Using FastMap to Solve Graph Problems in a Euclidean Space *

Jiaoyang Li
CS Department
Univ. of Southern California
jiaoyanl@usc.edu

Ariel Felner
SISE Department
Ben-Gurion University
felner@bgu.ac.il

Sven Koenig
CS Department
Univ. of Southern California
skoenig@usc.edu

T. K. Satish Kumar
CS Department
Univ. of Southern California
tkskwork@gmail.com

Abstract

It is well known that many graph problems, like the Traveling Salesman Problem, are easier to solve in a Euclidean space. This motivates the idea of quickly preprocessing a given graph by embedding it in a Euclidean space to solve graph problems efficiently. In this paper, we study a near-linear time algorithm, called FastMap, that embeds a given non-negative edge-weighted undirected graph in a Euclidean space and approximately preserves the pairwise shortest path distances between vertices. The Euclidean space can then be used either for heuristic guidance of A* (as suggested previously) or for geometric interpretations that facilitate the application of techniques from analytical geometry. We present a new variant of FastMap and compare it with the original variant theoretically and empirically. We demonstrate its usefulness for solving a path-finding and a multi-agent meeting problem.

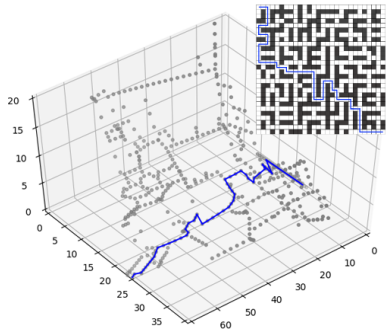


Figure 1: Shows an example of a maze structure interpreted as an undirected graph with unit cost edges and embedded in a 3D Euclidean space using FastMap. The blue line shows the shortest path between a pair of vertices.

1 Introduction

Graphs discussed in this paper are non-negative edge-weighted undirected graphs. Many graph problems on such graphs have variants that are also studied in a Euclidean space. For example, the Traveling Salesman Problem (TSP) can be posed on a graph or in a Euclidean space (i.e., finding a minimum cost cycle that goes through each vertex on the graph or each point in the Euclidean space exactly once). The cost between two vertices on the graph is the graph distance (= the length of the shortest path) between them, while the cost between two points in the Euclidean space is the Euclidean distance (= the length of the line segment) between them. Both variants of the TSP are NP-hard to solve optimally. However, the TSP on a graph is NP-hard to approximate within any polynomial factor (Cormen 2009), while the TSP in a Euclidean space has a polynomial-time approximation scheme (Arora 1998).

The properties of a Euclidean space can be leveraged for computational benefits. For example, a Euclidean space is a metric space in which the triangle inequality on distances holds. In addition, in a Euclidean space, geometric objects (like straight lines, angles and bisectors) are well defined. Our ability to conceptualize these objects facilitates human intuition in the design of clever algorithms. These observations motivate the idea of quickly preprocessing a given graph and embedding it in a Euclidean space to solve graph problems efficiently. Although it may not necessarily close the gap between known negative results for graph problems and positive results for their Euclidean variants, it could still be useful for heuristic search or the design of practical approximation algorithms for graph problems.

Therefore, we study a near-linear time algorithm, called FastMap, that embeds a given graph in a Euclidean space and approximately preserves the pairwise the graph distances between vertices, that is, the graph distances between any two vertices are similar to the Euclidean distances between their corresponding points. Figure 1 shows an example of a maze structure interpreted as a graph with unit cost edges and embedded in a 3D Euclidean space using FastMap. The efficiency and effectiveness of FastMap, combined with nearest-neighbor algorithms, such as Locality Sensitive Hashing (Datar et al. 2004), can be used to establish a framework that allows one to quickly switch between the original graph interpretation and the new geometric interpretation of graph problems. We present a new variant

*The research at the University of Southern California was supported by the National Science Foundation (NSF) under grant numbers 1724392, 1409987, 1817189 and 1837779. The research was also supported by the United States-Israel Binational Science Foundation (BSF) under grant number 2017692.

of FastMap and compare it with the original variant theoretically and empirically. We demonstrate its usefulness for solving a path-finding and a multi-agent meeting problem.

