The FastMap Algorithm for Shortest Path Computations

Liron Cohen$^1$, Tansel Uras$^1$, Shiva Jahangiri$^2$, Aliyah Arunasalam$^1$, Sven Koenig$^1$ and T. K. Satish Kumar$^1$

$^1$University of Southern California
$^2$University of California, Irvine
{lironcoh, turas, arunasal, skoenig}@usc.edu, shivaj@uci.edu, tkskwork@gmail.com

Abstract

We present a new preprocessing algorithm for embedding the nodes of a given edge-weighted undirected graph into a Euclidean space. The Euclidean distance between any two nodes in this space approximates the length of the shortest path between them in the given graph. Later, at runtime, a shortest path between any two nodes can be computed with an A* search using the Euclidean distances as heuristic. Our preprocessing algorithm, called FastMap, is inspired by the data-mining algorithm of the same name and runs in near-linear time. Hence, FastMap is orders of magnitude faster than competing approaches that produce a Euclidean embedding using Semidefinite Programming. FastMap also produces admissible and consistent heuristics and therefore guarantees the generation of shortest paths. Moreover, FastMap applies to general undirected graphs for which many traditional heuristics, such as the Manhattan Distance heuristic, are not well defined. Empirically, we demonstrate that A* search using the FastMap heuristic is competitive with A* search using other state-of-the-art heuristics, such as the Differential Distance heuristic.

1 Introduction and Related Work

Shortest path computations commonly occur in the inner procedures of many AI programs. In video games, for example, a large fraction of CPU cycles is spent on shortest path computations [Uras and Koenig, 2015]. Many other tasks in AI, including motion planning, temporal reasoning, and decision making [Russell and Norvig, 2009], also involve finding and reasoning about shortest paths. While Dijkstra’s algorithm [Dijkstra, 1959] can be used to compute shortest paths in polynomial time, speeding up shortest path computations allows one to solve the aforementioned tasks faster. One way to do that is to use A* search with an informed heuristic [Hart et al., 1968].

A perfect heuristic is one that returns the true distance between any two nodes in a given graph. A* with such a heuristic and proper tie-breaking is guaranteed to expand nodes only on a shortest path between the given start and goal nodes. In general, computing the perfect heuristic between two nodes is as hard as computing the shortest path between them. Hence, A* search benefits from a perfect heuristic only if it is computed offline. However, precomputing all pairwise distances is not only time-intensive but also requires a prohibitive $O(N^2)$ memory where $N$ is the number of nodes. The memory requirements for storing all-pairs shortest paths data can be somewhat addressed through compression [Botea and Harabor, 2013; Strasser et al., 2015].

Existing methods for preprocessing a given graph (without precomputing all pairwise distances) can be grouped into the following categories: Hierarchical abstractions that yield suboptimal paths have been used to reduce the size of the search space by abstracting groups of nodes [Botea et al., 2004; Sturtevant and Buro, 2005]. More informed heuristics [Björnsson and Halldórsson, 2006; Cazenave, 2006; Sturtevant et al., 2009] focus A* searches better, resulting in fewer expanded nodes. Hierarchies can also be used to derive heuristics during the search [Leighton et al., 2008; Holte et al., 1994]. Dead-end detection and other pruning methods [Björnsson and Halldórsson, 2006; Goldenberg et al., 2010; Pochter et al., 2010] identify areas of the graph that do not need to be searched to find shortest paths. Search with contraction hierarchies [Geisberger et al., 2008] is an optimal hierarchical method, where every level of the hierarchy contains only a single node. It has been shown to be efficient on road networks but seems to be less efficient on graphs with higher branching factors, such as grid-based game maps [Storandt, 2013]. N-level graphs [Uras and Koenig, 2014], constructed from undirected graphs by partitioning the nodes into levels, also allow for significant pruning during the search.

