Rapidly Computing Approximate Graph Convex Hulls via FastMap

Ang Li[1], Peter Stuckey[2,3], Sven Koenig[1], T. K. Satish Kumar[1]

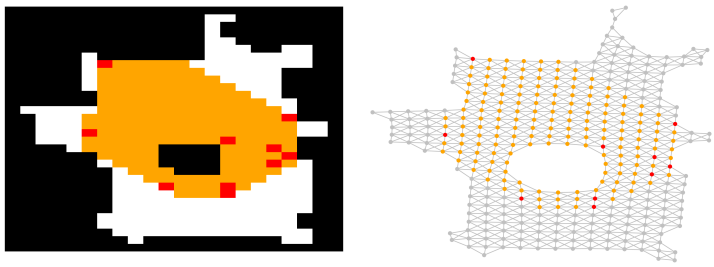
[1]University of Southern California, USA
[2]Monash University, Australia
[3]OPTIMA ARC Industrial Training and Transformation Centre, Australia
ali355@usc.edu, peter.stuckey@monash.edu, skoenig@usc.edu, tkskwork@gmail.com

Abstract. Given an undirected edge-weighted graph G and a subset of vertices S in it, the graph convex hull CH_S^G of S in G is the set of vertices obtained by the process of initializing CH_S^G to S and iteratively adding until convergence all vertices on all shortest paths between all pairs of vertices in CH_S^G of one iteration to constitute CH_S^G of the next iteration. Computing the graph convex hull has applications in shortest-path computations, active learning, and in identifying geodesic cores in social networks, among others. Unfortunately, computing it exactly is prohibitively expensive on large graphs. In this paper, we present a FastMap-based algorithm for efficiently computing approximate graph convex hulls. FastMap is a graph embedding algorithm that embeds a given undirected edge-weighted graph into a Euclidean space in near-linear time such that the pairwise Euclidean distances between vertices approximate the shortest-path distances between them. Using FastMap's ability to facilitate geometric interpretations, our approach invokes the power of well-studied algorithms in Computational Geometry that efficiently compute the convex hull of a set of points in Euclidean space. Through experimental studies, we show that our approach not only is several orders of magnitude faster than the exact brute-force algorithm but also outperforms the state-of-the-art approximation algorithm, both in terms of generality and the quality of the solutions produced.

Keywords: Graph Convex Hull · FastMap · Graph Embedding.

1 Introduction

In Computational Geometry, the convex hull of a finite set of points in Euclidean space is defined as the smallest convex polygon in that space that contains all of them. The problem of computing the convex hull of a given finite set of points is a cornerstone problem with numerous applications: In discrete Geometry, several results rely on convex hulls [13]. In Mathematics, convex hulls are used to study polynomials [12] and matrix eigenvalues [7]. In Statistics, they are used to define risk sets [6]. They also play a key role in polyhedral combinatorics [8].



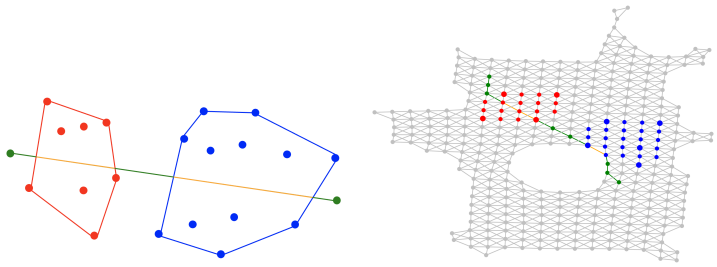
(a) geometric environment (b) graph environment

Fig. 1: Illustrates a graph convex hull. (a) shows a geometric environment with obstacles (black regions) that is discretized as a grid-world. (b) shows a graph representation G of the environment in (a), with vertices representing the top-left corners of the free cells (non-black regions) and edges, weighted by their Euclidean lengths, connecting pairs of vertices on the boundary of the same free cell. In both (a) and (b), the red dots indicate S and the union of the red and orange dots indicates CH_S^G.

While convex hulls have been traditionally studied in geometric spaces, they can also be defined on graphs. In particular, given an undirected edge-weighted graph $G = (V, E, w)$, where V is the set of vertices, E is the set of edges, and for any edge $e \in E$, $w(e)$ is the non-negative weight on it, and a subset of vertices $S \subseteq V$, the *graph convex hull* of S in G is the smallest set of vertices CH_S^G that contains S and all vertices that appear on any shortest path between any pair of vertices in CH_S^G. Procedurally, CH_S^G can be obtained by the process of initializing it to S and iteratively adding until convergence all vertices on all shortest paths between all pairs of vertices in CH_S^G of one iteration to constitute CH_S^G of the next iteration.

Modulo discretization, graphs are capable of representing complex manifolds and geometric spaces. Hence, graph convex hulls "generalize" geometric convex hulls. For example, Figure 1 shows a graph convex hull computed on a graph that represents a 2-dimensional Euclidean space with obstacles. Graph convex hulls have many important properties and applications that are analogous to those of geometric convex hulls. Figure 2 shows one such important analogous property: Every shortest path between any two query vertices on the graph intersects a graph convex hull in a single continuum. Moreover, graph convex hulls have many other applications in active learning [16] and identifying geodesic cores in social networks [14], among others.

While geometric convex hulls can be computed efficiently, little is known about efficient algorithms for computing graph convex hulls. In particular, it is well known that geometric convex hulls can be computed with the following complexities: Given input points S, Qhull [1] can compute the geometric convex hull with corners C in $O(|S| \log |C|)$ time in 2-dimensional and 3-dimensional Euclidean spaces and in $O(|S| f(|C|)/|C|)$ time in higher-dimensional Euclidean



(a) geometric convex hulls (b) graph convex hulls

Fig. 2: Illustrates an important property of graph convex hulls analogous to geometric convex hulls. (a) shows that the straight line (shown in green) joining two points in Euclidean space intersects a geometric convex hull (one shown in red and one shown in blue) in exactly one continuum (shown in orange). Here, the red and blue dots indicate the set of points for which the red and blue geometric convex hulls are computed, respectively. (b) shows that any shortest path (shown in green) between two vertices intersects a graph convex hull (one shown in red and one shown in blue) in exactly one continuum (shown in orange). Here, the larger red and blue dots indicate the set of vertices for which the red and blue graph convex hulls are computed, respectively.

spaces. Here, the function $f(|C|)$ returns the maximum number of faces of a convex polytope with $|C|$ corners and is given by the expression $f(|C|) = O(|C|^{\lfloor \kappa/2 \rfloor}/\lfloor \kappa/2 \rfloor!)$ for a κ-dimensional Euclidean space. Thus, when $\kappa = 2$ or 3, $f(|C|)$ is $O(|C|)$. In contrast, computing graph convex hulls may not be that efficient. In fact, it is folklore that computing graph convex hulls on a general graph $G = (V, E, w)$ takes at least $O(|V||E|)$ time [11]. Hence, brute-force approaches for computing graph convex hulls exactly are prohibitively expensive on large graphs.