2 Background

FastMap (Faloutsos and Lin 1995) was introduced in the Data Mining community for automatically generating Euclidean embeddings of abstract objects. It gets as input a complete graph $G = (V, E)$, where each vertex $v_i \in V$ represents an abstract object O_i, and each edge $(v_i, v_j) \in E$ with weight $D(O_i, O_j)$ represents the distance between objects O_i and O_j. A Euclidean embedding assigns a K-dimensional point $p_i \in \mathbb{R}^K$ to each object O_i. A good Euclidean embedding is one in which the Euclidean distance between any two points p_i and p_j as given by their L_1 or L_2 distance closely approximates $D(O_i, O_j)$. We refer to $\chi_{ij}^{L_1} = \sum_{r=1}^{K} |[p_i]_r - [p_j]_r|$ and $\chi_{ij}^{L_2} = \sqrt{\sum_{r=1}^{K} ([p_i]_r - [p_j]_r)^2}$ as the L_1 and L_2 distances, respectively, between two points $p_i = ([p_i]_1, \ldots, [p_i]_K)$ and $p_j = ([p_j]_1, \ldots, [p_j]_K)$ in a K-dimensional Euclidean space.

FastMap creates a Euclidean embedding by first assuming the existence of a very high dimensional embedding and then carrying out dimensionality reduction to a user-specified number of dimensions. We re-use the description from (Cohen et al. 2018) in the following. In the very first iteration, FastMap heuristically identifies the farthest pair of objects O_a and O_b in linear time. Once O_a and O_b are determined, every other object O_i defines a triangle with sides of lengths $d_{ai} = D(O_a, O_i)$, $d_{ab} = D(O_a, O_b)$ and $d_{ib} = D(O_i, O_b)$ (Figure 2(a)). The sides of the triangle define its entire geometry, and the projection of O_i onto O_aO_b is given by

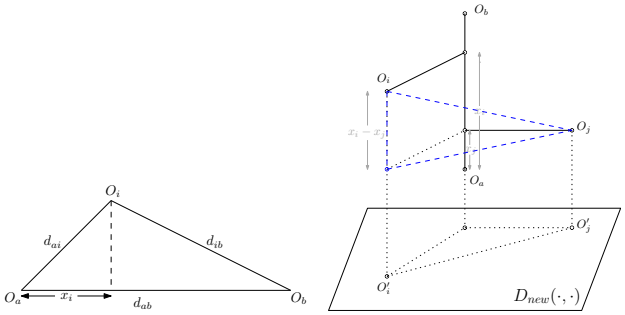
$$x_i = (d_{ai}^2 + d_{ab}^2 - d_{ib}^2)/(2d_{ab}). \quad (1)$$

FastMap sets the first coordinate of p_i, the embedding of O_i, to x_i. In the subsequent $K-1$ iterations, the same procedure is followed for computing the remaining $K-1$ coordinates of each object. However, the distance function is adapted for different iterations. For example, for the first iteration, the coordinates of O_a and O_b are 0 and d_{ab}, respectively. Because these coordinates fully explain the graph distance between these two objects, from the second iteration onward, the rest of p_a and p_b's coordinates should be identical. Intuitively, this means that the second iteration should mimic the first one on a hyperplane that is perpendicular to the line O_aO_b (Figure 2(b)). Although the hyperplane is never constructed explicitly, its conceptualization implies that the distance function for the second iteration should be changed for all i and j in the following way:

$$D_{new}(O_i', O_j')^2 = D(O_i, O_j)^2 - (x_i - x_j)^2. \quad (2)$$

Here, O_i' and O_j' are the projections of O_i and O_j, respectively, onto this hyperplane, and $D_{new}(\cdot, \cdot)$ is the new distance function.