A different approach that does not rely on preprocessing of the graph is to use some notion of a geometric distance between two nodes as a heuristic of the distance between them. One such heuristic for gridworlds is the Manhattan Distance heuristic.$^1$ For many gridworlds, A* search using

$^1$In a 4-neighbor 2D gridworld, for example, the Manhattan Distance between two cells $(x_1, y_1)$ and $(x_2, y_2)$ is $|x_1 - x_2| + |y_1 - y_2|$. Generalizations exist for 8-neighbor 2D and 3D gridworlds.
the Manhattan Distance heuristic outperforms Dijkstra’s algorithm. However, in complicated 2D/3D gridworlds like mazes, the Manhattan Distance heuristic may not be sufficiently informed to focus A* searches effectively. Another issue associated with Manhattan Distance-like heuristics is that they are not well defined for general graphs. For a graph that cannot be conceived in a geometric space, there is no closed-form formula for a “geometric” heuristic for the distance between two nodes because there are no coordinates associated with them.

For a graph that does not already have a geometric embedding in Euclidean space, a preprocessing algorithm can be used to generate one. As described before, at runtime, A* search would then use the Euclidean distance between any two nodes in this space as an estimate for the distance between them in the given graph. One such approach is Euclidean Heuristic Optimization (EHO) [Rayner et al., 2011]. EHO guarantees admissibility and consistency and therefore generates shortest paths. However, it requires solving a Semidefinite Program (SDP). SDPs can be solved in polynomial time [Vandenberghe and Boyd, 1996]. EHO leverages additional structure to solve them in cubic time. Still, a cubic preprocessing time limits the size of the graphs that are amenable to this approach.

The Differential heuristic is another state-of-the-art approach that has the benefit of a near-linear runtime. However, unlike the approach in [Rayner et al., 2011], it does not produce an explicit Euclidean embedding. In the preprocessing phase of the Differential heuristic approach, some nodes of the graph are chosen as pivot nodes. The distances between each pivot node and every other node are precomputed and stored [Sturtevant et al., 2009]. At runtime, the heuristic between two nodes a and b is given by \( \max_p |d(a, p) - d(p, b)| \), where \( p \) is a pivot node and \( d(\cdot , \cdot) \) is the precomputed distance. The preprocessing time is linear in the number of pivots times the size of the graph. The required space is linear in the number of pivots times the number of nodes, although a more succinct representation is presented in [Goldenberg et al., 2011]. Similar preprocessing techniques are used in Portal-Based True Distance heuristics [Goldenberg et al., 2010].

In this paper, we present a new preprocessing algorithm, called FastMap, that produces an explicit Euclidean embedding while running in near-linear time. It therefore has the benefits of the small preprocessing time of the Differential heuristic approach and of producing an embedding from which a heuristic between two nodes can be quickly computed using a closed-form formula. Our preprocessing algorithm, dubbed FastMap, is inspired by the data-mining algorithm of the same name [Faloutsos and Lin, 1995]. It is orders of magnitude faster than SDP-based approaches for producing Euclidean embeddings. FastMap also produces admissible and consistent heuristics and therefore guarantees the generation of shortest paths.

The FastMap heuristic has several advantages: First, it is defined for general (undirected) graphs. Second, we observe empirically that, in gridworlds, A* using the FastMap heuristic runs faster than A* using the Manhattan or Octile distance heuristics. A* using the FastMap heuristic runs equally fast or faster than A* using the Differential heuristic, with the same memory resources. The (explicit) Euclidean embedding produced by FastMap also has representational benefits like recovering the underlying manifolds of the graph and/or visualizing them. Moreover, we observe that the FastMap and Differential heuristics have complementary strengths and can be easily combined to generate an even more informed heuristic.

2 The Origin of FastMap

The FastMap algorithm [Faloutsos and Lin, 1995] was introduced in the data-mining community for automatically generating geometric embeddings of abstract objects. For example, if we are given objects in form of long DNA strings, multimedia datasets such as voice excerpts or images or medical datasets such as ECGs or MRIs, there is no geometric space in which these objects can be naturally visualized. However, there is often a well defined distance function between every pair of objects. For example, the edit distance between two DNA strings is well defined although an individual DNA string cannot be conceptualized in geometric space.

Clustering techniques, such as the k-means algorithm, are well studied in machine learning [Alpaydin, 2010] but cannot be applied directly to domains with abstract objects because they assume that objects are described as points in geometric space. FastMap revives their applicability by first creating a Euclidean embedding for the abstract objects that approximately preserves the pairwise distances between them. Such an embedding also helps to visualize the abstract objects, for example, to aid physicians in identifying correlations between symptoms from medical records.