In this paper, we present a novel algorithm for efficiently computing approximate graph convex hulls based on FastMap. FastMap [2, 10] is a graph embedding algorithm that embeds a given undirected edge-weighted graph into a Euclidean space in near-linear time such that the pairwise Euclidean distances between vertices approximate the shortest-path distances between them. Since FastMap facilitates geometric interpretations of graph-theoretic problems, our proposed approach utilizes the efficient algorithms mentioned above for computing the geometric convex hull of a set of points in Euclidean space, particularly in 2-dimensional and 3-dimensional Euclidean spaces.

Although our FastMap-based transformation of the graph convex hull problem to the geometric convex hull problem is promising, it does not guarantee exactness. Thus, as a further contribution of this paper, we advance this approach using an iterative refinement procedure. This procedure significantly improves the recall without compromising the precision. Hence, our iterative FastMap-based algorithm has much better Jaccard scores compared to the naive FastMap-

based algorithm. It also runs several orders of magnitude faster than the exact brute-force algorithm. Moreover, we compare our iterative FastMap-based algorithm against the state-of-the-art approximation algorithm and demonstrate two advantages over it. First, our approach is applicable to undirected edge-weighted graphs while the competing approach is applicable only to undirected unweighted graphs. Second, even on undirected unweighted graphs, our iterative FastMap-based algorithm experimentally produces higher-quality solutions and is faster on large graphs.

2 Background: FastMap

FastMap [3] was introduced in the Data Mining community for automatically generating Euclidean embeddings of abstract objects. For many real-world objects such as long DNA strings, multi-media datasets like voice excerpts or images, or medical datasets like ECGs or MRIs, there is no geometric space in which they can be naturally visualized. However, there is often a well-defined distance function between every pair of objects in the problem domain. For example, the edit distance[1] between two DNA strings is well defined although an individual DNA string cannot be conceptualized in geometric space.

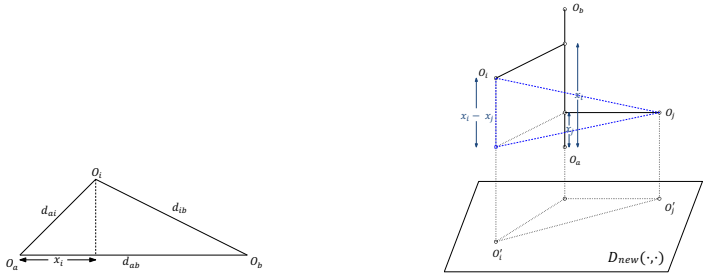
FastMap embeds a collection of abstract objects in an artificially created Euclidean space to enable geometric interpretations, algebraic manipulations, and downstream Machine Learning algorithms. It gets as input a collection of abstract objects $\mathcal{O}$, where $D(O_i, O_j)$ represents the domain-specific distance between objects $O_i, O_j \in \mathcal{O}$. A Euclidean embedding assigns a κ-dimensional point $p_i \in \mathbb{R}^\kappa$ to each object O_i. A good Euclidean embedding is one in which the Euclidean distance χ_{ij} between any two points p_i and p_j closely approximates $D(O_i, O_j)$. For $p_i = ([p_i]_1, [p_i]_2 \dots [p_i]_\kappa)$ and $p_j = ([p_j]_1, [p_j]_2 \dots [p_j]_\kappa)$, $\chi_{ij} = \sqrt{\sum_{r=1}^{\kappa} ([p_j]_r - [p_i]_r)^2}$.

FastMap creates a κ-dimensional Euclidean embedding of the abstract objects in $\mathcal{O}$, for a user-specified value of κ. 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 $d_{ib} = D(O_i, O_b)$, as shown in Figure 3a. The sides of the triangle define its entire geometry, and the projection of O_i onto the line $\overline{O_a O_b}$ is given by

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

FastMap sets the first coordinate of p_i, the embedding of O_i, to x_i. In the subsequent $\kappa - 1$ iterations, the same procedure is followed for computing the remaining $\kappa - 1$ 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

[1] The edit distance between two strings is the minimum number of insertions, deletions, or substitutions that are needed to transform one to the other.



(a) the "cosine law" projection in a triangle

(b) projection onto a hyperplane that is perpendicular to $\overline{O_aO_b}$

Fig. 3: Illustrates how coordinates are computed and recursion is carried out in FastMap, borrowed from [2].

explain the true domain-specific 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 $\overline{O_aO_b}$, as shown in Figure 3b. 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:

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

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.

FastMap can also be used to embed the vertices of a graph in a Euclidean space to preserve the pairwise shortest-path distances between them. The idea is to view the vertices of a given graph $G = (V, E, w)$ as the objects to be embedded. As such, the Data Mining FastMap algorithm cannot be directly used for generating an embedding in linear time. This is because it assumes that the distance d_{ij} between any two objects O_i and O_j can be computed in constant time, independent of the number of objects. However, computing the shortest-path distance between two vertices depends on the size of the graph.

The issue of having to retain (near-)linear time complexity can be addressed as follows: In each iteration, after we heuristically identify the farthest pair of vertices O_a and O_b, the distances d_{ai} and d_{ib} need to be computed for *all* other vertices O_i. Computing d_{ai} and d_{ib} for any single vertex O_i can no longer be done in constant time but requires $O(|E|+|V|\log|V|)$ time instead [4]. However, since we need to compute these distances for all vertices, computing two shortest-path trees rooted at each of the vertices O_a and O_b yields all necessary shortest-path distances in one shot. The complexity of doing so is also $O(|E| + |V|\log|V|)$, which is only linear in the size of the graph.[2]

[2] unless $|E|$ is $O(|V|)$, in which case the complexity is near-linear in the size of the input because of the $\log|V|$ factor

Algorithm 1 FastMap: A near-linear-time graph embedding algorithm.