Cohen et al. (2018) extend FastMap to the shortest path-finding problem on graphs. They present an L_1 variant of FastMap that builds a Euclidean embedding in near-linear time, which can then be used to obtain heuristics for A*. They prove that the heuristics are *admissible* and *consistent*. Their admissibility guarantees that A* finds shortest paths, while their consistency allows A* to avoid re-expansions of search nodes as well.



(a) The "cosine law" projection in a triangle.

(b) Projection onto a new hyperplane that is perpendicular to line O_aO_b.

Figure 2: Illustrates how coordinates are computed and recursion is carried out in FastMap, taken from (Cohen et al. 2018).

There are many other approaches to embedding a graph in a Euclidean space, such as (Ng and Zhang 2002; Shavitt and Tankel 2004; Rayner, Bowling, and Sturtevant 2011). However, most of them require solving a Semi-Definite Program or a problem of similar time complexity. They require at least cubic time (Rayner, Bowling, and Sturtevant 2011), and it is therefore prohibitive to apply them to large graphs.

3 Variants of FastMap

In this section, we propose an L_2 variant of FastMap that is more akin to the Data Mining FastMap. We compare it to the L_1 variant theoretically and empirically.

3.1 L_2 Variant of FastMap

Cohen et al. (2018) use L_1 distances primarily to calculate admissible and consistent heuristics, which are helpful for finding optimal solutions efficiently with A*. In many applications, however, suboptimal solutions are sufficient. In such cases, the Data Mining FastMap can be adapted directly to creating an L_2 variant of FastMap. Another motivating factor for the L_2 variant of FastMap is the observation that the straight-line distances of a Euclidean space are most naturally defined as L_2 distances and that it is easier to find good solutions for many problems in computational geometry if L_2 distances are used.

Algorithm 1 shows the L_2 variant of FastMap. K is the maximum number of dimensions allowed in the Euclidean embedding, and ϵ is the threshold that marks a point of diminishing returns when the graph distance between the farthest pair of vertices becomes negligible. d_{ij} represents the graph distance between vertices v_i and v_j. In each iteration, the algorithm identifies the farthest pair (v_a, v_b) of vertices in G heuristically in near-linear time (lines 2-9). Then, it builds two shortest-path trees rooted at vertices v_a and v_b

Algorithm 1: L_2 variant of FastMap.

Input: $G = (V, E)$, K and ϵ.
Output: $p_i \in \mathbb{R}^r$ for all $v_i \in V$.

1 **for** $r = 1, \ldots, K$ **do**
2 Choose $v_a \in V$ randomly and let $v_b = v_a$;
3 **for** $t = 1, \ldots, C$ **do** // C is a constant.
4 $\{d_{ai} | v_i \in V\} \leftarrow$ ShortestPathTree(G, v_a);
5 $v_c \leftarrow \text{argmax}_{v_i}\{d_{ai}^2 - \sum_{j=1}^{r-1}([p_a]_j - [p_i]_j)^2\}$;
6 **if** $v_c == v_b$ **then**
7 Break;
8 **else**
9 $v_b \leftarrow v_a$; $v_a \leftarrow v_c$;
10 $\{d_{ai} | v_i \in V\} \leftarrow$ ShortestPathTree(G, v_a);
11 $\{d_{ib} | v_i \in V\} \leftarrow$ ShortestPathTree(G, v_b);
12 $d'_{ab} \leftarrow d_{ab}^2 - \sum_{j=1}^{r-1}([p_a]_j - [p_b]_j)^2$;
13 **if** $d'_{ab} < \epsilon$ **then**
14 Break;
15 **for** *each* $v_i \in V$ **do**
16 $d'_{ai} \leftarrow d_{ai}^2 - \sum_{j=1}^{r-1}([p_a]_j - [p_i]_j)^2$;
17 $d'_{ib} \leftarrow d_{ib}^2 - \sum_{j=1}^{r-1}([p_i]_j - [p_b]_j)^2$;
18 $[p_i]_r \leftarrow (d'_{ai} + d'_{ab} - d'_{ib})/(2\sqrt{d'_{ab}})$;