The data-mining FastMap gets as input a complete undirected edge-weighted graph \( G = (V, E) \). Each node \( v_i \in V \) represents an abstract object \( O_i \). Between any two nodes \( v_i \) and \( v_j \), there is an edge \( (v_i, v_j) \in E \) with weight \( D(O_i, O_j) \) that corresponds to the given distance between objects \( O_i \) and \( O_j \). A Euclidean embedding assigns to each object \( O_i \) a K-dimensional point \( p_i \in \mathbb{R}^K \). A good Euclidean embedding is one in which the Euclidean distance between any two points \( p_i \) and \( p_j \) closely approximates \( D(O_i, O_j) \).

One of the early approaches for generating such an embedding is based on the idea of multi-dimensional scaling (MDS) [Torgerson, 1952]. Here, the overall distortion of the pairwise distances is measured in terms of the “energy” stored in “springs” that connect each pair of objects. MDS, however, requires \( O(|V|^2) \) time and hence does not scale well in practice. On the other hand, FastMap [Faloutsos and Lin, 1995] requires only linear time. Both methods embed the objects in a K-dimensional space for a user-specified K.

FastMap works as follows: In the very first iteration, it heuristically identifies the farthest pair of objects \( O_a \) and \( O_b \) in linear time. It does this by initially choosing a random object \( O_b \) and then choosing \( O_a \) to be the farthest object

\[ \text{The edit distance between two strings is the minimum number of insertions, deletions or substitutions that are needed to transform one to the other.} \]
Figure 1: (a) The three sides of a triangle define its entire geometry. In particular, $x_i = (d_{ai}^2 + d_{ib}^2 - d_{ab}^2)/(2d_{ab})$. (b) Shows a geometric conceptualization of the recursive step in FastMap. In particular, $D_{new}(O_i', O_j')^2 = D(O_i, O_j)^2 - (x_i - x_j)^2$.

away from $O_b$. It then reassigns $O_b$ to be the farthest object away from $O_a$. Once $O_a$ and $O_b$ are determined, every other object $O_i$ defines a triangle with sides of lengths $d_{ai} = D(O_a, O_i)$, $d_{ib} = D(O_a, O_b)$ and $d_{ib} = D(O_i, O_b)$. Figure 1(a) shows this triangle. The sides of the triangle define its entire geometry, and the projection of $O_i$ onto $O_aO_b$ is given by $x_i = (d_{ai}^2 + d_{ib}^2 - d_{ab}^2)/(2d_{ab})$. FastMap sets the first coordinate of $p_i$, the embedding of object $O_i$, to $x_i$. In particular, the first coordinate of $p_a$ is $x_a = 0$ and of $p_b$ is $x_b = d_{ab}$. Computing the first coordinates of all objects takes only linear time since the distance between any two objects $O_i$ and $O_j$ for $i, j \notin \{a, b\}$ is never computed.

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 true distance between them, from the second iteration onwards, 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 1(b) explains this intuition. Although the hyperplane is never constructed explicitly, its conceptualization implies that the distance function for the second iteration should be changed in the following way: $D_{new}(O_i', O_j')^2 = D(O_i, O_j)^2 - (x_i - x_j)^2$. Here, $O_i'$ and $O_j'$ are the projections of $O_i$ and $O_j$, respectively, onto this hyperplane, and $D_{new}$ is the new distance function.

3 FastMap for Shortest Path Computations

In this section, we provide the high-level ideas for how to adapt the data-mining FastMap algorithm to shortest path computations. In the shortest path computation problem, we are given a non-negative edge-weighted undirected graph $G = (V, E)$ along with a start node $v_s$ and a goal node $v_g$. As a preprocessing technique, we can embed the nodes of $G$ in a Euclidean space. As $A^*$ searches for a shortest path from $v_s$ to $v_g$, it can use the Euclidean distance from $v \in V$ to $v_g$ as a heuristic for $v$. The number of node expansions of $A^*$ search depends on the informedness of the heuristic which, in turn, depends on the ability of the embedding to preserve the pairwise distances.