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

for $r = 1, 2 \dots \kappa$ **do**
  Choose $v_a \in V$ randomly and let $v_b = v_a$.
  for $t = 1, 2 \dots Q$ (a small constant) **do**
    $\{d_{ai} : v_i \in V\} \leftarrow \text{ShortestPathTree}(G, v_a)$.
    $v_c \leftarrow \text{argmax}_{v_i}\{d_{ai}^2 - \sum_{j=1}^{r-1}([p_a]_j - [p_i]_j)^2\}$.
    if $v_c == v_b$ **then**
      Break.
    else
      $v_b \leftarrow v_a$; $v_a \leftarrow v_c$.
    end if
  end for
  $\{d_{ai} : v_i \in V\} \leftarrow \text{ShortestPathTree}(G, v_a)$.
  $\{d_{ib} : v_i \in V\} \leftarrow \text{ShortestPathTree}(G, v_b)$.
  $d'_{ab} \leftarrow d_{ab}^2 - \sum_{j=1}^{r-1}([p_a]_j - [p_b]_j)^2$.
  if $d'_{ab} < \epsilon$ **then**
    $r \leftarrow r - 1$; Break.
  end if
  for each $v_i \in V$ **do**
    $d'_{ai} \leftarrow d_{ai}^2 - \sum_{j=1}^{r-1}([p_a]_j - [p_i]_j)^2$.
    $d'_{ib} \leftarrow d_{ib}^2 - \sum_{j=1}^{r-1}([p_i]_j - [p_b]_j)^2$.
    $[p_i]_r \leftarrow (d'_{ai} + d'_{ab} - d'_{ib})/(2\sqrt{d'_{ab}})$.
  end for
end for
return $p_i \in \mathbb{R}^r$ for all $v_i \in V$.

The foregoing observations are used in [10] to build a graph-based version of FastMap that embeds the vertices of a given undirected graph in a Euclidean space in near-linear time. The Euclidean distances approximate the pairwise shortest-path distances between vertices. Algorithm 1 presents the pseudocode for this algorithm. Here, κ is user-specified, but a threshold parameter ϵ is introduced to detect large values of κ that have diminishing returns on the accuracy of approximating pairwise shortest-path distances.

3 FastMap-Based Algorithms for Graph Convex Hull

In this section, we first introduce a naive version of our FastMap-based algorithm. We then improve it to an iterative version using certain geometric intuitions, which, in turn, are also enabled by FastMap. This iterative version of our FastMap-based algorithm is the final product we use in our experimental comparisons against the state-of-the-art competing approach.

The naive version of our FastMap-based algorithm computes an approximation of the graph convex hull as follows: (1) It embeds the vertices of the given

graph $G = (V, E, w)$ in a Euclidean space with κ dimensions, typically for $\kappa = 2$, 3, or 4; (2) It computes the geometric convex hull of the points corresponding to the vertices in S; and (3) It reports all the vertices that map to the interior[3] of this geometric convex hull as the required approximation of the graph convex hull. This algorithm is very efficient, especially if $\kappa = 2$ or 3: Step (1) runs in $O(|E| + |V| \log |V|)$ time; Step (2) runs in $O(|S| \log |C|)$ time; and Step (3) runs in $O(|V||C|)$ time.[4] In Steps (2) and (3), $|C|$ is upper-bounded by $|S|$ since the computation is done in Euclidean space.

The foregoing algorithm is not guaranteed to return an under-approximation or an over-approximation of the vertices in the graph convex hull. Hence, we introduce the measures of precision, recall, and Jaccard score. Here, the precision refers to the fraction of reported vertices that belong to the ground-truth graph convex hull. The recall refers to the fraction of vertices in the ground-truth graph convex hull that are reported. The Jaccard score refers to the ratio of the number of reported vertices that are in the ground-truth graph convex hull to the number of vertices that are either reported or in the ground-truth graph convex hull. Empirically, we observe that even this naive version of our FastMap-based algorithm generally yields very high precision values on a wide variety of graphs. However, there is a leeway for improving its recall and, consequently, its Jaccard score. Towards this end, we design an iterative version of our FastMap-based algorithm drawing intuitions from the geometric convex hull.

The iterative version of our FastMap-based algorithm computes an approximation of the graph convex hull as follows: In the first iteration, it follows Steps (1) and (2) mentioned above. However, it does not terminate by merely identifying the vertices mapped to the interior of the geometric convex hull. Instead, it identifies the vertices mapped to the corners of the geometric convex hull and computes all shortest paths between all pairs of them directly on the input graph G. Vertices on any of these shortest paths that are not already in S are considered in addition to S for the second iteration of computing the geometric convex hull. The algorithm may not terminate by identifying the vertices mapped to the interior of the new geometric convex hull produced in the second iteration, either. In such a case, it identifies the vertices mapped to the corners of the new geometric convex hull and aims to compute all shortest paths between all pairs of them directly on the input graph G. In doing so, it avoids any redundant computations for pairs of vertices with cached results from the first iteration[5]. The algorithm continues this process until convergence, that is, no new vertices are added to S for the next iteration. Upon convergence, it reports all the vertices mapped to the interior of the geometric convex hull from the last iteration as the required approximation of the graph convex hull. Moreover, the algorithm is guaranteed to converge since: (a) The set of vertices inducted into the graph convex hull after any iteration subsumes that of the previous iteration; and (b)

[3] The interior includes the boundaries and the corners of the geometric convex hull.

[4] To check if a given point is inside a convex polytope, we have to check its relationship to each of the convex polytope's $f(|C|)$ faces. $f(|C|) = O(|C|)$ for $\kappa = 2$ or 3.

[5] a previous iteration, in general

G has a finite number of vertices. Before convergence, the algorithm can also be terminated after a user-specified number of iterations.