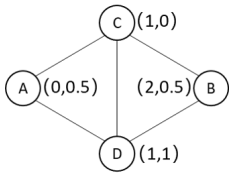


Figure 3: Shows an undirected graph with unit cost edges. The L_2 variant of FastMap embeds it in a 2D Euclidean space with the coordinates shown in the figure. Vertices A and B are picked as pivots during the first iteration, and vertices C and D are picked as pivots during the second iteration. The Euclidean distance between A and C is $\sqrt{5}/2$, which is larger than their graph distance.

to yield all necessary graph distances (lines 10,11). The r^{th} coordinate $[p_i]_r$ of each vertex v_i is computed using Equation (1) (line 18). The pairwise graph distances are updated, when necessary, by the update rule for $D_{new}(\cdot, \cdot)$ in Equation (2) (lines $12, 16, 17$). Although d'_{ab}, d'_{ai} and d'_{ib} computed on these lines can be negative, line 18 never computes the square-root of a negative number because line 13 guarantees it to be positive.

3.2 Comparison of the L_1 and L_2 Variants

Unlike the L_1 variant, the L_2 variant of FastMap does not produce admissible heuristics. This is illustrated in Figure 3. Empirically, the two variants can be compared on the distortion of pairwise graph distances created by their embeddings. We measure distortion as follows: We first pick S vertices sampled uniformly at random. We then compute the graph distances d_{ij} between all possible $S(S-1)/2$ pairs and compare them against $\chi_{ij}^{L_1}$ and $\chi_{ij}^{L_2}$. More precisely, we use the Normalized Root Mean Square Deviation (NRMSD) to standardize the data coming from graphs of different sizes. The NRMSD is given by $\sigma/\bar{d}$ where

$$\sigma = \sqrt{\frac{\sum_{1 \le i < j \le S}(d_{ij} - \chi_{ij})^2}{S(S-1)/2}}$$

$$\bar{d} = \frac{\sum_{1 \le i < j \le S} d_{ij}}{S(S-1)/2}.$$

We use four types of graphs, namely game grids, maze grids, random grids and general (weighted) graphs. Instances of the first three types are from (Sturtevant 2012), and instances of the fourth type are from (Beasley 1990). We use $S = 1{,}000$. Figure 4 shows some results of the comparison, i.e., one graph per type.

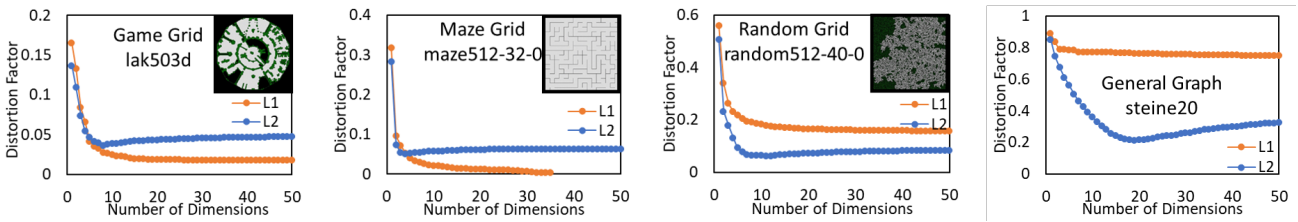
By definition, the Euclidean distances produced by both the L_1 and L_2 variants of FastMap are non-decreasing in the number of dimensions. In particular, the Euclidean distance produced by the L_1 variant is always smaller than the corresponding graph distance and gets closer to it as the number of dimensions increases. Therefore, all distortion curves of the L_1 variant in the four figures are monotonically decreasing. However, the L_2 variant can overestimate the graph distances when the number of dimensions is large. Therefore, all distortion curves of the L_2 variant in the four figures are first decreasing and then increasing.

