Fig. 3 and 6). Except for AS graphs, the speed-up in terms of CPU time is close to that for the number of visited nodes and achieves values up to 20–40. Furthermore, it is interesting to note that for shortest-path containers the generated layouts are as

---

[1]This is very close to the generator of planar graphs in LEDA except that we use a Delaunay triangulation. We think that these graphs correspond more to our intuition of a random planar graph.

good as the given layouts with the high-dimensional layout for AS graphs being the only exception. For AS graphs, spectral layouts result in the highest speed-ups. However, the high variance reflects the problematic convergence of the eigenvectors for AS graphs.

The speed-up for adding reach-based routing (Fig. 3 and 6) is smaller, but it is nice to see that this speed-up technique is well-suited for street graphs as originally intended [11] (but works very well for small word graphs, too). Again, all generated layouts provide a fairly good performance compared to the provided layouts. One would expect that force-directed layouts are the best for goal-directed search and reach-based routing, because they improve if edge length match with the Euclidean distance of the nodes. Surprisingly, this is only the case for railway graphs and graphs with preferential attachment.

If you combine goal-directed search with shortest-path containers (Fig. 5), the speed-up is dominated by the latter speed-up techniques. Therefore, the results are very similar to the results in Fig. 3. The same is true for the combination of shortest-path containers with reach (Fig. 10 and 13) and the combination of all three speed-up techniques (Fig. 11 and 14), which reveal the same characteristics. More interesting are Fig. 9 and 12, which show the combination of goal-directed search and reach. The speed-up of the two techniques add up for most graphs.

For AS graphs, the speed-up concerning CPU time is generally not as good as for the number of visited nodes. This can be explained by the special structure of these graphs, which consists of a highly connected core to which a lot of path-like graphs are attached. The ratio of excluded and visited nodes is very high in this case, which increases the average running time per visited node. For such dense graphs, performing the pruning test almost outweighs its gain.

The speed-up for graphs with preferential attachment is generally very low. This can be attributed to the fact that these graphs are far from being planar (although they are sparse). It is therefore not surprising that it is difficult to find a helpful layout.

6 Conclusion and Outlook

We have seen, that sometimes a fairly good speed-up of the query time is possible by first generating a layout and then applying geometric speed-up techniques. Apart from few exceptions, all three graph-drawing methods produce equally good layouts concerning geometric speed-up techniques. Generating force-directed and spectral layouts relies non-trivially on parameters that are sometimes hard to optimize. Furthermore, a high-dimensional layout is best suited for goal-directed search. We therefore recommend this type of layout.

It is also notable, that in the case a layout is already given for a graph, it is sometimes possible to generate a layout that results in a better speed-up. Obviously, it is not so important for geometric speed-up techniques whether edge lengths correspond to the Euclidean distances, also long-range distances must be preserved to provide good lower bounds for goal-directed search and reach-based routing. For shortest-path containers, the distances are not important at all, but the relative positions of the nodes are the crucial part of the layout.

Motivated by these results, it would be interesting to develop a specialized "graph drawing" method that optimizes the layout for geometric speed-up techniques. Such a layout algorithm would not necessarily produce nice drawings, but generate even better layouts to speed up shortest-path computations.

In [15], geometric speed-up techniques have been compared with other methods that do not use a layout of the graph. An open question is how these results change if recent speed-up techniques like [10, 20, 18] are included.

Acknowledgment

References

[1] R. Ahuja, T. Magnanti, and J. Orlin, *Network Flows*, Prentice–Hall, 1993.

[2] A.-L. Barabàsi and R. Albert, *Emergence of scaling in random networks*, Science, 286 (1999), pp. 357–367.

[3] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. Scott, *GraphML progress report*, in Proceedings of the 9th International Symposium on Graph Drawing (GD '01), P. Mutzel, M. Jünger, and S. Leipert, eds., vol. 2265 of LNCS, Springer, 2001, pp. 501–512.

[4] U. Brandes, F. Schulz, D. Wagner, and T. Willhalm, *Generating node coordinates for shortest-path computations in transportation networks*, ACM Journal on Experimental Algorithmics, 9 (2004), p. R1.

[5] E. W. Dijkstra, *A note on two problems in connexion with graphs*, Numerische Mathematik, 1 (1959), pp. 269–271.

[6] P. Eades, *A heuristic for graph drawing*, Congressus Numerantium, 42 (1984), pp. 149–160.