Overall, our iterative FastMap-based algorithm hybridizes the exact brute-force algorithm and the naive FastMap-based algorithm. While the exact brute-force algorithm performs all its convex hull-related computations directly on the input graph G, the naive FastMap-based algorithm performs all its convex hull-related computations on the Euclidean embedding of G. The iterative FastMap-based algorithm hybridizes them and performs its convex hull-related computations partly on G and partly on its Euclidean embedding, interleaving them intelligently so that the shortest paths on G are computed only between pairs of vertices mapped to the corners of the geometric convex hull obtained in the previous iteration. On the one hand, it is significantly more efficient than the exact brute-force algorithm that computes all shortest paths between all pairs of vertices in every iteration until convergence. On the other hand, it is more informed than the naive FastMap-based algorithm, which may occasionally miss qualifying vertices—and, consequently, their substantial downstream effects—when they are placed even marginally outside the geometric convex hull in the Euclidean embedding of G.

Figure 4 shows the step-wise working of our iterative FastMap-based algorithm on an example graph. Figure 4a shows the input graph G with the set S indicated in red. Figure 4b shows the geometric convex hull of the red points from Figure 4a in the FastMap embedding of G. The points on the geometric convex hull are indicated in the overriding color blue. Figure 4c shows the interior points of the geometric convex hull from Figure 4b. The blue points are carried over from Figure 4b and the orange points indicate the rest of the interior points. Figure 4d shows the interior (blue and orange) points from Figure 4c identified on G in orange. This set of orange vertices is the algorithm's approximation of CH_S^G after the first iteration.

Figure 4e marks the blue points from Figure 4c as larger green vertices. It also shows all vertices that appear on any of the shortest paths between these larger green vertices as regular green vertices. However, this set of regular green vertices excludes those in S, since we compute only the incremental update to the set of vertices that are deemed to be in CH_S^G. Figure 4f shows the green vertices from Figure 4e and the red vertices from Figure 4a as the vertices deemed to be in CH_S^G at this stage. The points on the geometric convex hull of the points corresponding to these vertices are shown in the overriding color blue. Figure 4g is similar to Figure 4c and carries over the blue points from Figure 4f. However, the new internal points identified in this iteration, compared to the previous one, are shown as larger points. Figure 4h is similar to Figure 4d and is derived from Figure 4g. The new set of orange vertices is the algorithm's approximation of CH_S^G after the second iteration, where the new orange vertices are larger.

Figure 4i marks the blue points from Figure 4g as larger green vertices. It also shows all vertices that appear on any of the shortest paths between these larger green vertices as regular green vertices. However, this set of regular green vertices excludes those in S (red vertices from Figure 4a) and those computed

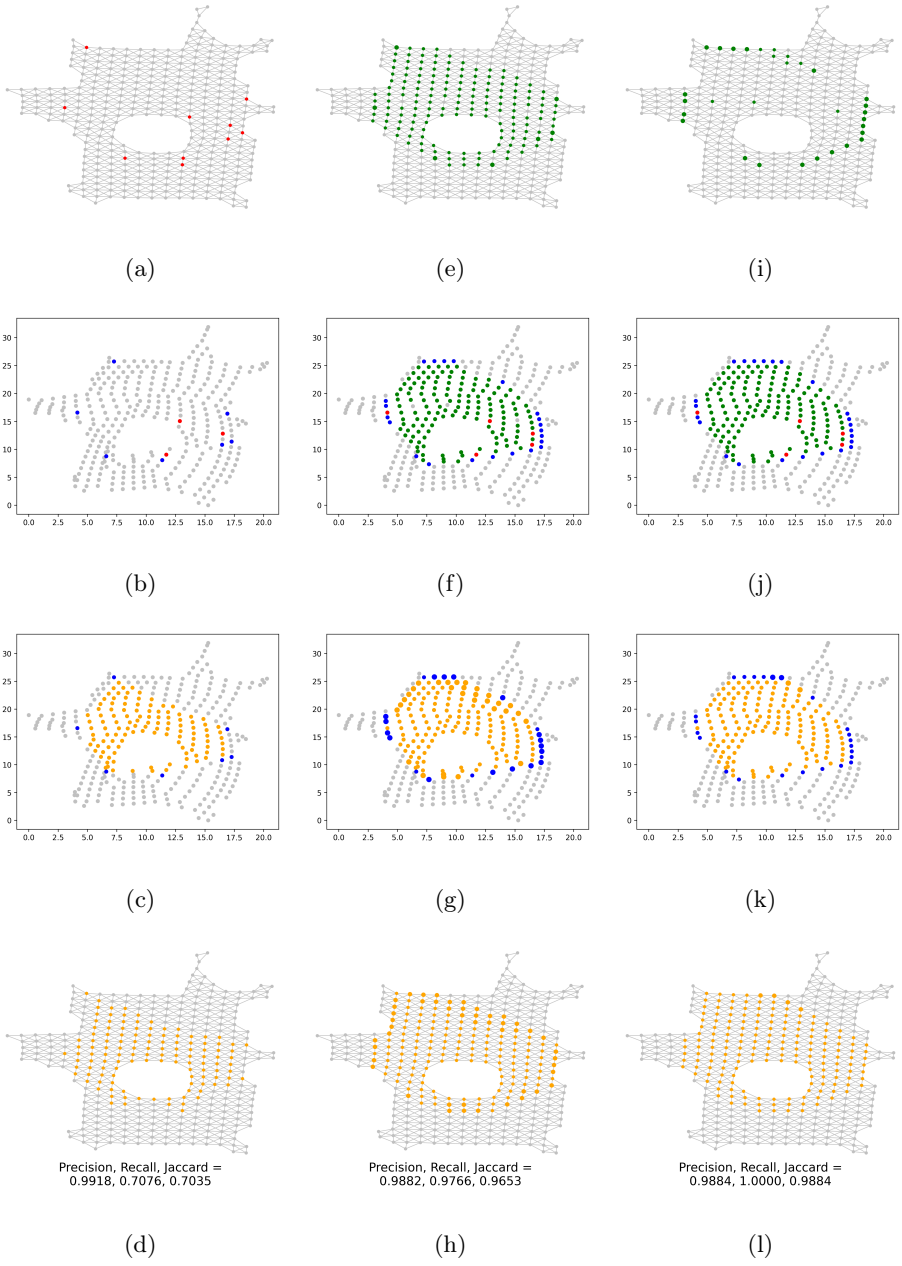


Fig. 4: Shows the behavior of our iterative FastMap-based algorithm on the running example from Figure 1. The individual panels are explained in the main text of the paper. The precision, recall, and Jaccard score are reported after each iteration.