We observe that, in general, the L_2 variant of FastMap has lower distortion than the L_1 variant when the number of dimensions is small. Intuitively, the L_1 variant is constrained to meet the requirement of admissibility and consistency, while the L_2 variant can focus on accuracy. However, with an increasing number of dimensions, we observe that the comparative performances of the L_1 and L_2 variants change with the nature of the graph. On game grids and maze grids, the L_1 variant is marginally better than the L_2 variant (because the L_2 variant overestimates many graph distances in such cases). On random grids and general graphs, the L_2 variant can be significantly better than the L_1 variant.

4 Applications

In this section, we present two application domains to illustrate the power of FastMap, namely a path-finding and a multi-agent meeting problem.

An optimal solution or any other point of interest in the Euclidean space created by FastMap for a given graph may not correspond to the coordinates of any vertex of that graph. In such a case, we have to assign the point of interest to a vertex in that graph with the closest coordinates. In this paper, we use Locality Sensitive Hashing (LSH) (Datar et al. 2004), specifically a software package from (Andoni et al. 2015), for answering nearest neighbor queries in only $O(\log |V|)$ time. All algorithms were implemented in C++, and all experiments were conducted on a 2.2 GHz Intel Core i5-5200 laptop with 4 GB RAM. On each graph, we used the same 50 random instances for each algorithm.



(a) $|V|$=17,953, $|E|$=33,781. (b) $|V|$=253,840, $|E|$=499,377. (c) $|V|$=104,950, $|E|$=132,954. (d) $|V|$=2,500, $|E|$=62,500.

Figure 4: Compares the distortion of pairwise graph distances created by the L_1 and L_2 variants of FastMap.

4.1 Path-Finding Problem

FastMap can be used to apply techniques from heuristic search as well as analytical geometry to the path-finding problem, where one has to find *any* path from a given start vertex to a given goal vertex on a given graph.

We call our algorithm the path-splitting algorithm. Given an input parameter k, the path-splitting algorithm recursively divides the problem into two subproblems up to recursion depth k. Each recursive step chooses an intermediate vertex and creates two subproblems, namely one between the start vertex and the intermediate vertex and another one between the intermediate vertex and the goal vertex. The intermediate vertex is the Euclidean midpoint chosen with FastMap. At the bottom of the recursion tree, the subproblems can be solved using A* with a choice of different heuristics, including the Manhattan distance heuristics and the FastMap heuristics themselves.

Algorithm 2 shows the pseudo-code of the path-splitting algorithm. There are two possible stop conditions on line 2, namely the depth of the recursion tree being k or v_s and v_g being neighbors in the graph. The algorithm has a choice of using either the L_1 variant or the L_2 variant of FastMap but uses the same variant of FastMap on lines 3, 5 and 9. The path-splitting algorithm with $k = 0$ reduces to A*.

Table 1 shows the empirical performance of the path-splitting algorithm with $k = 0$ and $k = 1$. A* uses the Manhattan distance heuristics. Here, path splitting ($k = 1$) is indeed beneficial since it reduces the number of A* node expansions at the expense of only marginal deviations from the optimal costs. Although the total runtime increases significantly with path splitting, the overhead is associated with the LSH nearest-neighbor queries and not with A*. In other words, improvements to nearest-neighbor algorithms can be used as a black box to directly benefit the path-splitting algorithm.

Figure 5 shows the empirical performance of the path splitting algorithm with higher values of k. Here, too, path splitting is useful with all choices of heuristics, and especially with weak heuristics such as the Manhattan distance heuristics. It is significantly easier to find low-quality solutions than high-quality solutions. In other words, the FastMap framework allows us to "build" paths more efficiently than having to "search" for paths. The power of this framework stems from FastMap being able to summarize the essential information in a graph in a single precomputation step and represent it in a Euclidean space.

Algorithm 2: Path-splitting algorithm.

