Journal of Computer and Knowledge Engineering, Vol. 6, No. 1, 2023. (27-35)
DOI: 10.22067/cke.2023.73447.1033

Improving Greedy Spanner Construction Algorithm*

Research Article

Hosein Salami[1] Mostafa Nouri-Baygi[2]

Abstract: In recent years, several algorithms with different time complexities have been proposed for the construction of greedy spanners. However, a not so apparently suitable algorithm with running time complexity $O(n^3 \log n)$, namely the FG algorithm, is proved to be practically the fastest algorithm known for this task. One of the common bottlenecks in the greedy spanner construction algorithms is their use of the shortest path search operation (usually using Dijkstra's algorithm). In this paper, we propose some improvements to the FG algorithm in order to reduce the imposed costs of the shortest path search operation, and therefore to reduce the time required for the construction of greedy spanners. In the first improvement, we reduce the number of this operation calls and in the second one, we reduce the cost of each run of the operation. Experimental results show these improvements are able to significantly accelerate the construction of greedy spanners, compared to the other existing algorithms, especially when the stretch factor gets close to 1.

Keywords: Computational Geometry; Greedy Spanner; Construction Algorithm

1. Introduction

In most applications, designing a suitable network requires a short path between each pair of the network vertices. The simple idea to meet this requirement is to have a direct link between each pair of vertices, which transforms the network into a complete graph. A low-cost alternative for this requirement is spanner graphs. The spanners are some subgraphs of the complete graph that, despite having fewer edges than the complete graph, ensure that there is a short path for each vertex pair in the graph. The spanner graphs are introduced by Plug and Schaffer [1] in the context of distributed computing, and independently by Chew [2] in the geometric context.

In this paper, we focus on the geometric spanner graphs whose formal definition is as follows: Suppose a set V of n points on the plane and a real number $t > 1$. Let $G = (V, E)$ be a weighted graph such that the weight of each edge $e = (u, v) \in E$ is equal to the Euclidean distance between points u, v (e.g., $|uv|$). Graph G is called a geometric t-spanner on V if for each pair of points $u, v \in V$ there exists a path between u and v in G whose weight is at most $t|uv|$. This path is called a t-spanner path (or t-path for short) between u and v. The minimum value of t such that G is a t-spanner for V is called the *stretch factor* of G.

Since the introduction of spanner graphs, their efficient construction has been of interest to researchers. To date, several algorithms have been proposed to efficiently construct spanner graphs. These algorithms differ with respect to the quality of the resulted spanners (size, weight, maximum degree, etc.) or the features of the construction method (time and space complexity, ease of construction).

Althöfer et al. [3] proposed a greedy algorithm, named *Path-greedy*, for the construction of spanner graphs whose time and space complexity was $O(n^3 \log n)$ and $O(n^2)$, respectively. The spanner constructed by Path-greedy algorithm, called the greedy spanner, has $O(n)$ edges, maximum degree $O(1)$, and total weight $O(wt(MST(V)))$, where $wt(MST(V))$ is the weight of a minimum spanning tree of V. Although other types of spanners have also been suggested that can be constructed in $O(n \log n)$ time, empirical experiments [4] have shown that greedy spanners have higher quality than other types of spanner graphs.

This motivates the researchers to increase the effectiveness of Path-greedy algorithm [4, 5, 6, 7, 8, 9, 10, 11]. The aim of most efforts has been improving the time complexity, resulting in $O(n^2 \log n)$ algorithms [4, 5, 6], and more recently an algorithm [9] with $O(n\, log^2 n\, log^2 log\, n)$ average time complexity on points with uniform distribution.

Another set of attempts has been made to reduce the space complexity, resulting in linear space complexity algorithms with time complexity $O(n^2 \log n^2)$ [7] and recently $O(n\, log^2 n\, log^2 log\, n)$ [9], again in average for the set of points with uniform distribution.

1.1. Our contribution

Searching for the shortest path, using the Dijkstra's algorithm [11], is the most expensive part of the greedy spanner construction algorithms, so the main strategy of these algorithms for improving the greedy spanner construction process is limiting the number of times this operation is required to be invoked.

In this paper, we also follow this strategy by introducing some changes to the FG algorithm [4] that is one of the most cited algorithms for the construction of greedy spanner graphs. The changes offer low-cost alternatives for the role that is played by the Dijkstra's algorithm in the greedy spanner construction algorithms, that is, determining the *presence* or *absence* of a t-path between pair of points studied. To this end, the proposed improvements use some local information that is maintained by FG algorithm to estimate the presence or absence of a t-path between pair of points before invoking the Dijkstra's algorithm. In addition, we propose some changes to the Dijkstra's algorithm to avoid unnecessary actions and, therefore, reduce the cost of each execution of the operation.

Although these improvements do not change the time complexity of the FG algorithm, the results of experiments showed its effectiveness in constructing greedy spanners, especially when the stretch factor gets close to 1.

* Manuscript received, 2021 May 11, Revised, 2022 July 5, Accepted, 2023 January 28.

[1] PhD Candidate, Department of Computer Engineering, Ferdowsi University of Mashhad, Mashhad, Iran.
[2] Corresponding author. Assistant Professor, Department of Computer Engineering, Ferdowsi University of Mashhad, Mashhad, Iran.
Email: nouribaygi@um.ac.ir

The paper is organized as follows. In Section 0 we briefly describe some of the existing algorithms for constructing greedy spanners. In Section 0 we present and discuss our proposed improvements. Section 0 deals with the experiments that are done and the results achieved. Finally, Section 0 concludes the paper.

2. Greedy spanner construction algorithms

In almost all greedy spanner construction algorithms, construction process starts with an empty spanner graph $G = (V, \emptyset)$, and then the edges are added to partial spanner graph incrementally. Some greedy spanner construction algorithms [3, 4, 5, 6] process pairs of points in ascending order of their distances; so these algorithms start with sorting the pairs with respect to their distances.

In Path-greedy algorithm [3], processing of a pair of points (u, v) involves invoking Dijkstra's algorithm to determine the length of the shortest path between $u\ and\ v$ in the current spanner. If the length is greater than $t|uv|$, then the edge (u, v) is added to the spanner.

FG algorithm [4] acts like Path-greedy algorithm except that after invoking Dijkstra's algorithm, the achieved distances between point pairs are kept in a *distance matrix*. These distances are used for determining the existence of t-paths between next pairs of points before invoking the Dijkstra's algorithm.