Algorithm 2 FMGCH (FastMap-Based Graph Convex Hull): A FastMap-based algorithm for computing graph convex hulls.

Input: $G = (V, E, w)$ and $S \subseteq V$
Parameter: κ and ϵ
Output: $\overline{CH}_S^G$

$P \leftarrow \text{FastMap}(G, \kappa, \epsilon)$.
$P_S \leftarrow \{p_i \in P : v_i \in S\}$.
$CH_S \leftarrow \text{ConvexHull}(P_S)$.
$CH'_S \leftarrow \{\}$.
$Dicts \leftarrow dictionary()$.
while $CH_S \neq CH'_S$ **do**
  $CH'_S \leftarrow CH_S$.
  $pairs \leftarrow \{(p_i, p_j) : p_i, p_j \in CH_S, i < j,$ and (p_i, p_j) is not cached$\}$.
  $S' \leftarrow S$.
  for $(p_i, p_j) \in pairs$ **do**
    if $v_i \in Dicts$ **then**
      $SPD_{v_i} \leftarrow Dicts[v_i]$.
    else
      $SPD_{v_i} \leftarrow \text{ShortestPathDictionary}(G, v_i)$.
      $Dicts[v_i] \leftarrow SPD_{v_i}$.
    end if
    $S_\Delta \leftarrow \{v \in \text{VerticesOfAllShortestPaths}(SPD_{v_i}, v_j)\}$.
    $S \leftarrow S \cup S_\Delta$.
  end for
  if $S = S'$ **then**
    break
  end if
  $P_S \leftarrow \{p_i \in P : v_i \in S\}$.
  $CH_S \leftarrow \text{ConvexHull}(P_S)$.
end while
$\overline{CH}_S^G \leftarrow \{v_i : p_i \in \text{PointsWithinHull}(CH_S, P)\}$.
return $\overline{CH}_S^G$.

in the previous iterations (green vertices from Figure 4e), since we compute only the incremental update to the set of vertices that are deemed to be in CH_S^G. Figure 4j shows the green vertices from Figures 4i and 4e and the red vertices from Figure 4a as the vertices deemed to be in CH_S^G at this stage. The points on the geometric convex hull of the points corresponding to these vertices are shown in the overriding color blue. Figure 4k is similar to Figure 4g and carries over the blue points from Figure 4j. However, the new internal points identified in this iteration, compared to the previous one, are shown as larger points. Figure 4l is similar to Figure 4h and is derived from Figure 4k. The new set of orange vertices is the algorithm's approximation of CH_S^G after the third iteration, where the new orange vertices are larger. At this stage, the algorithm converges; and, in general, would continue until convergence or a user-specified number of iterations.

Algorithm 2 shows the pseudocode for our iterative FastMap-based algorithm (FMGCH). It takes as input the graph $G = (V, E, w)$ and a set of vertices $S \subseteq V$, for which the graph convex hull needs to be computed. The input parameters κ and ϵ are pertinent to the FastMap embedding, as in Algorithm 1. The output $\overline{CH}_S^G$ is the required graph convex hull or an approximation of it. After the initial value of S is specified via the input, the algorithm maintains and updates it to represent the set of vertices deemed to be in CH_S^G. In addition, it maintains and updates CH_S to represent the geometric convex hull of S.

On Line 1, the algorithm calls Algorithm 1 and creates a κ-dimensional Euclidean embedding of G, with vertex $v_i \in V$ mapped to point $p_i \in \mathbb{R}^\kappa$. On Lines 2 and 3, the algorithm identifies the points corresponding to the initial vertices in S and computes their geometric convex hull. Here, the function ConvexHull$(\cdot)$ returns only the corners of the geometric convex hull. The algorithm then performs some initializations on Lines 4 and 5 and begins the iterative process on Lines 6-25 until convergence is detected on Lines 6 or 20. In each iteration, the old value of CH_S, that is, CH'_S, is first replaced by CH_S on Line 7. Subsequently, CH_S is updated on Lines 8-24. This update starts on Line 8 by identifying the necessary pairs of vertices between which all vertices on all shortest paths need to be computed, while avoiding any redundant computations with respect to the previous iterations. On Lines 11-16, the algorithm then computes the shortest-path dictionary rooted at v_i, for each necessary pair (p_i, p_j) with $i < j$, if this dictionary is not already cached in the '*Dicts*' data structure. The function ShortestPathDictionary$(\cdot, \cdot)$ returns a list of predecessors of each vertex that lead to v_i along a shortest path.[6] On Lines 17 and 18, the algorithm calls the function VerticesOfAllShortestPaths$(\cdot, \cdot)$ to gather all the vertices that appear on any of the shortest paths from v_i to v_j and adds them incrementally to S. On Lines 20-22, the algorithm checks for convergence and breaks the iterative loop if necessary. On Lines 23 and 24, it updates the geometric convex hull in preparation for the next iteration. Upon termination of the iterative loop, on Lines 26 and 27, the algorithm computes and returns the entire interior of the geometric convex hull from the last iteration.

As analyzed before, the running time complexity of the naive FastMap-based algorithm is superior to the folklore results for the exact computation of graph convex hulls. Although it is much harder to analyze the running time complexity of our iterative FastMap-based algorithm, we provide an analysis here under certain realistic assumptions. Let CH_S^G be the ground truth and $\kappa = 2$ or 3.[7] We assume that the algorithm runs for τ iterations, for a small constant τ, has a high precision in all iterations, and that the geometric convex hull CH_S has at most $\bar{C}$ corners in all iterations, with $\bar{C} \ll |CH_S^G|$. These assumptions have been observed to be true in extensive experimental studies. Hence, in each iteration, the algorithm computes the geometric convex hull in $O(|S| \log \bar{C})$ time,

[6] In Python 3.9, this can be realized using the 'dijkstra_predecessor_and_distance()' function of NetworkX [5].