Input: G, v_s and v_g.
Output: *Path*.

```
1  Function Split(v_s, v_g)
2      if stop condition is met then
3          return A*(v_s, v_g);
4      p_s ←FastMap(v_s); p_g ←FastMap(v_g);
5      p_m ← (p_s + p_g)/2;
6      v_m ← NearestNeighborQuery(p_m);
7      if v_m == v_s || v_m == v_g then
8          return A*(v_s, v_g);
9      Path_1 ← Split(v_s, v_m); Path_2 ← Split(v_m, v_g);
10     return Combine(Path_1, Path_2);
```

Table 1: Shows the empirical performance of the path-splitting algorithm. PS(0) and PS(1) refer to the path-splitting algorithm with $k = 0$ and $k = 1$, respectively.

Graph	FastMap		Cost		Runtime (ms)		A* Nodes	
	Variant	K	PS(0)	PS(1)	PS(0)	PS(1)	PS(0)	PS(1)
game	L_1	10	242	246	1.30	23.87	5,527	3,228
maze	L_1	10	979	1,003	18.06	279.75	83,887	73,086
random	L_2	10	615	653	7.15	115.95	27,088	16,054

4.2 Multi-Agent Meeting Problem

FastMap can be used to apply techniques from analytical geometry to the multi-agent meeting problem, which arises when cooperative agents have to meet at a common goal vertex (Lanthier, Nussbaum, and Wang 2005). We formally define it as a graph problem as follows: Given a graph $G = (V, E)$ and k vertices $s_1, \ldots, s_k \in V$ (the start vertices of the agents), the multi-agent meeting problem is the problem of finding a vertex $v^* \in V$ (the common goal vertex) such that $\sum_{i=1}^{k} D(s_i, v^*) \leq \sum_{i=1}^{k} D(s_i, v)$ for all $v \in V$, where $D(s_i, v)$ is the graph distance between s_i and v.

The multi-agent meeting problem can be solved by building k shortest path trees rooted at $s_1, \ldots, s_k$. The runtime of this algorithm is k times that of Dijkstra's algorithm (Dijkstra 1959; Fredman and Tarjan 1987), i.e., its time complexity is $O(k(|E| + |V| \log |V|))$. In a Euclidean space, however, the multi-agent meeting problem is referred to as the Fermat-Weber problem (Durier and Michelot 1985), which is known to be NP-hard to solve optimally for L_2 distances but solvable in linear time for L_1 distances with the Quickselect algorithm (Hoare 1961).

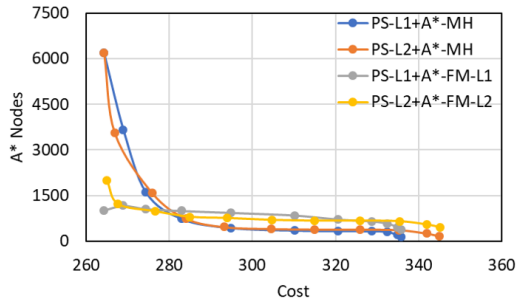


Figure 5: Shows the empirical performance of the path-splitting algorithm on the game grid. The number of dimensions K is 8. Each dot on a curve represents a different depth k of the recursion tree, from 0 (left) to 10 (right). PS-L1+A*-MH indicates that the L_1 variant of FastMap and A* with Manhattan distance heuristics are used. The other names have similar interpretations.

Table 2: Shows the empirical performance of the L_1 variant of FastMap with $K = 5$ on instances of the multi-agent meeting problem. PrepTime refers to the preprocessing time of FastMap, and FastMap Runtime refers to the time needed for solving the Fermat-Weber problem plus answering the nearest-neighbor query. All times are reported in ms.

