Strategies for generating tree spanners: algorithms, heuristics and optimal graph classes

Fernanda Couto[a,1], Luís Felipe I. Cunha[b,1], Daniel Juventude[a], Leandro Santiago[b,1]

[a] *Universidade Federal Rural do Rio de Janeiro, Nova Iguaçu, Brazil*
[b] *Universidade Federal Fluminense, Niterói, Brazil*

Abstract

The t-admissibility problem aims to decide whether a graph G has a spanning tree T in which the distance between any two adjacent vertices of G is at most t. Regarding its optimization version, we want to determine the smallest $t = \sigma_T(G)$ for which G is t-admissible, i.e., the stretch index of G. The t-admissibility problem is NP-complete for $\sigma_T(G) \leq t$, $t \geq 4$. We develop strategies and implementations of two exact brute-force algorithms: sequential and parallel, in such a way that we propose parallel strategies for generating all spanning trees of a graph. We also propose two greedy heuristics to obtain solutions for the t-admissibility problem. We evaluate these implementations on random Barabási-Albert graphs, Erdös-Rényi graphs, Watts-Strogatz graphs and Bipartite graphs. Moreover, we determine graph classes for which the heuristics lead us to the stretch index value, such as split graphs, cographs and p-cycle power graphs.

Keywords: t-admissibility, stretch index, heuristics.

1. Introduction

Graphs are powerful tools widely used for modeling several real problems and solving them by computational efforts. The t-admissibility problem is a great challenging problem on graphs [4], it is a min-max problem where, essentially, opposite optimal analyzes are done within the same context.

*Corresponding author

Email addresses: fernandavdc@ufrrj.br (Fernanda Couto), lfignacio@ic.uff.br (Luís Felipe I. Cunha), deljuven@ymail.com (Daniel Juventude), leandro@ic.uff.br (Leandro Santiago)

Given a spanning tree T of G, if the maximum distance in T between adjacent vertices in G is t, then G is *t-admissible*. The integer t is called the *stretch factor* of T (or simply the *factor* of T), denoted by $\sigma(T)$. The smallest t such that G is t-admissible is the *stretch index* of G (or simply *index* of G), denoted by $\sigma_T(G)$. The *t-admissibility* problem aims to decide whether a given graph G has $\sigma_T(G) \leq t$.

A brute-force strategy to solve such a problem is as follows. After generating all spanning trees of G, we determine each stretch factor and set the smallest one as the stretch index of G. Although this strategy is correct, it is not efficient, since there is an exponential number of spanning trees of a graph, *cf.* [21].

Cai and Corneil [4] proved that t-admissibility is NP-complete, for $t \geq 4$, whereas 2-admissible graphs can be recognized in polynomial time. However, surprisingly, the characterization of 3-admissible graphs has been an open problem for over 20 years. In very recent works, some advances in the theoretical context of this problem were accomplished when restricted to graph classes [5, 6, 7].

Due to the difficulty of obtaining the stretch index of a graph, heuristics and approximation algorithms are good alternatives. An $O(\log n)$-approximation algorithm was developed in [11], and the authors proved that it cannot be approximated by adding any $o(n)$ term. Heuristics and integer programming formulation approaches were recently developed considering the edge-weighted version of t-admissibility. Such strategies were implemented for specific graph classes [1, 9, 17, 19].

To evaluate how close to an optimal solution these heuristics are, comparisons with brute-force algorithms are a possible approach, but inefficient. Parallelizing brute-force algorithms might be a good attempt to improve their efficiency. As far as we know, in the literature the unique parallel strategy on this context is to construct tree 3-spanners on interval graphs [16].

Our main interest is to develop, implement and evaluate algorithms. We develop sequential and parallel algorithms (Sections 2.1 and 2.2), and propose two simple greedy heuristics (Sections 3.1 and 3.2). Moreover, we determine graph classes for which the proposed heuristics succeed on determining the stretch index (Section 3.3). In addition, we implement these algorithms for random graphs (Section 4.1) and compare heuristic solutions with those of brute-force and known lower bounds (Section 4.2). Such strategies give us clues to seek theoretical foundation of our results and also to understand the computational gain of parallelism, making possible further studies on this problem (Section 5). All the implementations and analyses we developed are available in [8].