[7] T. M. Fruchterman and E. M. Reingold, *Graph drawing by force-directed placement*, Software – Practice & Experience, 21 (1991), pp. 1129–1164.

[8] P. Gajer and S. G. Kobourov, *Grip: Graph drawing with intelligent placement*, Journal of Graph Algorithms and Applications, 6 (2002), pp. 203–224.

[9] C. Godsil and G. Royle, *Algebraic Graph Theory*, vol. 207 of Graduate Texts in Mathematics, Springer, 2001.

[10] A. V. Goldberg and C. Harrelson, *Computing the shortest path: a* search meets graph theory*, Tech. Rep. MSR-TR-2004-24, Microsoft Research, 2003. Accepted at SODA 2005.

[11] R. Gutman, *Reach-based routing: A new approach to shortest path algortihms optimized for road networks*, in Proc. Algorithm Engineering and Experiments (ALENEX'04), L. Arge, G. F. Italiano, and R. Sedgewick, eds., SIAM, 2004, pp. 100–111.

[12] K. Hall, *An r-dimensional quadratic placement algorithm*, Management Science, 17 (1970), pp. 219–229.

[13] D. Harel and Y. Koren, *A fast multi-scale method for drawing large graphs*, Journal of graph algorithms and applications, 6 (2002), pp. 179–202.

[14] ———, *Graph drawing by high-dimensional embedding*, in Proceedings of the 10th International Symposium on Graph Drawing (GD '02), vol. 2528 of LNCS, Springer, 2002, pp. 207–219.

[15] M. Holzer, F. Schulz, and T. Willhalm, *Combining speed-up techniques for shortest-path computations*, in Experimental and Efficient Algorithms: Third International Workshop, (WEA 2004), C. C. Ribeiro and S. L. Martins, eds., vol. 3059 of LNCS, Springer, 2004, pp. 269–284.

[16] R. Jacob, M. Marathe, and K. Nagel, *A computational study of routing algorithms for realistic transportation networks*, in Proc. 2nd Workshop on Algorithm Engineering (WAE'98), K. Mehlhorn, ed., 1998, pp. 167–178.

[17] T. Kamada and S. Kawai, *An algorithm for drawing general undirected graphs*, Information Processing Letters, 31 (1989), pp. 7–15.

[18] E. Köhler, R. H. Möhring, and H. Schilling, *Acceleration of shortest path computation*, Tech. Rep. 42-2004, Institute of Mathematics, TU Berlin, 2004.

[19] Y. Koren, *On spectral graph drawing*, in Proceedings of The Ninth International Computing and Combinatorics Conference (COCOON'03), vol. 2697 of LNCS, Springer, 2003, pp. 496–508.

[20] U. 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, M. Raubal, A. Sliwinski, and W. Kuhn, eds., vol. 22 of IfGI prints, Institut für Geoinformatik, Münster, 2004, pp. 219–230.

[21] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Wiley, 1990.

[22] K. Mehlhorn and S. Näher, *LEDA, A platform for Combinatorial and Geometric Computing*, Cambridge University Press, 1999.

[23] J. Nocedal and S. J. Wright, *Numerical Optimization*, Springer Series in Operations Research, Springer, 1999.

[24] F. Schulz, D. Wagner, and K. Weihe, *Dijkstra's algorithm on-line: An empirical case study from public railroad transport*, ACM Journal of Experimental Algorithmics, 5 (2000).

[25] D. Wagner and T. Willhalm, *Geometric speed-up techniques for finding shortest paths in large sparse graphs*, in Proc. 11th European Symposium on Algorithms (ESA 2003), G. D. Battista and U. Zwick, eds., vol. 2832 of LNCS, Springer, 2003, pp. 776–787.

[26] C. Walshaw, *A multilevel algorithm for force-directed graph-drawing*, Journal of Graph Algorithms and Applications, 7 (2003), pp. 253–285.

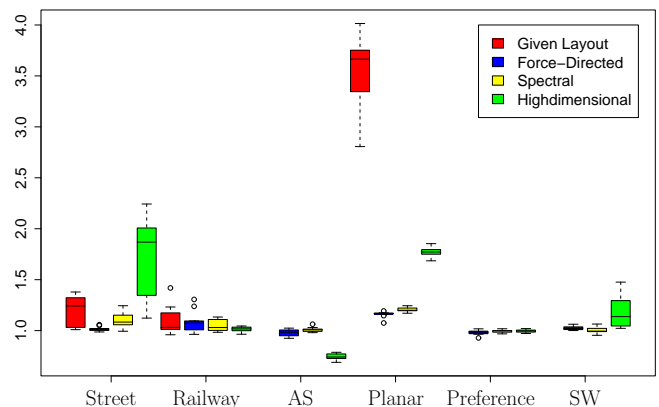[27] D. J. Watts and S. H. Strogatz, *Collective dynamics of 'small-world' networks*, Nature, 393 (1998), pp. 440–442.