Bose et al. [5] showed FG algorithm has $\theta(n^3 \log n)$ time complexity in the worst case. They presented an algorithm that we call BCFMS, whose time complexity is $O(n^2 \log n)$. In BCFMS algorithm, the objective is to keep the distance matrix values consistent with the current structure of the spanner. In this algorithm, the pairs of points are first divided into some buckets so that the size of the largest element in a bucket is at most 2 times the smallest element, and then the buckets are processed in the order of their sizes. At the beginning of the process of each bucket, the distance matrix is updated. During the process of a bucket, if one edge is added to the spanner, Dijkstra's algorithm is called from the points that are within a certain distance from the endpoints of the added edge to update their distance to other points.

Farshi and Hekmat Nasab [12] compared the time complexity of algorithms "Path-greedy", "FG", and "BCFMS" from a practical point of view.

Bar-on and Carmi [6] introduced an algorithm called δ-Greedy with $O(n^2 \log n)$ time complexity, where each point holds a set of *cones*. The characteristic of the cones of a point p is that if a point q lies in a cone, there is a t-path between p and q. Accordingly, the processing of a point pair begins by first examining if each point lies in the cone of the other one. If this is not the case, Dijkstra's algorithm is called to find the shortest path between them.

Some algorithms are presented that allow the construction of greedy spanners with linear space [7, 8, 9, 10]. These algorithms use some methods to partition $\binom{n}{2}$ pairs of points to $O(n)$ partitions of points. The resulting partitions are then processed to construct the greedy spanner. Bakhshesh and Farshi [10] used cones for partitioning and provided a linear space algorithm with $O(n^3 log^2 n)$ time complexity. It should be noted that we do not consider their proposed algorithm in our experiments.

Alewijnse et al. [7] used Well Separated Pair Decomposition (WSPD) to partition the pair of points. They showed that the time complexity of their algorithm, we call WSPD, is $O(n^2 log^2 n)$. Bouts et al. [8] proposed another partitioning method and showed that their proposed algorithm, named Lazy-greedy, reduces the memory requirement of the WSPD algorithm by a factor of $O(1/(t-1))$. By introducing and applying the greedy spanner algorithms framework, they also showed that Lazy-greedy has a time complexity of $O(n^2 \log n \log \Phi)$, where Φ is a function of the distribution of points.

Bouts et al. [9] presented a three-step algorithm that has average time complexity $O(n\ log^2 n\ log^2 log\ n)$ for a set of points that are distributed uniformly. Considering the fact that in the greedy spanners, most of edges are small, their proposed algorithm, named Bucket, in the first step only processes close pairs of points to identify small edges with a method similar to FG algorithm. In the second step, the remaining pairs of points are partitioned. The authors showed that most of obtained partitioned can be bypass using evidence obtained from edges added in the first step. Finally, the algorithm processes the remaining partitions using the WSPD algorithm to identify possible large edges.

3. Proposed improvements

3.1. Reducing the number of shortest path search operation calls

Determining the existence of a t-path between two given points p and q by searching for the shortest path between them imposes a high cost. In this section we describe two low-cost alternatives that estimate the presence/absence of a t-path between p and q using some local information. In the following sections, we use $N(p)$ as the set of neighbors of p in G, that is, $N(p) = \{q | q \in V, (p,q) \in E\}$. Also $w(p,q)$ indicates the corresponding element of pair p, q in the distance matrix.

3.1.1. Estimating the presence of a t-path

We apply and extend the idea of FG algorithm to estimate the distance between vertices using the distance matrix, before invoking the Dijkstra's algorithm. For this purpose, when the pair (p,q) is to be processed, in addition to examining their corresponding element in the distance matrix, e.g., $w(p,q)$, for all vertices $r \in N(p)$, the existence of a t-path between p and q using r is examined.

Suppose that r is a neighbor of p. We know that an upper bound for the length of the path between $p\ \ and\ q$ that passes through r is $|pr| + t|rq|$. Clearly if this value is at most $t|pq|$, we have already a t-path between $p\ and\ q$. However, it could be the case that the value of $w(r,q)$ is much less than $t|rq|$, so, in our modified FG algorithm, referred to as IFGBN1, we use the most recent value of $w(r,q)$ to estimate the distance between r and q.

The following lemma, which is a modified version of Lemma 1 in [6], gives a sufficient condition for the existence of a t-path.

Lemma 1. Let t and θ be real numbers, such that $t \geq 1$ and $0 \leq \theta \leq \frac{\pi}{4}$. Let p, q, and r be points in the plane and $r \in N(p)$, such that:

1. $|pr| \leq |pq|$,
2. $\frac{1}{(cos\theta - sin\theta)} \leq t$, where θ is the angle $\angle rpq$, in other words $\angle rpq = \theta \leq \frac{\pi}{4} - arcsin(\frac{1}{t\sqrt{2}})$

Then $|pr| + t|rq| \le t|pq|$.

Proof. Let r' be the orthogonal projection of r onto segment pq (see Figure 1). Then $|rr'| = |pr|\sin\theta$, $|pr'| = |pr|\cos\theta$, and $|r'q| = |pq| - |pr'|$.
Thus

$$|r'q| = |pq| - |pr|\cos\theta \quad (1)$$

By triangle inequality

$$|rq| \le |rr'| + |r'q| \le |pr|sin\theta + |pq| - |pr|\,cos\theta = |pq| - |pr|(cos\theta - sin\theta). \quad (2)$$

We have,

$$|pr| + t|rq| \le |pr| + t(|pq| - |pr|(cos\theta - sin\theta) = t|pq| - |pr|(t(cos\theta - sin\theta) - 1) \le t|pq|. \quad (3)$$

Lemma 1 states that for every neighbor r of p, there exists a cone with p as the apex and ray $\overrightarrow{pr}$ as the bisector, such that for each point q inside the cone, there is a t-path from p to q (see
Figure 2). The thickness of the cone depends on the value of the stretch factor, i.e., the apex angle of the cone decreases as the stretch factor t gets close to 1. We call this apex the clear apex.