[7] For higher values of κ, the analysis has to explicitly factor in the number of faces of κ-dimensional convex polytopes.

in which $|S|$ is upper-bounded by $|CH_S^G|$. In addition, in each iteration, the algorithm computes all vertices on the shortest paths between all pairs of vertices on CH_S. It does so in two phases. First, it computes the shortest-path dictionaries rooted at each of the vertices of CH_S in $O(\bar{C}(|E| + |V| \log |V|))$ time. Second, it post-processes each such dictionary with respect to each of the destination vertices in CH_S in time linear in the size of the dictionary. This takes $O(\bar{C}^2(|E| + |V|))$ time. Finally, the algorithm computes the interior of the geometric convex hull in $O(|V|\bar{C})$ time. Therefore, the overall time complexity is $O(\tau(\log \bar{C}|CH_S^G| + \bar{C}(|V| \log |V|) + \bar{C}^2(|E| + |V|)))$. This complexity is still better than the folklore results for the exact computation of graph convex hulls. Moreover, the real benefits of the iterative FastMap-based algorithm become evident in the experimental studies presented in the next section.

4 Experimental Results

In this section, we present experimental results that compare FMGCH, that is, Algorithm 2, against the state-of-the-art algorithm for computing graph convex hulls, which is encapsulated within GCoreApproximation (GCA) [14][8]. However, GCA is also an approximation algorithm. Hence, to produce the ground truth (GT), we implemented the brute-force algorithm with as many algorithmic, data-structure, and code-level optimizations as possible.[9] We do not include our naive FastMap-based algorithm in the experiments due to limited space. However, as expected, it is more efficient than FMGCH, our iterative FastMap-based algorithm, but produces a lower recall and Jaccard score. We implemented FMGCH in Python 3.9. For computing the geometric convex hull of a collection of points in Euclidean space, we used the 'Qhull' library [1]. We invoked GCA using a simple Python wrapper function. We conducted all experiments on a laptop with an Apple M2 Max chip and 96 GB RAM.

While FMGCH is applicable to both unweighted and (edge-)weighted graphs, GCA is applicable to only unweighted graphs [14]. That is, FMGCH already has the advantage of being a more general algorithm compared to GCA. Hence, we perform two kinds of experiments. First, we compare FMGCH against GCA on unweighted graphs. Second, we study the performance of FMGCH on weighted graphs. In both cases, the GT procedure is included as the baseline.

We used four datasets in our experiments from which both unweighted and weighted graphs can be derived: the DIMACS, movingAI, SNAP, and the Waxman graphs. The DIMACS graphs[10] are a standard benchmark collection of unweighted graphs. They can be converted to weighted graphs by assigning an integer weight chosen uniformly at random from the interval $[1, 10]$ to each edge. The movingAI graphs model grid-worlds with obstacles [15]. They can be used as unweighted graphs if the grid-worlds are assumed to be four-connected (with

[8] https://github.com/fseiffarth/GCoreApproximation

[9] Describing all of these optimizations is beyond the scope of this paper because of limited space.