Figure 1: Average speed-up in terms of visited nodes using *goal-directed search*
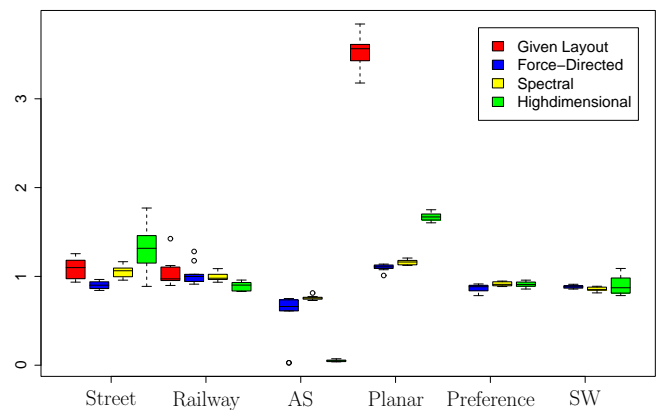

Figure 2: Average speed-up in terms of CPU time using *goal-directed search*
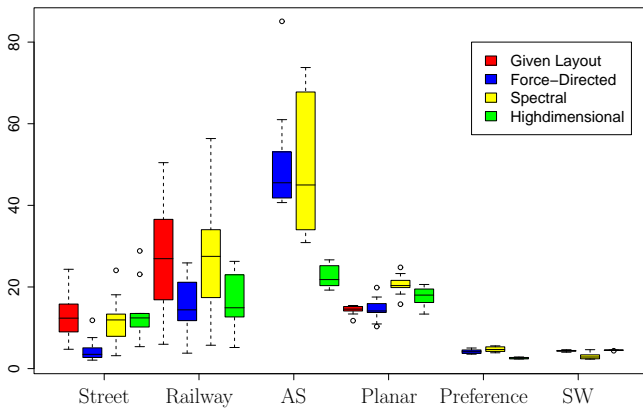
Figure 3: Average speed-up in terms of visited nodes using *shortest-path containers*
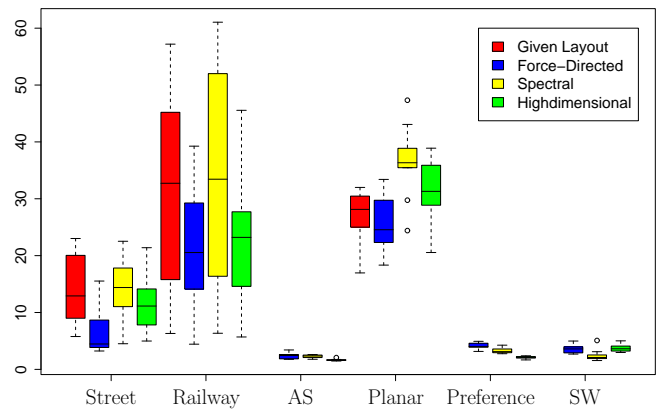

Figure 6: Average speed-up in terms of CPU time using *shortest-path containers*
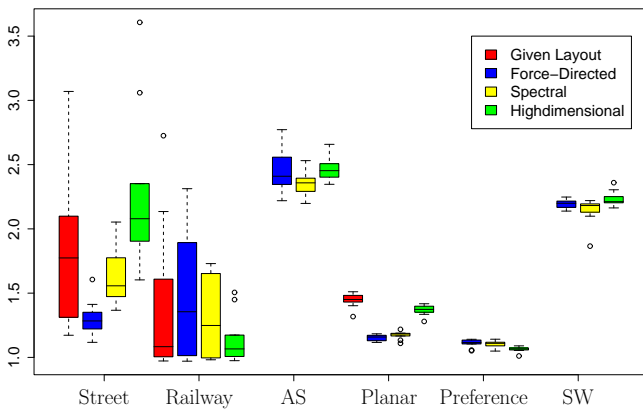