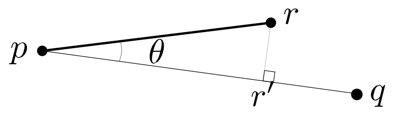


Figure 1. Illustration of Lemma 1

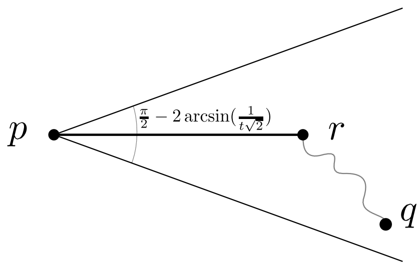


Figure 2. Demonstration of a clear apex

Note that we can interpret Lemma 1 in another way. Consider the moment that the pair (p,q) is to processed. We have a cone with p (or q) as its apex, apex angle equals to $\frac{\pi}{4} - arcsin(\frac{1}{t\sqrt{2}})$ and pq as its bisector. If the apex has a neighbor r that lies in this cone, the lemma assures that there exists a t-path between p and q.

Lemma 1 uses the worst-case value of $t|rq|$ as the distance between r and q, however as stated above, it could be the case that the value of $w(r,q)$ is much less than $t|rq|$, so ignoring neighbors of p (resp. q), whose clear apex does not contain q (resp. p), may cause losing some potential t-paths. As a result, IFGBN1 investigates all neighbors of p (resp. q).

3.1.2. Estimating the absence of a t-path

In this section we show how it is possible to definitely estimate the absence of a t-path between two points. Before describing the method, we have to define an *obligated pair*. We call (p,q) an obligated pair if for every path π between p and q there exists at least one point in π such that it lies outside the closed *ellipse* with the focal points p and q, and eccentricity $1/t$, that is the ellipse defined as:

$$e_{pq} = \{r \in \mathbb{R}^2 |\ |pr| + |rq| \le t|pq|\} \quad (4)$$

Our definition of obligated pair is a generalization of the concept *mandatory pair* provided by Bouts et al. [9]. Note that the set of mandatory pairs is a subset of the set of obligated pairs. In other words, each mandatory pair is an obligated pair, but the opposite is not necessarily true.

Lemma 2. Given an obligated pair (p,q), for any t-spanner G, $(p,q) \in E(G)$.

Proof. Assume, for the sake of contradiction, that there exists a t-spanner G such that $(p,q) \notin E(G)$. It means there exists a path $\langle p, r_1, r_2, \dots, r_k, q\rangle$ between p and q in G whose length is at most $t|pq|$. Because of the definition of obligated pairs, there exists a point r_j, $1 \le j \le k$ in this path such that r_j does not lie in ellipse e_{pq}. In other words.

$$|pr_j| + |r_jq| > t|pq| \quad (5)$$

We can bound the length of the path as follows:

$$|\langle p, r_1, \dots, r_k, q\rangle| = |\langle p, r_1, \dots, r_j\rangle| + |\langle r_j, \dots, q\rangle| \quad (6)$$

$$\ge |pr_j| + |r_jq| \quad (7)$$

$$> t|pq| \quad (8)$$

which contradicts the assumption that the path is a t-path. Equation **Error! Reference source not found.**7 follows the triangle inequality, and Equation 8 by applying Equation 5.

Using Lemma 2, when processing an obligated pair (p,q), we just need to add edge (p,q) to the spanner, without searching for the shortest path between them. But testing whether a given pair (p,q) is obligated is rather time-consuming. We here use a simple method to find some obligated pairs quickly.

We denote the set of neighbors of p that lie in e_{pq} as $ELN(p,q)$. When processing (p,q), if either of $ELN(p,q)$ or $ELN(q,p)$ is empty, then (p,q) is obligated. Now suppose that $ELN(p,q)$ and $ELN(q,p)$ are not empty. If there exists a t-path between p and q such as $\pi = \langle p, r_1, \cdots, r_k, q\rangle$, the points r_1 and r_k will be members of $ELN(p,q)$ and $ELN(q,p)$, respectively. Because of the triangle inequality, the length of $\langle p, r_1, r_k, q\rangle$ is an upper bound for the length of π. Therefore, we consider lengths of all paths like $\langle p, r_i, r_j, q\rangle$, where $r_i \in ELN(p,q)$ and $r_j \in ELN(q,p)$, and if for all of them the spanning condition does not hold, we conclude that (p,q) is an obligated pair.

We modify the original FG algorithm so that it uses this method before performing the shortest path search operation. The modified algorithm, referred to as IFGBN2, will add the edge (p,q) to the spanner if the aforementioned method shows the infeasibility of having a t-path between p and q.

Algorithm 1 shows the final algorithm (IFGBN) after applying the two proposed modifications to the original FG algorithm. In the new algorithm, lines 10-14 apply IFGBN1 and lines 15-21 apply IFGBN2.

Algorithm 1. IFGBN

```
Input: A set $P$ of points in the plane and a real number $t$
Output: A Greedy $t$-spanner for $P$
1: sort the $\binom{n}{2}$ pairs of distinct points in non-decreasing order of their distances and store them in list $L$;
2: $E \leftarrow \emptyset$
3: $G \leftarrow (P, E)$
4: $w(p,q) \leftarrow \infty \ \forall p,q \in P$
5: $w(p,p) \leftarrow 0 \ \forall p \in P$
6: foreach $(p,q) \in L$ (in sorted order) do
7:     $if\ w(p,q) \le t|pq| then$
8:         $continue$
9:     $end$
10:    $nearestNeighbor \leftarrow min(\{|pr| + w(r,q) \mid r \in N(p)\} \cup \{|qr| + w(r,p) \mid r \in N(q)\})$
11:    $w(p,q) \leftarrow min(w(p,q),\ nearestNeighbor)$
12:    $if\ w(p,q) \le t|pq|\ then$
13:        $continue$
14:    $end$
15:    $ELN(p,q) \leftarrow N(p) \cap e_{pq}$
16:    $ELN(q,p) \leftarrow N(q) \cap e_{pq}$
17:    $if\ \forall r_i \in ELN(p,q), r_j \in ELN(q,p), |< p, r_i, r_j, q >| > t|pq|\ then$
18:        $E \leftarrow E \cup \{(p,q)\}$
19:        $w(p,q) \leftarrow |pq|$
20:        $w(q,p) \leftarrow |pq|$
21:    $end$
22:    $else$
23:        $perform\ an\ SSSP\ computation\ in\ G\ with\ source\ p$
24:        $foreach\ v \in P\ do$
25:            $update\ w(p,v)\ and\ w(v,p)$
26:        $end$
27:        $if\ w(p,q) > t|pq|\ then$
28:            $E \leftarrow E \cup \{(p,q)\}$
29:            $w(p,q) \leftarrow |pq|$
30:            $w(q,p) \leftarrow |pq|$
31:        $end$
32:    $end$
33: $end$
34: $return\ G$
```

3.2. Reducing Dijkstra's algorithm cost

In the original FG algorithm, when searching for the shortest path from a point (using Dijkstra's algorithm), shortest paths are computed to all other points of the graph, and then the obtained distances are stored in the distance matrix. These costs are paid in the hope that, when a pair (p, q) is to be processed later, the stored values proof the existence of a t-path and so avoid the need for invoking Dijkstra's algorithm.