2. Brute-force algorithms

In order to evaluate the heuristics proposed in Section 3, we develop a sequential strategy (Section 2.1). Moreover, to improve the running time and deal with larger graphs (where we say that the size of $G = (V, E)$ is given by $|V| + |E|$), we develop a parallel strategy (Section 2.2) at thread level on the generation of the spanning trees.

2.1. Sequential strategy

Algorithm 1 generates a spanning tree T of a graph G and after that, its factor is obtained by calculating the distances in T between all adjacent vertices of G (made by *factor(tree)*).

We consider two lower bounds on the stretch index of a graph G. First, the girth of G, which is the length of a minimum induced cycle $g(G)$, implies $\sigma_T(G) \geq g(G) - 1$ (cf. [5]). We call such lower bound of *girth lower bound*. For any edge e of G, let $l(e)$ denote the length of a shortest cycle minus 1 that contains e in G, where we assume that $l(e) = 1$ if e is not contained in any cycle. Another lower bound of $\sigma_T(G)$ is given by $max_{e \in E(G)} l(e)$. Indeed, consider any spanning tree T of G and any edge $e \in E(G)$. If $e \notin E(T)$, then the stretch index of G is at least $l(e)$. If $e \in E(T)$ and $l(e) > 1$, then there is an edge e' such that $T + e'$ has a cycle that contains e. Such a cycle must have a length of at least $l(e) + 1$. We call such lower bound of *smallest-e-cycle lower bound*.

Therefore, while generating a spanning tree of G if we have obtained $\sigma(T) = g(G) - 1$ or $\sigma(T) = \max_{\forall e \in E(G)}\{\ l(e)\}$, then we have obtained the index of G, and the algorithm ends (represented by $lower\_bound(G)$ in line 29). Otherwise, we continue generating spanning trees of G until we reach one of the lower bounds or all spanning trees of G are generated. Finally, we set as the index of G the smallest value obtained. Although there is an exponential number of spanning trees, the space complexity of Algorithm 1 has linear size, since at each step we store the input graph, the current tree and the smallest factor obtained.

The iterative algorithm uses integer vectors to indicate the next neighbor to be evaluated in the current iteration corresponding to the vertex v. The algorithm increments the next neighbor position ($next\_neighbor[v] \leftarrow next\_neighbor[v]+1$), after including the associated edge in the tree (line 21), and check if it causes a cycle. If it does not have a cycle, the current vertex v is incremented (line 33) to be analysed in next iteration. This process continues until checking all vertices. If the tree is not a spanning tree, the algorithm continues by selecting the next neighbor in the adjacent list of

Algorithm 1: findingAllTrees

```
   Input  : Connected graph G = (V, E), with V = {0, 1, ··· , |V| − 1}
   Output: Stretch index of G and an optimum tree t-spanner of G
 1 begin
 2     v = 0;
 3     index = |V|;
 4     tree = ' ';
 5     opt_tree = ' ';
 6     for i = 0, i < |V| do
 7         next_neighbor[i] = 0;
 8         last_neighbor[i] = −1;
 9     while v ≥ 0 do
10         if next_neighbor[v] == d_G(v) then
               // d_G(v) is the degree of v in G
11             next_neighbor[v] = 0;
12             v −= 1;
13             if v < 0 then
14                 break;
15             remove edge (v, last_neighbor[v]) ;
16             last_neighbor[i] = −1 ;
17         else
18             u = adj_list(v)[next_neighbor[v]]; // adj_list is an adjacency
                   list of V(G)
19             next_neighbor[v] += 1;
20             if tree contains edge (v, u) is false then
21                 tree = tree ∪ (v, u); // if tree does not contain the edge
                       (v, u), add (v, u) to tree
22                 last_neighbor[v] = u;
23                 if tree has a cycle is false then
24                     if tree has |V| − 1 edges is true then
25                         f = factor(tree) // computes the stretch factor
                               of the current spanning tree of G
26                         if f < index then
27                             index = f;
28                             opt_tree = tree;
29                             if index == lower_bound(G) then
30                                 break;
31                     else
32                         if v < |V| − 1 then
33                             v += 1;
34                         continue;
35                 remove edge (v, last_neighbor[v]) ;
36     return index, opt_tree
```

v (when condition in line 32 is false). Example 1 depicts an execution of Algorithm 1.