Figure 4: Average speed-up in terms of visited nodes using *reach*
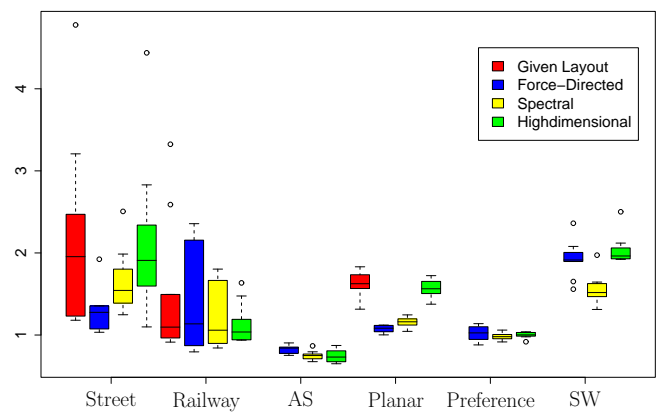

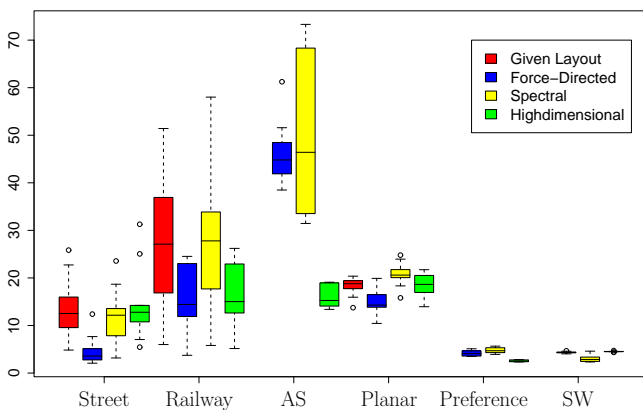Figure 7: Average speed-up in terms of CPU time using *reach*

Figure 5: Average speed-up in terms of visited nodes using *goal-directed search* and *shortest-path containers*
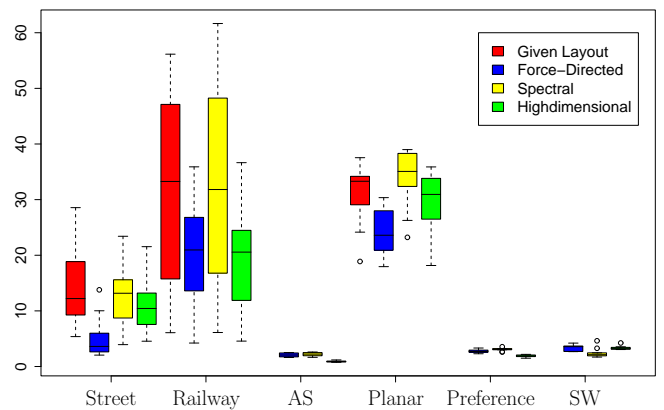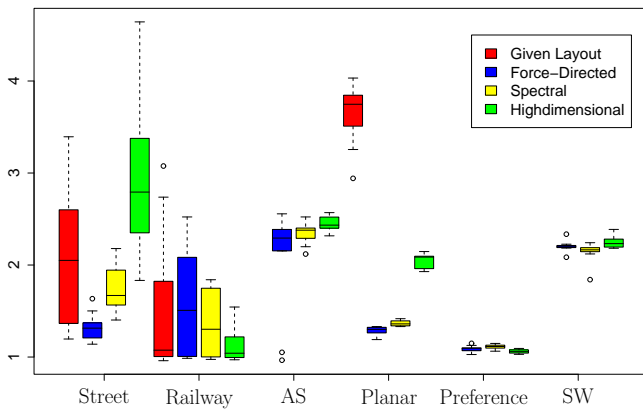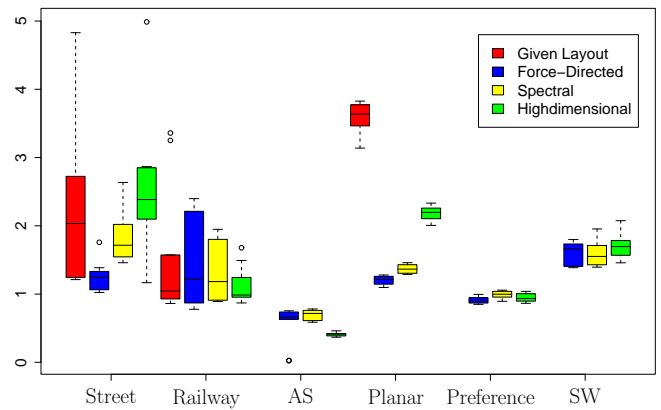