However, if the obtained shortest path did not have t-path condition, the paid costs are wasted, because it is required to run Dijkstra's algorithm once again. To avoid these extra costs, we propose a change to Dijkstra's algorithm. Before describing this change, consider the following lemma.

Lemma 3. Let $\pi = \langle p, r_1, r_2, \ldots, r_k, q\rangle$ be a t-path between pair p and q. For all $i = 1 \ldots k$, there exist a t-path between p and r_i.

Proof. For each $r_i \in \pi$, we have one of the following two cases:

1. $|pq| > |pr_i|$
2. $|pq| \le |pr_i|$

In the case (i), prior to processing (p, q), the pair (p, r_i) has been processed and therefore there is a t-path between them.

For the case (ii), let $d(p, r_i)$ and $d(p, q)$ denote the distance from p to r_i and q in π, respectively. We know:

$$\begin{aligned} d(p,r_i) &\le d(p,q) \\ &\le t|pq| \\ &\le t|pr_i| \end{aligned} \tag{9}$$

Lemma 3 states that if a t-path exists between p and q, which passes through a point r, then there will also be a t-path between p and r. In other words, if Dijkstra's algorithm from p reaches a point r, and the resulting path is not a t-path, then r could not be a middle point in a t-path from p to other points. According to Lemma 3, Dijkstra's algorithm is modified as shown in Algorithm 2.

Algorithm 2. ModifiedDijkstra(G, p, t)

```
$\textbf{Input}$: $Current\ Graph\ G, points\ p \in V(G)\ and\ real\ number\ t$
$\textbf{Output}$: $Shortest\ path\ from\ p\ to\ all\ points\ in\ V(G)$
1: $d(p) \leftarrow 0$
2: $d(v) \leftarrow \infty\ \ for\ all\ \ v \in V - \{p\}$
3: $Q \leftarrow V$
4: $\textbf{while}\ Q \ne \emptyset\ \textbf{do}$
5:     $r \leftarrow mindistance(Q, d)$
6:     $\textbf{if}\ d(r) = \infty\ \textbf{then}$
7:         $\textbf{break}$
8:     $\textbf{end}$
9:     $\textbf{foreach}\ v \in neighbors[r]\ \textbf{do}$
10:        $\textbf{if}\ d(v) > d(r) + dist(r,v)\ \textbf{then}$
11:            $\textbf{if}\ d(r) + dist(r,v) \le t|pv|\ \textbf{then}$
12:                $d(v) \leftarrow d(r) + dist(r,v)$
13:            $\textbf{end}$
14:        $\textbf{end}$
15:    $\textbf{end}$
16: $\textbf{end}$
17: $\textbf{return}\ d$
```

Line 11 of Algorithm 2 has been added to Dijkstra's algorithm to apply Lemma 3. With this change, point r is considered in the shortest path from p to v only if, in addition to shortening the distance between p and v, the path constructed is also a t-path between p and r. Note that for any point v, if none of the paths to v satisfies the t-path condition, the value of v will not be updated in the distance vector, and whenever v is processed in line 5, Dijkstra's algorithm will end, which prevents it from spending more wasted time in Dijkstra's algorithm.

4. Experimental results

Performance of the proposed changes to the original FG algorithm compared to other greedy spanner construction algorithms are presented in this section. Algorithms used include well-known greedy spanner construction algorithms (such as FG [4], BCFMS [5]) and algorithms that have been proposed in recent years (including WSPD [7], Lazy [8], Bucket [9] and δ-Greedy [6]). The criteria used for the comparisons include the number of times the shortest path search is called in the construction process (Section 0) and the duration of the construction (Section 0).

In both Sections 0 and 0, first, the original FG algorithm is compared with the two algorithms IFGBN1 and IFGBN2. We remind that IFGBN1 algorithm is a version of the original FG algorithm in which the proposed method in Section 0 is used. Likewise, IFGBN2 algorithm refers to a version of the original FG algorithm in which the proposed method in Section 0 is utilized. After that, IFGBN algorithm (a version of the original FG that incorporates both improvements proposed in Section 0) will be compared with other algorithms.

In Section 0, we compare FG and IFGBC algorithms, which is a version of the original FG that performs the shortest path search using the modified Dijkstra's algorithm presented in Section 0. Since the proposed method in Section 0 does not affect the number of times the shortest path search is invoked, these two algorithms will be compared only based on the duration of greedy spanner construction.

It should be noted that the output of all algorithms, except δ-Greedy, is greedy spanner. So, the results are of the same quality. In the case of δ-Greedy algorithm, there is a parameter called δ that according to [6], the output of the algorithm will be a greedy spanner whenever its value is equal to the stretch factor, therefore we set $\delta = t$ in the experiments.

All the experiments except for BCFMS algorithm were performed on point sets of size 500 to 16,000 and the stretch factor between 1.1 and 1.01. Due to the high memory usage of BCFMS algorithm, the experiments on this algorithm are limited to 4000 points. The point sets are randomly generated in the plane with uniform and clustered distributions, and with random coordinates ranging from 0 to 30,000. We followed the method proposed by Alewijnse et al. [7, 9] to generate points with clustered distribution.

We used the source code shared by Alewijnse et al. [7, 9] for WSPD, Lazy, Bucket, FG and BCFMS algorithms. Moreover, to implement the δ-Greedy and our proposed algorithms, we used their codes as much as possible. The source codes were written in C++ language and compiled into machine codes using Visual Studio 2015 compiler. The experiments performed on a machine with an Intel Core i7-7700HQ processor with 16GB of main memory on Windows 10 operating system.