	Suboptimality (%)			Prep-	FastMap Runtime			Dijkstra Runtime		
Graph	k=10	k=100	k=1000	Time	k=10	k=100	k=1000	k=10	k=100	k=1000
game	3.00	1.32	0.98	795	22	22	27	22	187	1,841
maze	6.76	3.95	1.17	10,930	268	268	274	362	3,535	35,450
random	5.96	2.99	2.69	5,441	117	118	124	181	1,744	17,633
general	34.80	16.79	15.57	551	4	4	7	10	90	810

The tractability of the Fermat-Weber problem for L_1 distances inspires us to use the L_1 variant of FastMap for preprocessing the given graph of the multi-agent meeting problem. It first embeds the graph in a Euclidean space using the L_1 variant of FastMap. It then solves the Fermat-Weber problem for L_1 distances in polynomial time. Finally, it uses LSH to map the optimal Euclidean solution back to a vertex in the graph.

Table 2 shows the resulting empirical performance. Generally speaking, FastMap produces close-to-optimal solutions. For game grids, maze grids and random grids, the FastMap solutions are very close to optimal. On general graphs, their suboptimality is at most about 35%. FastMap has a huge advantage with respect to runtime. First, FastMap demonstrates the power of precomputation on general graphs, since it is generally two orders of magnitude faster than Dijkstra's algorithm in terms of runtime. In fact, its time complexity is only $\tilde{O}(k + \log |V|)$ compared to $O(k(|E| + |V| \log |V|))$ for Dijkstra's algorithm. Second, FastMap also demonstrates the power of Euclidean embeddings, since its precomputation time plus its runtime is also less than the runtime of Dijkstra's algorithm when the number of agents is large. The table for the L_2 variant of FastMap is very similar to Table 2 and is therefore not presented here.

5 Conclusions

In this paper, we presented a new variant of FastMap and compared it with the original variant theoretically and empirically. We presented two application domains to illustrate the power of FastMap, namely the path-finding problem and the multi-agent meeting problem. Overall, the FastMap framework combined with Locality Sensitive Hashing allowed us to solve graph problems using the more efficient algorithms designed for solving their geometric counterparts in a Euclidean space.

References

Andoni, A.; Indyk, P.; Laarhoven, T.; Razenshteyn, I.; and Schmidt, L. 2015. Practical and optimal LSH for angular distance. In *NIPS*, 1225–1233.

Arora, S. 1998. Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. *Journal of the ACM* 45(5):753–782.

Beasley, J. E. 1990. OR-library: distributing test problems by electronic mail. *Journal of the Operational Research Society* 41(11):1069–1072.

Cohen, L.; Uras, T.; Jahangiri, S.; Arunasalam, A.; Koenig, S.; and Kumar, T. K. S. 2018. The FastMap algorithm for shortest path computations. In *IJCAI*, 1427–1433.

Cormen, T. H. 2009. *Introduction to algorithms*. MIT press.

Datar, M.; Immorlica, N.; Indyk, P.; and Mirrokni, V. S. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*, 253–262.

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

Durier, R., and Michelot, C. 1985. Geometrical properties of the Fermat-Weber problem. *European Journal of Operational Research* 20(3):332–343.

Faloutsos, C., and Lin, K.-I. 1995. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *SIGMOD*, 163–174.

Fredman, M., and Tarjan, R. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM* 34(3):596–615.

Hoare, C. A. 1961. Algorithm 65: Find. *Communications of the ACM* 4(7):321–322.

Lanthier, M. A.; Nussbaum, D.; and Wang, T. 2005. Calculating the meeting point of scattered robots on weighted terrain surfaces. In *Australasian Symposium on Theory of Computing-Volume 41*, 107–118.

Ng, T. S. E., and Zhang, H. 2002. Predicting internet network distance with coordinates-based approaches. In *INFOCOM*, 170–179.

Rayner, C.; Bowling, M.; and Sturtevant, N. 2011. Euclidean heuristic optimization. In *AAAI*, 81–86.

Shavitt, Y., and Tankel, T. 2004. Big-bang simulation for embedding network distances in Euclidean space. *IEEE/ACM Transactions on Networking* 12(6):993–1006.

Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games* 4(2):144–148.