Example 1. *Consider the graph in Fig. 1 a), whose adjacency list is the following:*

$$
\begin{array}{lcl}
0 & \rightarrow & [1,4] \\
1 & \rightarrow & [0,5,6] \\
2 & \rightarrow & [3,4] \\
3 & \rightarrow & [2,4,7] \\
4 & \rightarrow & [0,2,3,7,8] \\
5 & \rightarrow & [1,7] \\
6 & \rightarrow & [1,8] \\
7 & \rightarrow & [3,4,5,8] \\
8 & \rightarrow & [4,6,7]
\end{array}
$$

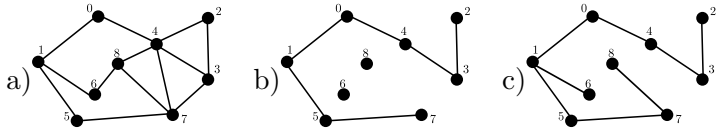

Figure 1: a) Graph G. b) Graph obtained by $(0,1),(1,5),(2,3),(3,4),(4,0),(5,7)$. c) Resulted tree after the generation of the first spanning tree of G.

Starting at $v = 0$, we consider the edge $(0,1)$. In each iteration, while there is an edge incident to v that does not belong to the actual generated tree, then such an edge is added to the tree (line 21). Next, we check whether a cycle was created and if it is a spanning tree of G (lines 23 and 24, respectively). Fig. 1b) depicts the graph obtained after the addition of the edge $(3,4)$ considering the adjacency list of $v = 3$. After that, we increment v ($v = 4$, line 33) and continue to the next neighbor of v, by adding the edge $(4,0)$. Following this procedure, when we reach $v = 7$ and the edge $(7,3)$ is added, then the algorithm detects a cycle, removes it and tries the next neighbor of v. Fig. 1c) depicts an obtained spanning tree. After that, we calculate its factor (line 25), remove the last added edge $(7,8)$ (line 35), and follow the algorithm to get a next spanning tree of G. Note that, from lines 10 to 16, when all edges incident to a vertex v is already analyzed in the current tree, the list next_neighbor *in position v turns back to be equal to 0 in order to all edges incident to v be analyzed with respect to the next vertex.*

Theorem 2.1. *If $\sigma_T(G) > g(G) - 1$, then all spanning trees of G are generated in Algorithm 1.*

Proof. Consider that $\sigma_T(G) > g(G) - 1$. Given the adjacency list of G, Algorithm 1 adds edges in increasing order up to obtain a spanning tree of

G. After that, the last added edge is removed, and then we continue by analyzing the next edge of the adjacency list. After we have substituted all last edges and checked if the resulted subgraph of G is a spanning tree of G, we remove the last two edges and proceed the strategy by increasing the second to last edge in the adjacency list of G, and restarting the analyze considering all pairs of the last two edges. Hence, we continue the procedure until all subsets of edges are obtained and analyzed, implying that all spanning trees are generated in Algorithm 1. □

2.2. Parallel strategy

All spanning trees are obtained by Algorithm 1 if the stretch index of the input graph is not equal to the girth lower bound. However, its time complexity is a function of the spanning trees number, implying in an exponential running time (function of the Cayley formula n^{n-2}, which is the number of trees with n vertices, *cf.* [21]). So, we can only run Algorithm 1 in an "acceptable time" for graphs with small size. As an attempt of decreasing such running time, we develop a parallel strategy to obtain all spanning trees of a graph, so that we are able to determine the stretch index of larger graphs (as discussed in Section 4.2.1).

The parallelism is executed at thread level. All threads have the input graph G and the edge vector of G. The efficiency of parallelism is limited by the number of *cores* (CPUs), since, when all cores are executing the threads, a core needs to finish a process before executing the next thread.