The idea is to view the nodes of $G$ as the objects to be embedded in Euclidean space. As such, the data-mining FastMap algorithm cannot directly be used for generating an embedding in linear time. The data-mining FastMap algorithm assumes that the distance $d_{ij}$ between two objects $O_i$ and $O_j$ can be computed in constant time, independent of the number of objects. Computing the distance between two nodes depends on the size of the graph. Another problem is that the Euclidean distances may not satisfy important properties such as admissibility or consistency. Admissibility guarantees that $A^*$ finds shortest paths while consistency allows $A^*$ to avoid re-expansions of nodes as well.

The first issue of having to retain (near-)linear time complexity can be addressed as follows: In each iteration, after we identify the farthest pair of nodes $O_a$ and $O_b$, the distances $d_{ai}$ and $d_{ib}$ need to be computed for all other nodes $O_i$. Computing $d_{ai}$ and $d_{ib}$ for any single node $O_i$ can no longer be done in constant time but requires $O(|E| + |V| \log |V|)$ time instead [Fredman and Tarjan, 1984]. However, since we need to compute these distances for all nodes, computing two shortest path trees rooted at nodes $O_a$ and $O_b$ yields all necessary distances. The complexity of doing so is also $O(|E| + |V| \log |V|)$, which is only linear in the size of the graph. The amortized complexity for computing $d_{ai}$ and $d_{ib}$ for any single node $O_i$ is therefore near-constant time.

The second issue of having to generate a consistent (and thus admissible) heuristic is formally addressed in Theorem 1. The idea is to use $L_1$ distances instead of $L_2$ distances in each iteration of FastMap. The mathematical properties of the $L_1$ distance can be used to prove that admissibility and consistency hold irrespective of the dimensionality of the embedding.

Algorithm 1 presents data-mining FastMap adapted to the shortest path computation problem. The input is an edge-weighted undirected graph $G = (V, E, w)$ along with two user-specified parameters $K_{max}$ and $\epsilon$. $K_{max}$ is the maximum number of dimensions allowed in the Euclidean embedding. It bounds the amount of memory needed to store the Euclidean embedding of any node. $\epsilon$ is the threshold that marks a point of diminishing returns when the distance between the farthest pair of nodes becomes negligible. The output is an embedding $p_i \in \mathbb{R}^K$ (with $K \leq K_{max}$) for each node $v_i \in V$.

The algorithm maintains a working graph $G' = (V, E, w')$ initialized to $G$. The nodes and edges of $G'$ are always identical to those of $G$ but the weights on the edges of $G'$ change with every iteration. In each iteration, the farthest pair $(n_a, n_b)$ of nodes in $G'$ is heuristically identified in near-linear time (line 4). The $K^{th}$ coordinate $[p_i]_K$ of each node $v_i$ is computed using a formula similar to that for $x_i$ in Figure 1(a). However, that formula is modified to $(d_{ai} + d_{ab} - d_{ib})/2$ to ensure admissibility and consistency of the heuristic. In each iteration, the weight of each edge

4unless $|E| = O(|V|)$, in which case the complexity is near-linear in the size of the input because of the log $|V|$ factor
ALGORITHM 1: Shows the FastMap algorithm. \( G = (V, E, w) \) is a non-negative edge-weighted undirected graph; \( K_{max} \) is the user-specified upper bound on the dimensionality; \( \epsilon \) is a user-specified threshold; \( K \leq K_{max} \) is the dimensionality of the computed embedding; \( p_i \) is the Euclidean embedding of node \( v_i \in V \). Line 11 is equivalent to \( w'(u, v) = w(u, v) - \|p_u - p_v\| \).

\[
\begin{align*}
\text{Input:} & \quad G = (V, E, w), K_{max} \text{ and } \epsilon. \\
\text{Output:} & \quad K \text{ and } p_i \in \mathbb{R}^K \text{ for all } v_i \in V.
\end{align*}
\]