4.1. Effects on the number of shortest path search operations

Figure 3 shows the number of shortest path searches performed by IFGBN1, IFGBN2 and FG algorithms.

As can be seen, in all conditions both IFGBN1 and IFGBN2 algorithms perform a smaller number of searches than FG algorithm.

Nonetheless IFGBN1 and IFGBN2 act differently in different distributions. While in the clustered distribution, IFGBN1 has performed the least number of searches, in the uniform distribution, the least number of searches was performed by IFGBN2. The effect of stretch factor is also different in the performance of IFGBN1 and IFGBN2. While IFGBN1 performs more searches and achieves a performance similar to FG as the stretch factor decreases, IFGBN2, despite the relative increase in the number of searches, showed to be less affected by the change in the stretch factor.

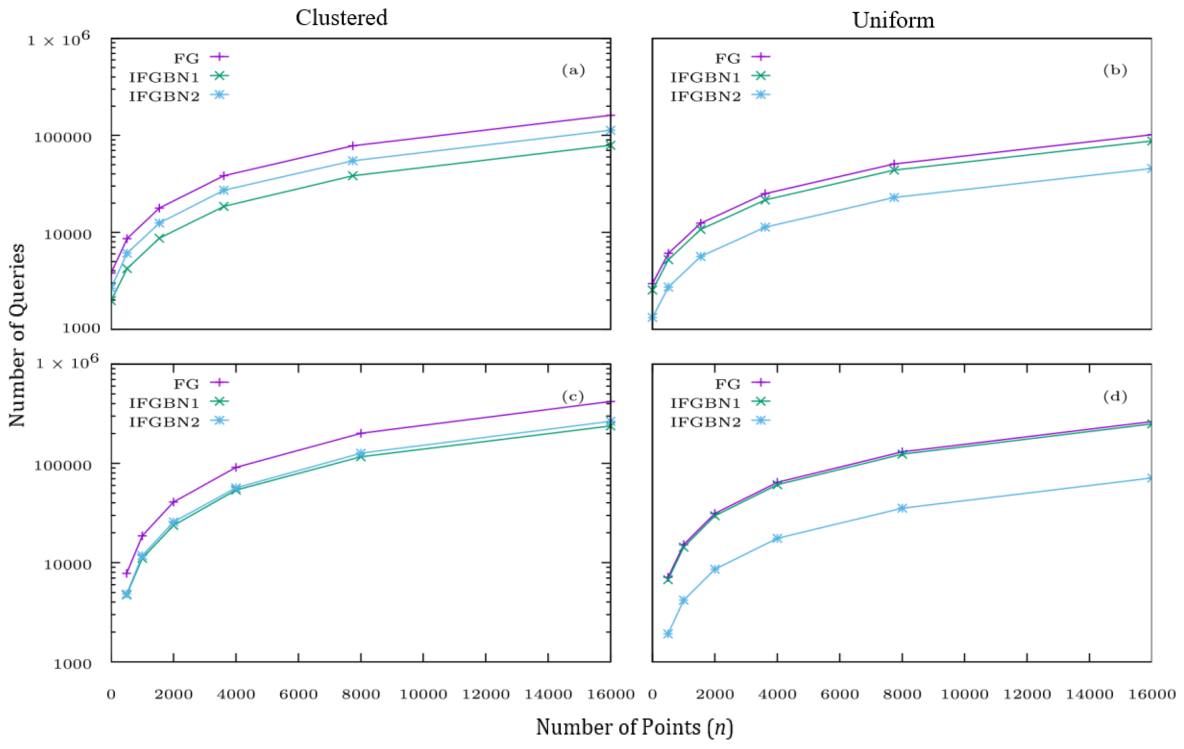


Figure 3. The number of the shortest path search operation performed by IFGBN1, IFGBN2 and FG algorithms
t = 1.1 (a, b), t = 1.01 (c, d)

In the case of IFGBN2, it seems that reducing the stretch factor increases the probability that the pairs will be obligated, and therefore, cutting down the number of shortest path searches. However, the results do not support this hypothesis. For further investigations, the performance of this algorithm was tested for smaller values of the stretch factor. In these experiments, we investigate the number of times the shortest path search operation is called by IFGBN2 on six-point sets of size 100, 250, and 500, with uniform and clustered distributions. The stretch factor values are selected in the range [1.00001…1.1].
Figure 4 shows the results.

According to the results, even though decreasing the stretch factor initially increases the number of shortest path searches, as the stretch factor continues to decrease, this value begins to reduce. The starting point of this reduction partly depends on the number and distribution of points. This effect happens earlier for smaller point sets and for the uniform distribution.

Now, we compare the performance of IFGBN with other algorithms in terms of the number of shortest path searches. As shown in
Figure 5, with a great difference, IFGBN algorithm has the smallest number of shortest path searches, under different conditions such as the number of points, distributions, and different values of the stretch factor. On the other hand, WSPD algorithm performs the greatest number of searches compared to other algorithms. In general, the performance of the algorithms which are based on the well separated pair decomposition (e.g., WSPD and Bucket) for clustered points sets is better than uniform points set [9]. In addition, reducing the stretch factor in the uniform distribution had more negative impact on WSPD algorithm than in the clustered distribution.

Even though IFGBN has the minimum number of shortest path searches, its searches increase as the stretch factor decreases. IFGBN inherits this behavior from IFGBN2. To ensure this, the performance of IFGBN was investigated at smaller values of the stretch factor, the results of which are given in Figure 6. The results confirm that the number of shortest path searches increases only within a small range of t, and then it starts to decrease again, similar to IFGBN2 algorithm.