[10] https://mat.tepper.cmu.edu/COLOR/instances.html

Instance	Size ($\|V\|$, $\|E\|$)	GT (s)	GCA				FMGCH													
							$\kappa = 2$					$\kappa = 3$					$\kappa = 4$				
			time (s)	jacc	prec	recall	pre (s)	query (s)	jacc	prec	recall	pre (s)	query (s)	jacc	prec	recall	pre (s)	query (s)	jacc	prec	recall
miles1000	(128, 3216)	0.4754	0.0016	0.9922	1.0000	0.9922	0.0091	0.0146	1.0000	1.0000	1.0000	0.0172	0.0394	1.0000	1.0000	1.0000	0.0227	0.0657	1.0000	1.0000	1.0000
myciel7	(191, 2360)	0.5030	0.0009	1.0000	1.0000	1.0000	0.0074	0.0049	1.0000	1.0000	1.0000	0.0099	0.0151	1.0000	1.0000	1.0000	0.0131	0.0300	1.0000	1.0000	1.0000
queen16_16	(256, 6320)	1.1260	0.0034	1.0000	1.0000	1.0000	0.0154	0.0264	1.0000	1.0000	1.0000	0.0346	0.1073	1.0000	1.0000	1.0000	0.0392	0.1241	1.0000	1.0000	1.0000
le450_25d	(450, 17425)	5.5958	0.0240	1.0000	1.0000	1.0000	0.0647	0.0601	0.9133	1.0000	0.9133	0.1187	0.2570	0.9889	1.0000	0.9889	0.0936	0.4376	0.9978	1.0000	0.9978
orz601d	(1890, 3473)	48.2478	0.0307	0.7801	0.8664	0.8867	0.0156	0.0169	0.6456	0.6456	1.0000	0.0278	0.0862	0.7724	0.7731	0.9988	0.0373	0.2785	0.9115	0.9125	0.9988
lak106d	(1909, 3589)	54.9086	0.0431	0.7510	0.8528	0.8628	0.0181	0.0410	0.8722	0.8736	0.9982	0.0319	0.1404	0.9104	0.9111	0.9991	0.0401	0.2190	0.9568	0.9576	0.9991
hrt001d	(3705, 6914)	1342.2423	0.1715	0.6319	0.7551	0.7948	0.0276	0.0714	0.6375	0.6453	0.9815	0.0516	0.2105	0.7344	0.7344	1.0000	0.0595	0.6886	0.9316	0.9354	0.9956
orz000d	(4057, 7744)	1936.5250	0.2134	0.9439	0.9711	0.9711	0.0315	0.0780	0.9683	0.9683	1.0000	0.0944	0.2351	0.9873	0.9884	0.9988	0.0774	0.9722	0.9891	0.9949	0.9941
ca-GrQc	(4158, 13428)	26.1400	0.2220	0.1945	0.3254	0.3259	0.0548	0.1805	0.3307	0.3330	0.9792	0.1027	0.5365	0.3389	0.3403	0.9881	0.0959	1.1852	0.3554	0.3566	0.9903
ca-HepTh	(8638, 24827)	219.1975	1.5639	0.2704	0.4258	0.4255	0.0975	0.2371	0.4219	0.4329	0.9432	0.2595	1.2079	0.4299	0.4313	0.9928	0.2274	3.5873	0.4372	0.4378	0.9967
wiki-Vote	(7066, 100736)	862.9993	2.7505	0.5034	0.6698	0.6696	0.2689	0.4216	0.6173	0.6613	0.9027	0.4657	2.1628	0.6408	0.6550	0.9673	0.5820	5.9597	0.6696	0.6854	0.9666
ca-HepPh	(11204, 117649)	799.8486	3.5258	0.3324	0.4989	0.4990	0.3205	1.1677	0.4352	0.4389	0.9810	0.6013	3.8834	0.4379	0.4404	0.9872	0.6396	13.0594	0.4446	0.4465	0.9907
wm04000	(4000, 20109)	207.3266	1.2004	0.9995	0.9997	0.9997	0.0651	0.1952	0.9945	0.9997	0.9947	0.1505	0.8467	0.9975	1.0000	0.9975	0.1726	2.7267	0.9992	1.0000	0.9992
wm08000	(8000, 40014)	1061.7808	5.3560	0.9998	0.9999	0.9999	0.2337	0.5732	0.9972	0.9999	0.9974	0.3414	2.3460	0.9986	0.9999	0.9987	0.3716	5.6938	0.9987	1.0000	0.9987
wm10000	(10000, 49980)	1755.0927	8.7739	0.9994	0.9997	0.9997	0.2436	0.5254	0.9991	0.9998	0.9993	0.3281	2.0855	0.9983	0.9997	0.9986	0.3831	7.6367	0.9987	0.9997	0.9990
wm12000	(12000, 59853)	2603.6225	12.9154	0.9991	0.9996	0.9995	0.2663	0.9198	0.9929	0.9997	0.9932	0.4050	3.1991	0.9992	0.9997	0.9994	0.4999	9.7890	0.9988	0.9999	0.9989
miles1000	(128, 3216)	0.2592	-	-	-	-	0.0068	0.0056	0.9444	1.0000	0.9444	0.0130	0.0269	0.9683	1.0000	0.9683	0.0174	0.0696	0.9921	1.0000	0.9921
myciel7	(191, 2360)	0.2875	-	-	-	-	0.0075	0.0099	0.8066	0.8957	0.8902	0.0096	0.0282	0.8268	0.9080	0.9024	0.0164	0.1033	0.9266	0.9266	1.0000
queen16_16	(256, 6320)	0.9702	-	-	-	-	0.0127	0.0515	0.9922	1.0000	0.9922	0.0258	0.1423	0.9922	1.0000	0.9922	0.0370	0.2167	0.9961	1.0000	0.9961
le450_25d	(450, 17425)	4.6087	-	-	-	-	0.0580	0.1069	0.9376	1.0000	0.9376	0.0875	0.4325	0.9866	1.0000	0.9866	0.0978	0.6410	0.9889	1.0000	0.9889
orz601d	(1890, 6746)	22.4946	-	-	-	-	0.0255	0.0340	0.5961	0.5961	1.0000	0.0373	0.0930	0.7192	0.7192	1.0000	0.0516	0.3527	0.8906	0.8916	0.9987
lak106d	(1909, 7029)	16.6713	-	-	-	-	0.0268	0.0929	0.7819	0.7819	1.0000	0.0379	0.1915	0.8666	0.8675	0.9988	0.0571	0.7088	0.9330	0.9362	0.9964
hrt001d	(3705, 13498)	318.5500	-	-	-	-	0.0586	0.0769	0.8019	0.8109	0.9864	0.0801	0.4049	0.8297	0.8821	0.9332	0.1104	1.5441	0.9596	0.9629	0.9965
orz000d	(4057, 15208)	439.4339	-	-	-	-	0.0560	0.1739	0.6183	0.6183	1.0000	0.1362	0.7398	0.9270	0.9275	0.9995	0.1106	2.1894	0.9439	0.9439	1.0000
ca-GrQc	(4158, 13428)	27.2310	-	-	-	-	0.0560	0.0823	0.3304	0.3491	0.8604	0.0707	0.5626	0.3424	0.3454	0.9748	0.1090	1.6527	0.3586	0.3605	0.9859
ca-HepTh	(8638, 24827)	219.0672	-	-	-	-	0.0961	0.2631	0.4261	0.4339	0.9599	0.1570	1.0497	0.4288	0.4341	0.9723	0.1943	3.2234	0.4352	0.4379	0.9859
wiki-Vote	(7066, 100736)	867.2150	-	-	-	-	0.2691	0.2840	0.5799	0.6571	0.8316	0.4012	1.8959	0.6610	0.6735	0.9727	0.6367	5.1408	0.6642	0.6849	0.9566
ca-HepPh	(11204, 117649)	805.3969	-	-	-	-	0.3188	0.4995	0.4281	0.4569	0.8717	0.4761	4.9446	0.4385	0.4414	0.9853	0.6435	10.3627	0.4446	0.4466	0.9901
wm04000	(4000, 20109)	139.1687	-	-	-	-	0.1148	0.2276	0.8095	0.9981	0.8107	0.1256	1.2747	0.9815	0.9980	0.9835	0.2004	3.2818	0.9900	0.9980	0.9920
wm08000	(8000, 40014)	628.2476	-	-	-	-	0.1793	0.4662	0.8419	0.9988	0.8427	0.3319	3.6184	0.9895	0.9987	0.9907	0.3983	7.6990	0.9901	0.9989	0.9912
wm10000	(10000, 49980)	1021.6094	-	-	-	-	0.2362	0.9538	0.9012	0.9979	0.9029	0.4013	4.3646	0.9922	0.9979	0.9943	0.5599	11.0225	0.9936	0.9980	0.9956
wm12000	(12000, 59853)	1544.6484	-	-	-	-	0.4505	0.9459	0.8570	0.9986	0.8580	0.5633	6.6372	0.9795	0.9984	0.9811	0.6056	14.4132	0.9935	0.9984	0.9951

Table 1: Shows the performance results of FMGCH, GCA, and the GT procedure. FMGCH is shown with different values of κ. The columns 'prec', 'recall', and 'jacc' indicate the precision, recall, and Jaccard score, respectively. 'GT (s)', 'time (s)' under 'GCA', and 'pre (s)' and 'query (s)' under 'FMGCH' represent the running time of GT, the running time of GCA, the precomputation time of FMGCH, and the query time of FMGCH, respectively, in seconds. The top half of the rows are unweighted graphs and the bottom half are their weighted counterparts. Within each half, the rows are divided into the categories: DIMACS, movingAI, SNAP, and Waxman.

only horizontal and vertical connections) or as weighted graphs if the grid-worlds are assumed to be eight-connected (with additional diagonal connections). The SNAP dataset refers to the Stanford Large Network Dataset Collection [9], some graphs from which were chosen as undirected unweighted graphs for experimentation in [14]. We use the same graphs for a fair comparison with GCA. Moreover, these graphs can be converted to weighted graphs by assigning an integer weight chosen uniformly at random from the interval $[1, 10]$ to each edge. Waxman graphs are used to generate realistic communication networks [17]. Here, we generated Waxman graphs using NetworkX [5] with parameter values $\alpha = 100/|V|$ and $\beta = 0.1$, within a rectangular domain of 100×100, and with the weight on each edge set to the Euclidean distance between its endpoints. These graphs are naturally weighted but can be made unweighted by ignoring the weights.