1. \( w' = w; \quad K = 1; \)
2. \( \text{while } K_{max} > 0 \) do
3. \( \quad (n_a, n_b) \leftarrow \text{GetFarthestPair}(G'); \)
4. \( \quad \text{Compute shortest path trees rooted at } n_a \text{ and } n_b \) on \( G' \) to obtain \( d_{ab}, d_{a}; \) \text{ and } \( d_{ib}, \) for all \( v_i \in V; \)
5. \( \text{if } d_{ab} < \epsilon \) then
6. \( \quad \text{Break;} \)
7. \( \quad \text{for each } v \in V \) do
8. \( \quad \quad [p_a]_K = (d_{uv} + d_{ab} - d_{ub})/2 \)
9. \( \quad \text{for each edge } (u, v) \in E \) do
10. \( \quad \quad [p_a]_K = (d_{uv} + d_{ab} - d_{ub})/2 \)
11. \( K = K + 1; \quad K_{max} = K_{max} - 1; \)

is decremented to resemble the update rule for \( D_{new} \) in Figure 1(b) (line 11). However, that update rule is modified to \( w'(u, v) = w'(u, v) - [[p_a]_K - [p_a]_K] \) to use the \( L_1 \) distances instead of the \( L_2 \) distances.

The method GetFarthestPair(\( G' \)) (line 4) computes shortest path trees in \( G' \) a small constant number of times, denoted by \( \tau \).\(^5\) It therefore runs in near-linear time. In the first iteration, we assign \( n_a \) to be a random node. A shortest path tree rooted at \( n_a \) is computed to identify the farthest node from it. \( n_b \) is assigned to be this farthest node. In the next iteration, a shortest path tree rooted at \( n_b \) is computed to identify the farthest node from it. \( n_a \) is reassigned to be this farthest node. Subsequent iterations follow the same switching rule for \( n_a \) and \( n_b \). The final assignments of nodes to \( n_a \) and \( n_b \) are returned after \( \tau \) iterations. This entire process of starting from a randomly chosen node can be repeated a small constant number of times.\(^6\)

Figure 2 shows the working of our algorithm on a small gridworld example.

3.1 Proof of Consistency

In this section, we prove the consistency of the FastMap heuristic. Since consistency implies admissibility, this also proves that \( A^* \) with the FastMap heuristic returns shortest paths. We use the following notation in the proofs: \( w_i^1 \) is the weight on the edge between nodes \( x \) and \( y \) in the \( i \)-th iteration; \( d_i^1 \) is the distance between nodes \( x \) and \( y \) in the \( i \)-th iteration (using the weights \( w_i' \); \( p_x \) is the vector of coordinates produced for node \( x \), and \( [p_x]_j \) is its \( j \)-th coordinate;\(^7\)

\( h_i^1 \) is the FastMap heuristic between nodes \( x \) and \( y \) after \( i \) iterations. Note that \( h_i^1 \) is the \( L_1 \) distance between \( p_x \) and \( p_y \) at iteration \( i \), that is \( h_i^1 := \sum_{j=1}^{i} \| p_x[j] - p_y[j] \|. \) We also define \( \Delta_{i+1} := d_i^1 - d_{i+1}^1 \). In the following proofs, we use the fact that \( |A| + |B| \geq |A + B| \) and \( |A| - |B| \leq |A - B| \).

**Lemma 1.** For all \( x, y \) and \( i \), \( d_i^1 \geq 0 \).

**Proof.** We prove by induction that, in any iteration \( i \), \( w_i^1 \geq 0 \) for all \((u, v) \in E\). Thus, the weight of each edge in the \( i \)-th iteration is non-negative and therefore \( d_i^1 \geq 0 \) for all \( u, v \). For the base case, \( w_i^1 = w(u, v) \geq 0 \). We assume that \( w_i^1 \geq 0 \) and show that \( w_{i+1}^1 \geq 0 \). Let \( n_a \) and \( n_b \) be the farthest pair of nodes identified in the \( i \)-th iteration. From lines 9 and 11, \( w_{i+1}^1 = w_i^1 - (d_{uv} - d_{av})/2 + (d_{vb} - d_{ub})/2 \). To show that \( w_{i+1}^1 \geq 0 \) we show that \( w_i^1 \geq |(d_{uv} - d_{av})/2 + (d_{vb} - d_{ub})/2| \). From the triangle inequality, for any node \( u \), \( d_i^1 = \sum_{j=1}^{i} \| p[u[j] - p_i[j]\| \geq \max(d_i^1, d_i^1) \). Therefore, \( d_i^1 \geq |d_{uv} - d_{av}| + |d_{vb} - d_{ub}| \). This means that \( d_i^1 \geq d_i^1 + d_{uv} - d_{av} - d_{vb} - d_{ub} \). Therefore, \( d_i^1 \geq |(d_{uv} - d_{av})/2 + (d_{vb} - d_{ub})/2| \). This concludes the proof since \( w_i^1 \geq d_i^1 \).