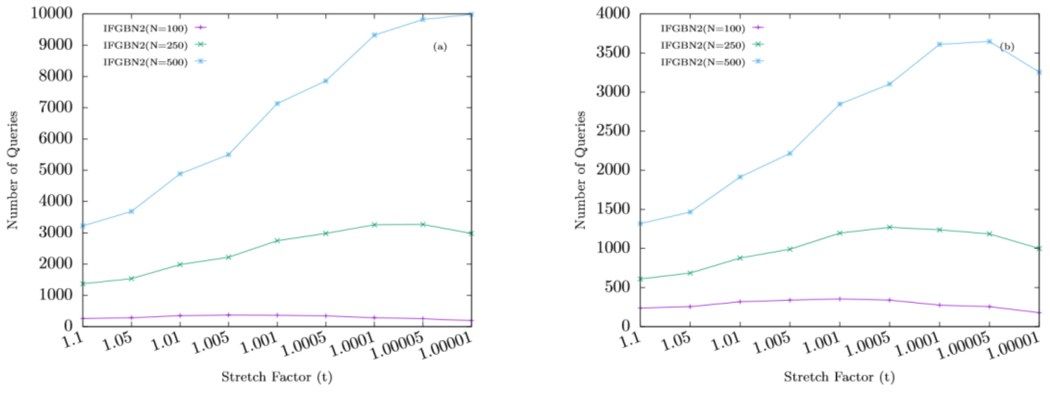


Figure 4. The number of shortest path search operation performed by IFGBN2 algorithm. (a): Clustered, (b): Uniform

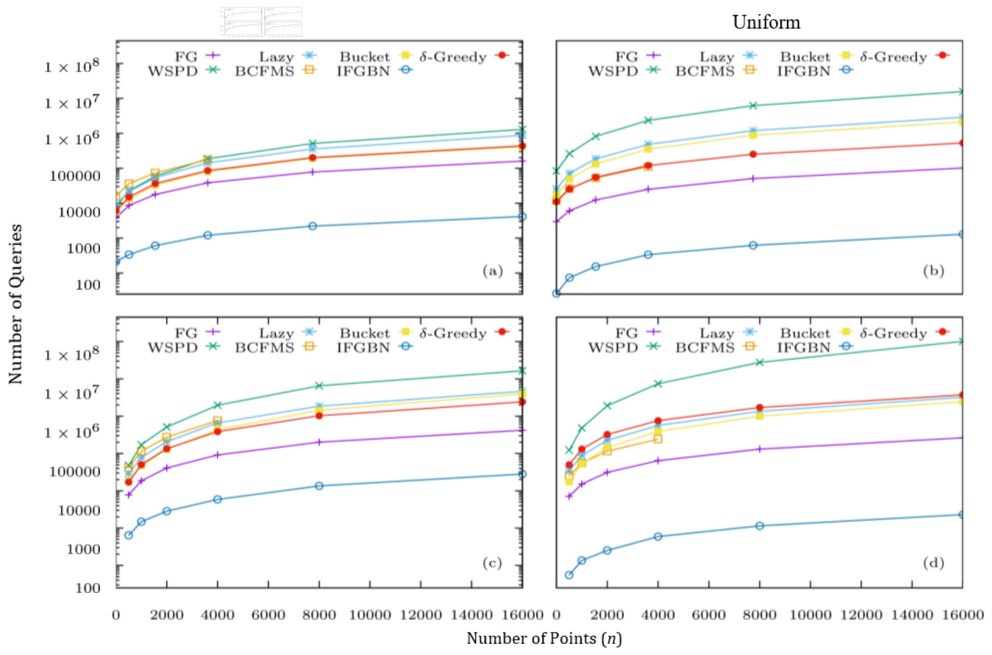
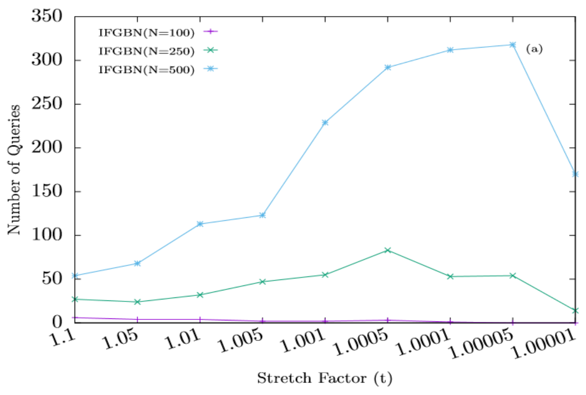


Figure 5. The number of shortest path search operation performed by the construction algorithms. t = 1.1 (a, b), t = 1.01 (c, d)



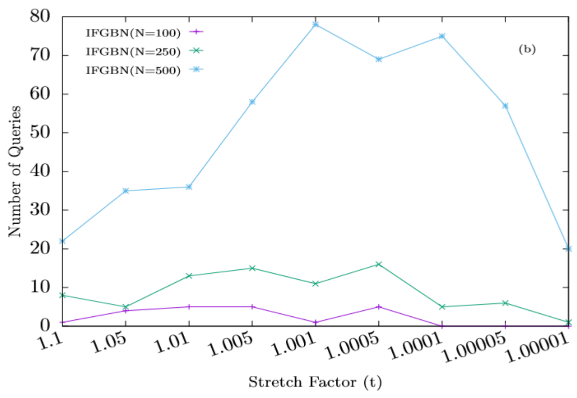


Figure 6. The number of shortest path searches performed by IFGBN algorithm. (a): Clustered, (b): Uniform

4.2. Effects on the running-time

In this section, the construction time of the greedy spanner by construction algorithms is presented. Similar to the previous section, we will again examine the results of the proposed improvements in Section 0 separately and in combination. Figure 7 and 8 show the results of the experiments. As can be seen, there is a significant relationship between the number of shortest path searches performed by the algorithms (Figure 3 and 4), and the duration of the construction of the greedy spanner.

IFGBN always spends less time to construct a greedy spanner. Furthermore, reducing the stretch factor amplifies the gap between IFGBN and other algorithms. The worst algorithm in terms of the running time depends on the distribution of points. In the clustered distribution, BCFMS, and in the uniform distribution, WSPD spends the most time to construct greedy spanners. According to the results, Bucket algorithm, similar to IFGBN, is less sensitive to the stretch factor.

4.3. Effect of reducing the cost of Dijkstra's algorithm

Figure 9 shows the greedy spanner construction time of two algorithms, FG and IFGBC (which is a version of FG that uses the modified version of the Dijkstra's algorithm presented in Algorithm 2). Although there is a great similarity between the performance of the two algorithms, as can be seen, by reducing the stretch factor or increasing the number of points, IFGBC algorithm gradually shows its superiority over FG and increases its distance with it. Furthermore, the distribution of points does not have much impact on the behavior of IFGBC and it has acted almost identically in both distributions.