Figure 8: Average speed-up in terms of CPU time using *goal-directed search* and *shortest-path containers*

Figure 9: Average speed-up in terms of visited nodes using *goal-directed search* and *reach*

Figure 12: Average speed-up in terms of CPU time using *goal-directed search* and *reach*
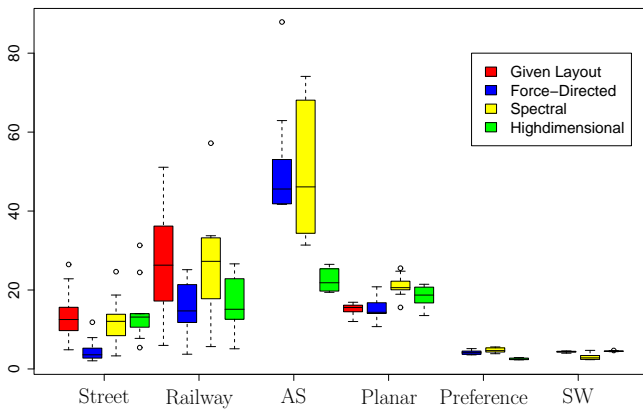

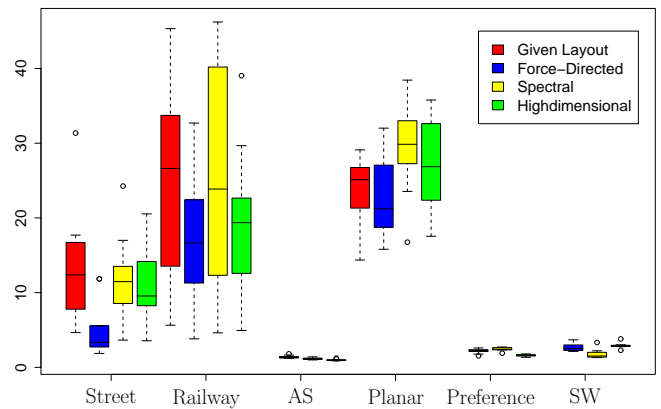Figure 10: Average speed-up in terms of visited nodes using *shortest-path containers* and *reach*

Figure 13: Average speed-up in terms of CPU time using *shortest-path containers* and *reach*
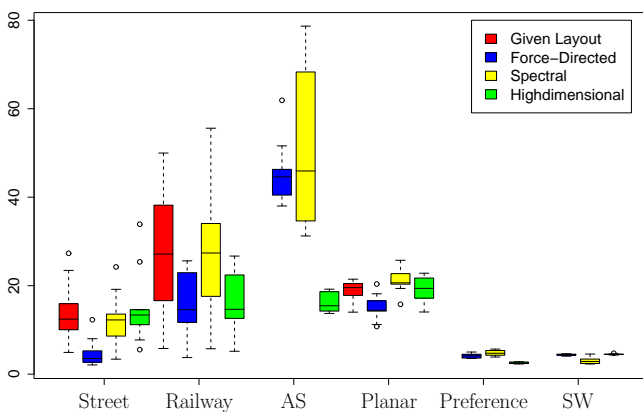

Figure 11: Average speed-up in terms of visited nodes using *goal-directed search*, *shortest-path containers* and *reach*
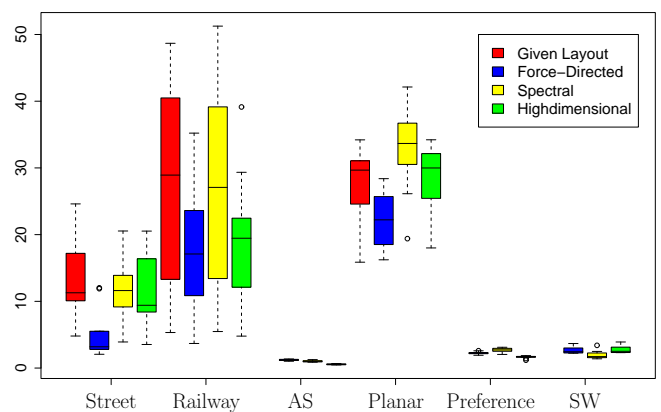

Figure 14: Average speed-up in terms of CPU time using *goal-directed search*, *shortest-path containers* and *reach*