Table 1 shows the performance results of FMGCH, GCA, and the GT procedure. FMGCH is shown with three subdivisions, for $\kappa = 2$, 3, and 4. In each case, the table reports the precomputation time and the query time. The precomputation time is the time required by the FastMap component of FMGCH to generate the κ-dimensional Euclidean embedding. This precomputation is done only once per graph and can serve the purpose of answering many queries on the same graph with different input S. Due to limited space, only representative results are shown on selected graphs from each dataset. Queries were formulated on a given graph by choosing 10 randomly chosen vertices to constitute the input set S. In most cases, FMGCH required ≤ 10 iterations to converge.

In the first set of experiments (top half of Table 1), it is easy to observe that both FMGCH and GCA are orders of magnitude faster than GT, with the efficiency gaps being more pronounced on large graphs. In fact, FMGCH is also faster than GCA on large graphs. On the DIMACS dataset, both FMGCH and GCA produce very high-quality solutions. On the movingAI dataset, GCA does not produce high-quality solutions. While the same is true for FMGCH with $\kappa = 2$, the quality increases for $\kappa = 3$ and increases further for $\kappa = 4$. In fact, FMGCH with $\kappa = 4$ produces very high-quality solutions. On the SNAP dataset, GCA produces low-quality solutions and FMGCH produces better-quality solutions, particularly on recall. However, this quality deterioration is not related to the larger sizes of the SNAP graphs. In fact, on the Waxman graphs, which are also large, both FMGCH and GCA produce very high-quality solutions.

In the second set of experiments (bottom half of Table 1), GCA is not applicable at all. In contrast, FMGCH is fully applicable and can be evaluated using GT. Even here, it is easy to observe that FMGCH is orders of magnitude faster than GT. The qualities of the solutions that it produces follow the same pattern as in the unweighted case. On some weighted graphs, FMGCH and/or GT may run faster than on their unweighted counterparts. This happens because the number of shortest paths between a pair of vertices is usually more on unweighted graphs and, consequently, computing all of them is more expensive.

5 Conclusions and Future Work

The graph convex hull problem is an important graph-theoretic problem that is analogous to the geometric convex hull problem. The two problems also share many important analogous properties and real-world applications. Yet, while the geometric convex hull problem is very well studied, the graph convex hull problem has not received much attention thus far. Moreover, while the geometric convex hulls can be computed very efficiently in low-dimensional Euclidean spaces, folklore results for algorithms that compute the graph convex hulls exactly make them prohibitively expensive on large graphs. In this paper, we presented a FastMap-based algorithm for efficiently computing approximate graph convex hulls. Our FastMap-based algorithm utilizes FastMap's ability to facilitate geometric interpretations. While the naive version of our algorithm uses a single shot of such a geometric interpretation, the iterative version of our algorithm repeatedly interleaves the graph and geometric interpretations to reinforce one with the other. This iterative version was encapsulated in our solver, FMGCH, and experimentally compared against the state-of-the-art solver, GCA. On a variety of graphs, we showed that FMGCH not only runs several orders of magnitude faster than a highly-optimized exact algorithm but also outperforms GCA, both in terms of generality and the quality of the solutions produced. It is also faster than GCA on large graphs.

In the future, we will improve FMGCH with other geometric intuitions derived from the FastMap embedding of the graphs. We will also explore the importance of the graph convex hull problem in relation to other graph-theoretic problems. This can be done by following the analogy of the importance of the geometric convex hull problem in Computational Geometry.

References

1. Barber, C.B., Dobkin, D.P., Huhdanpaa, H.: The quickhull algorithm for convex hulls. ACM Transactions on Mathematical Software (1996)
2. Cohen, L., Uras, T., Jahangiri, S., Arunasalam, A., Koenig, S., Kumar, T.K.S.: The FastMap algorithm for shortest path computations. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence (2018)
3. Faloutsos, C., Lin, K.I.: FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (1995)
4. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM (1987)
5. Hagberg, A., Swart, P., S Chult, D.: Exploring network structure, dynamics, and function using NetworkX. Tech. rep., Los Alamos National Lab, Los Alamos, NM (United States) (2008)
6. Harris, B.: Mathematical models for statistical decision theory. In: Optimizing Methods in Statistics. Elsevier (1971)
7. Johnson, C.R.: Normality and the numerical range. Linear Algebra and its Applications (1976)

8. Katoh, N.: Bicriteria network optimization problems. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences (1992)
9. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data (2014)
10. Li, J., Felner, A., Koenig, S., Kumar, T.K.S.: Using FastMap to solve graph problems in a Euclidean space. In: Proceedings of the 29th International Conference on Automated Planning and Scheduling (2019)
11. Pelayo, I.M.: Geodesic convexity in graphs. Springer (2013)
12. Prasolov, V.V.: Polynomials. Springer Science & Business Media (2004)
13. Seidel, R.: Convex hull computations. In: Handbook of discrete and computational geometry. Chapman and Hall/CRC (2017)
14. Seiffarth, F., Horváth, T., Wrobel, S.: A fast heuristic for computing geodesic cores in large networks. arXiv preprint arXiv:2206.07350 (2022)
15. Sturtevant, N.: Benchmarks for grid-based pathfinding. Transactions on Computational Intelligence and AI in Games (2012)
16. Thiessen, M., Gärtner, T.: Active learning of convex halfspaces on graphs. Advances in Neural Information Processing Systems (2021)
17. Waxman, B.M.: Routing of multipoint connections. IEEE Journal on Selected Areas in Communications (1988)