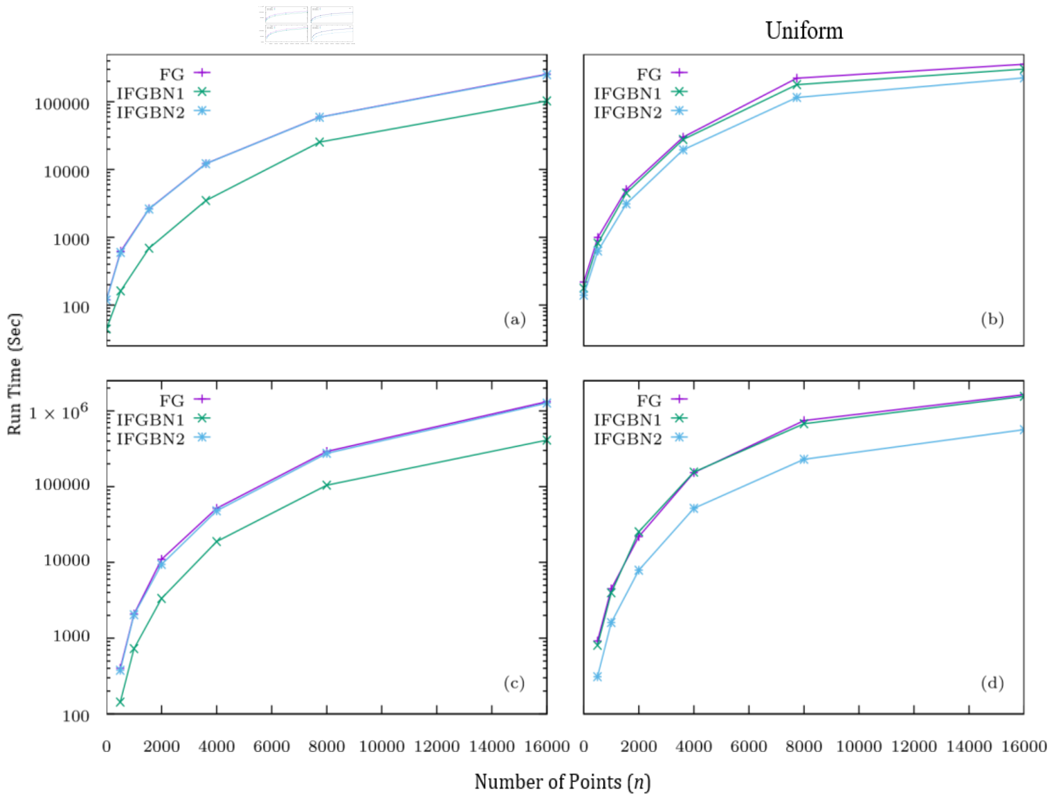


Figure 7. Running time of IFGBN1, IFGBN2 and FG algorithms for constructing greedy spanner. t=1.1 (a, b), t=1.01 (c, d)

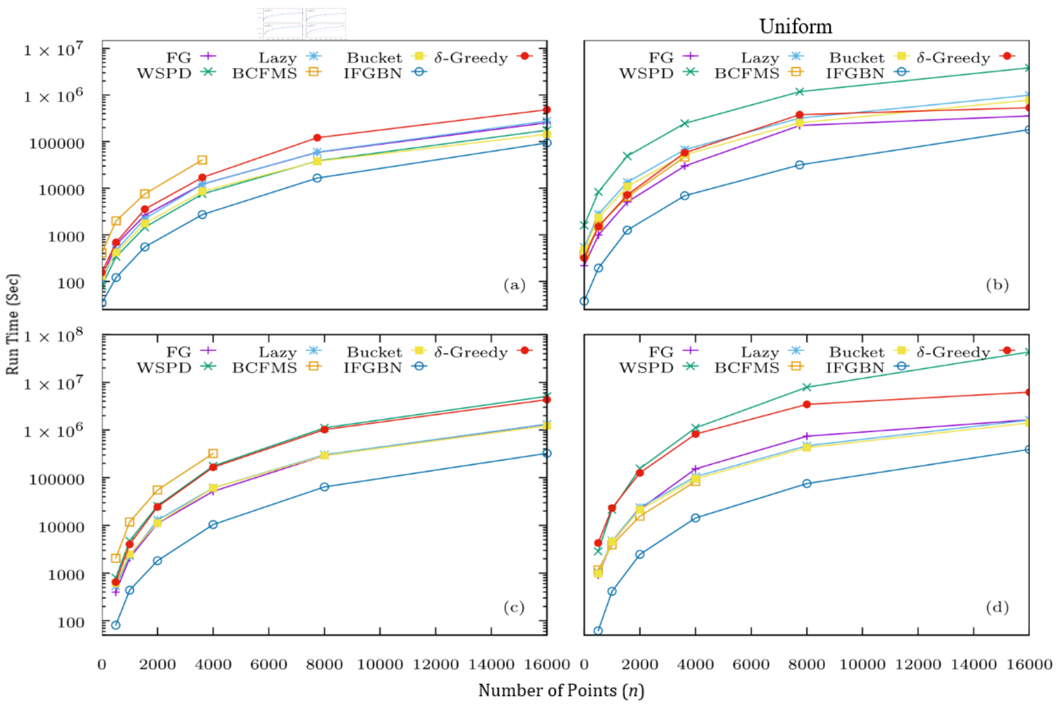


Figure 8. Running time of greedy spanner construction algorithms. t=1.1 (a, b), t= 1.01 (c, d)