\[ \Box \]

**Lemma 2.** For all \( x, y \), and \( i \), \( \Delta_{i+1} \geq |[p_x]_i - [p_y]_i| \).

**Proof.** Let \((u_1 = x, \ldots, u_m = y)\) be the shortest path from \( x \) to \( y \) in iteration \( i \). By definition, \( d_i^1 = \sum_{j=1}^{m} w_{u_j[u_j+1]} \) and \( d_{i+1}^1 = \sum_{j=1}^{m} w_{u_j[u_j+1]} \). From line 11, \( w_{u[u_j[u_j+1] = w_i^1 - \sum_{j=1}^{m} |[p[u_j] - [p[u_j+1]| | \). Therefore, \( \Delta_{i+1} = d_i^1 - d_{i+1}^1 \geq \sum_{j=1}^{m-1} |[p[u_j] - [p[u_j+1]| \). This concludes the proof since \( \sum_{j=1}^{m-1} |[p[u_j] - [p[u_j+1]| \leq |\sum_{j=1}^{m-1} |[p[u_j] - [p[u_j+1]| \leq |\sum_{j=1}^{m-1} |[p[u_j] - [p[u_j+1]| \).

\[ \Box \]

**Lemma 3.** For all \( x, y, g \) and \( i \), \( d_i^1 + h_y^1 - h_x^1 \geq d_{i+1}^1 \).

**Proof.** We prove the lemma by induction on \( i \). The base case for \( i = 1 \) is implied by Lemma 2. We assume that \( d_i^1 + h_y^1 - h_x^1 \geq d_{i+1}^1 \) and show \( d_{i+1}^1 + h_y^1 - h_x^1 \geq d_{i+2}^1 \). We know that \( h_y^1 - h_x^1 = h_y^1 - h_x^1 - |[p[x]_i-[p[y]_i| - |[p[y]_i-[p[y]_i]| \). Since \( |[p[x]_i-[p[y]_i| - |[p[y]_i-[p[y]_i]| \geq |[p[x]_i-[p[y]_i| - |[p[y]_i-[p[y]_i]| \), we have \( h_y^1 - h_x^1 \geq h_y^1 - h_x^1 - |[p[x]_i-[p[y]_i| \). Hence, \( d_{i+1}^1 + h_y^1 - h_x^1 \geq (d_{i+1}^1 + h_y^1 - h_x^1) - |[p[x]_i-[p[y]_i| \). Using the inductive assumption, we get \( d_i^1 + h_y^1 - h_x^1 \geq d_{i+1}^1 - |[p[x]_i-[p[y]_i| \). By definition, \( d_{i+1}^1 = \Delta_{i+2} + d_{i+2} \). Substituting for \( d_{i+1}^1 \), we get \( d_i^1 + h_y^1 - h_x^1 \geq d_{i+2} + \Delta_{i+2} + (\Delta_{i+2} - |[p[x]_i-[p[y]_i| \). Lemma 2 shows that \( \Delta_{i+2} \geq |[p[x]_i-[p[y]_i| \), which concludes the proof.

\[ \Box \]

**Theorem 1.** The FastMap heuristic is consistent.

**Proof.** For all \( x, y, g \) and \( i \): From Lemma 3, we know \( d_i^1 + h_y^1 - h_x^1 \geq d_{i+1}^1 \). From Lemma 1, we know \( d_{i+1}^1 \geq 0 \). Put together, we have \( d_i^1 + h_y^1 \geq h_x^1 \). Finally, \( h_y^1 = \sum_{j=1}^{i} |[p[j]_i - [p[j]_i| = 0 \).

\[ \Box \]
Theorem 2. The informedness of the FastMap heuristic increases monotonically with the number of dimensions.