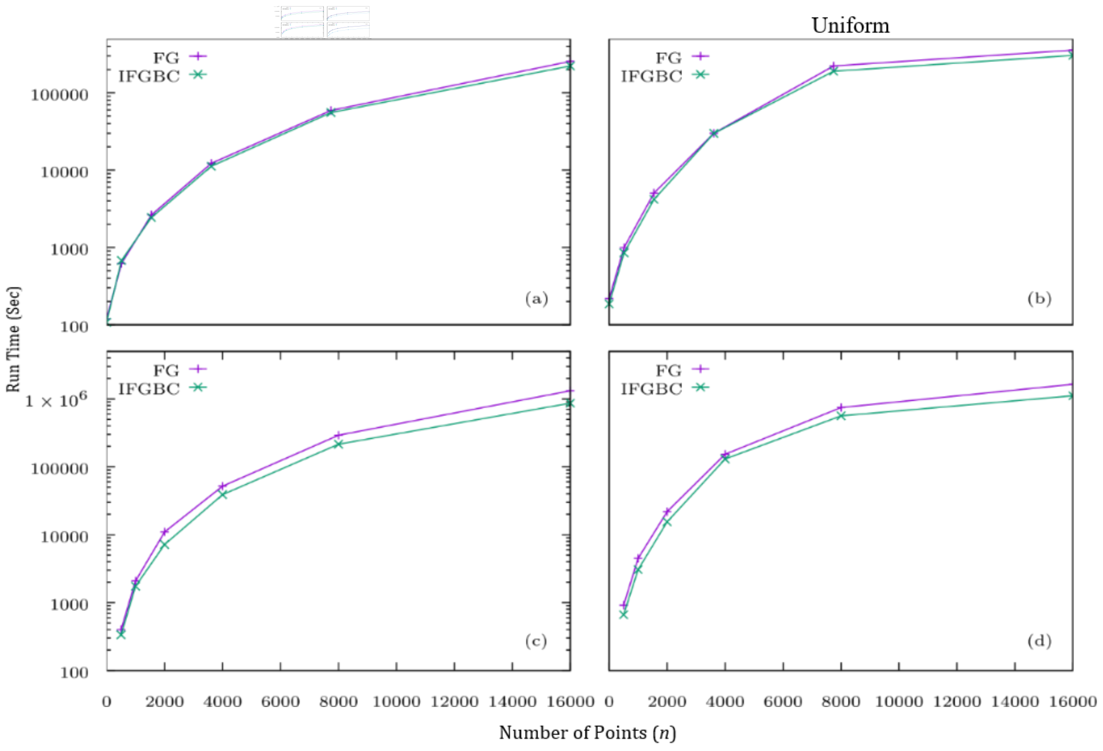


Figure 9. Time spent by FG and IFGBC algorithms for constructing greedy spanner. t= 1.1 (a, b), t = 1.01 (c, d)

5. Discussion

The results presented in the previous sections showed that each of the proposed improvements provide better performance in certain conditions.

IFGBN1 algorithm performs well when the stretch factor is large, but as the stretch factor gets closer to 1, it loses its effectiveness in improving the performance of FG algorithm. As mentioned in Section 0, the thickness of the cones around a point p depends on the value of the stretch factor in such a way that its size decreases with the decrease of the stretch factor. Shrinking the thickness of the cones reduces the probability of placing other points in them and therefore increases the need to search for the shortest path. This justifies the inappropriate performance of IFGBN1 in small values of stretch factor.

IFGBN2 algorithm does not perform very well at first when the stretch factor approaches 1, but it performs better as stretch factor gets closer to 1. This behavior can be attributed to the increase in the degree of vertices of the greedy spanner at lower values of the stretch factor. As the degree of vertices increases, the sets $ELN(p,q)$ and $ELN(q,p)$ become larger, and hence the likelihood of finding a path that is eligible for being t-path increases. However, as the value of the stretch factor becomes closer to 1, the ellipse e_{pq} gradually converges to a line segment, and thus more pairs become obligated.

IFGBN algorithm takes best advantage of both improvements and, as presented in Figure 5 and Figure 8, exhibits good performance in most conditions.

6. Conclusion

One way of accelerating the process of constructing the greedy spanners is to replace or modify Dijkstra's algorithm due to its costly nature. In this paper, two improvements were made to the process of creating the greedy spanners: 1) reducing the number of Dijkstra's call; and 2) reducing the running time of Dijkstra's algorithm. Reducing the number of Dijkstra's call is achieved by replacing it with heuristics that approximate the distance between the points. In order to reduce the cost of executing Dijkstra's algorithm, the algorithm was modified in such a way that it advances in the graph only on paths that their results can be used for following steps.

The experiments showed that the proposed improvements, especially when the stretch factor is close to 1, have significant impacts on reducing the running time. Furthermore, the results showed the decisive role of Dijkstra's algorithm in the running time and emphasized the need to pay more attention to replacing or modifying this algorithm in the process of the greedy spanner construction.

Acknowledgement

We greatly appreciate Dr. Kevin Buchin for sharing source code of their proposed algorithms.

6. References

[1] D. Peleg and A. A. Schäffer, "Graph spanners," Journal of graph theory, vol. 13, no. 1, pp. 99–116, 1989.
[2] L. P. Chew, "There are planar graphs almost as good as the complete graph," Journal of Computer and System Sciences, vol. 39, no. 2, pp. 205–219, 1989.
[3] I. Althöfer, G. Das, D. Dobkin, D. Joseph, and J. Soares, "On sparse spanners of weighted graphs," Discrete & Computational Geometry, vol. 9, pp. 81–100, Jan 1993.
[4] M. Farshi and J. Gudmundsson, "Experimental study of geometric t-spanners: A running time comparison," in International Workshop on Experimental and Efficient Algorithms, pp. 270–284, Springer, 2007.
[5] P. Bose, P. Carmi, M. Farshi, A. Maheshwari, and M. Smid, "Computing the greedy spanner in near-quadratic time," Algorithmica, vol. 58, no. 3, pp. 711–729, 2010.
[6] G. Bar-On and P. Carmi, "δ-greedy t-spanner," Computational Geometry 100 (2022): 101807.
[7] S. P. Alewijnse, Q. W. Bouts, P. Alex, and K. Buchin, "Computing the greedy spanner in linear space," Algorithmica, vol. 73, no. 3, pp. 589–606, 2015.
[8] Q. W. Bouts, A. P. ten Brink, and K. Buchin, "A framework for computing the greedy spanner," in Proceedings of the thirtieth annual symposium on Computational geometry, pp. 11–19, ACM, 2014.
[9] S. P. Alewijnse, Q. W. Bouts, P. Alex, and K. Buchin, "Distribution-sensitive construction of the greedy spanner," Algorithmica, vol. 78, no. 1, pp. 209–231, 2017.
[10] D. Bakhshesh and M. Farshi, "A new construction of the greedy spanner in linear space," in 1st Iranian Conference on Computational Geometry, pp. 33–36, 2018.
[11] E. W. Dijkstra et al., "A note on two problems in connexion with graphs," Numerische mathematik, vol. 1, no. 1, pp. 269–271, 1959.
[12] M. Farshi, and M. J. HekmatNasab. "Greedy spanner algorithms in practice." Scientia Iranica, vol. 21, no. 6, pp. 2142-2152, 2014.