Proof. This theorem follows from the fact that for any two nodes \( x \) and \( y \), \( h_{xy}^{i+1} = h_{xy}^i + |[p_x]_{i+1} - [p_y]_{i+1}| \geq h_{xy}^i \).

4 Experimental Results

We performed experiments on many benchmark maps from [Sturtevant, 2012]. Figure 3 presents representative results. The FastMap heuristic (FM) and the Differential heuristic (DH) with equal memory resources\(^8\) are compared against each other. In addition, we include the Octile heuristic (OCT) as a baseline, that also uses a closed-form formula for the computation of its heuristic.

We observe that, as the number of dimensions increases, (a) FM and DH perform better than OCT; (b) the median number of expanded nodes when using the FM heuristic decreases (which is consistent with Theorem 2); and (c) the median absolute deviation (MAD) of the number of expanded nodes when using the FM heuristic decreases. When FM’s MADs are high, the variabilities can possibly be exploited in future work using Rapid Randomized Restart strategies.

FastMap also gives us a framework for identifying a point of diminishing returns with increasing dimensionality. This happens when the distance between the farthest pair of nodes stops being “significant”. For example, such a point is observed in Figure 3(f) around dimensionality 5.\(^9\)

In mazes, such as in Figure 3(g), A* using the DH heuristic outperforms A* using the FM heuristic. This leads us to believe that FM provides good heuristic guidance in domains that can be approximated with a low-dimensional manifold. This observation also motivates us to create a hybrid FM+DH heuristic by taking the maximum of the two heuristics. Some relevant results are shown in Table 1. We use FM\((K)\) to denote the FM heuristic with \( K \) dimensions and DM\((K)\) to denote the DH heuristic with \( K \) pivots. For the results in Table 1, all heuristics have equal memory resources. We observe that the number of node expansions of A* using the FM\((5)+DH(5)\) heuristic is always second best compared to A* using the FM\((10)\) heuristic and A* using the DH\((10)\) heuristic. On one hand, this decreases the percentages of instances on which it expands the least number of nodes (as seen in the second row of Table 1). But, on the other hand, its median number of node expansions is not far from that of the best technique in each breakdown.

5 Conclusions

We presented a near-linear time preprocessing algorithm, called FastMap, for producing a Euclidean embedding of a general edge-weighted undirected graph. At runtime, the Euclidean distances were used as heuristic by A* for shortest path computations. We proved that the FastMap heuristic is admissable and consistent, thereby generating shortest paths. FastMap produces the Euclidean embedding in near-linear time, which is significantly faster than competing approaches for producing Euclidean embeddings with optimality guarantees that run in cubic time. We also showed that it is competitive with other state-of-the-art heuristics derived in near-linear preprocessing time. However, FastMap has the combined benefits of requiring only near-linear preprocessing time and producing explicit Euclidean embeddings that try to recover the underlying manifolds of the given graphs.

Acknowledgments

This work was supported by NSF under grant numbers 1724392, 1409987, and 1319966.
Table 1: Shows the median and MAD numbers of A* node expansions for different maps using three different heuristics with equal memory resources on 1000 random instances. FM(10) denotes the FastMap heuristic with 10 dimensions, DH(10) denotes the Differential heuristic with 10 pivots, and FM(5)+DH(5) is a combined heuristic which takes the maximum of a 5-dimensional FastMap heuristic and a 5-pivot Differential heuristic. The results are split into bins according to winners (along with their number of wins).

Figure 3: Shows empirical results on 3 maps from Bioware’s Dragon Age: Origins. (a) is map ‘lak503d’ containing 17,953 nodes and 33,781 edges; (d) is map ‘brc300d’ containing 5,214 nodes and 9,687 edges; and (g) is map ‘maze512-32-0’ containing 253,840 nodes and 499,377 edges. In (b), the x-axis shows the number of dimensions for the FastMap heuristic (or the number of pivots for the Differential heuristic). The y-axis shows the number of instances (out of 1,000) on which each technique expanded the least number of nodes. Each instance has randomly chosen start and goal nodes. (c) shows the median number of expanded nodes across all instances. Vertical error bars indicate the MADs. The figures in the second and third rows follow the same order. In the legends, “FM” denotes the FastMap heuristic, “DH” denotes the Differential heuristic, and “OCT” denotes the Octile heuristic.
References