We develop and implement the following parallel strategy: *Edge-list-parallelism*. By analyzing this strategy, we assure that two threads do not generate repeated trees.

The algorithm begins with the first element of $edge_G$. We start each thread by referencing a different position of the edge vector. The edges of previous positions of the list of edge of G (called $edge_G$) are not considered in the current thread, thus each thread obtains distinct spanning trees with respect to the ones obtained by any other thread. When there is no spanning tree as a subset of the edges a thread, no value is returned. If a thread obtains a factor that achieves the girth lower bound of G, then the algorithm ends. If the index of G is different from $g(G) - 1$, then the threads execute all spanning trees. After all thread executions, the smallest value between the local indexes of each thread is the index of G.

Considering the graph of Example 1, $thread_1$ starts with $i = 0$, $edge_i = (0, 1)$, $thread_2$ starts with $i = 1$, $edge_i = (0, 4)$, and so on.

Note that the number of trees obtained per thread is not uniformly distributed. The thread starting at position 1 creates more trees than the

one starting at position 3. Furthermore, note that a thread, eventually, may not provide a tree.

To avoid unnecessary processing, we do the following preprocess before the execution of each thread: Consider $edge_G$ the edge vector of size m and any thread starts the algorithm in position i from the vector. This means that a thread only considers edges from $edges_i$ to $edges_m$. So, if a graph formed by all edges of this range is disconnected, it is impossible to obtain a tree, and then the thread aborts its execution.

3. Heuristics and exact algorithms for graph classes

Sections 3.1 and 3.2 present two simple greedy heuristics to obtain the stretch factors of a given graph G, which yield upper bounds of the stretch index of G. These spanning trees are obtained by the intuition that vertices of high degrees are adjacent in the tree to as many neighbors as possible of the original graph G.

Heuristic 1 generates trees in such a way that vertices chosen in each step tend to reach the most possible number of neighbors in the resulted tree. Heuristic 2 is similar to Heuristic 1, however, in each step, the vertices candidate to be chosen are adjacent to another one already in the tree. For this reason, the main difference between Heuristic 1 and 2 is that the first one is obtained connecting disjoint trees of a forest, and the second tree is obtained from a unique tree.

In Section 3.3, we determine graph classes for which one of the proposed heuristics provides not only an upper bound on the stretch index, but it is an exact algorithm to determine such a parameter.

3.1. Heuristic 1

We list the vertices of the graph G in decreasing order by their degrees. Thus, at each step, we add to the tree the first vertex of the list which is not in the tree yet and all its neighbors in G that are not in the tree either. We continue the process as long as there is a vertex which is not in the tree.

Example 2. *Consider the graph G of Example 1. Its vertices list sorted in decreasing order by degree is* $[4, 7, 1, 3, 0, 2, 5, 6]$. *In Fig. 2 we illustrate the execution of Heuristic 1 in order to create a spanning tree of G.*

3.2. Heuristic 2

Let u be the highest degree vertex of G. We create a tree T as follows: first, T is empty; then, we add to T the vertex u and all its neighbors; after

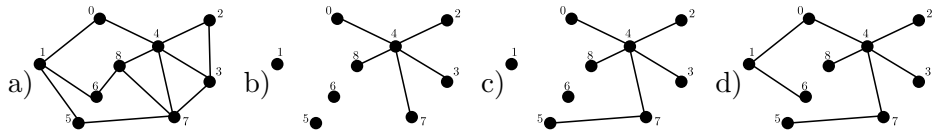

Figure 2: a) Graph G. b) Add vertex 4 and all its neighbors. c) Add vertex 7 and all its valid neighbors, note that $(7, 8)$ and $(7, 3)$ are not valid edges. d) Add vertex 1 and all its valid neighbors. That way we have the tree created.

that, we create a list L with all the leaves of T (which are all neighbors of u in G), we get a vertex v of L that has the greatest number of neighbors not in T yet, and then we add to T and L all valid neighbors of v. The vertex v is obtained by the function $\max_{v \in L}\{f(v)\}$, such that $f(v) = d_G(v) - A_{tree}(v)$, where $A_{tree}(v)$ is the current number of neighbors of v with respect to G that are vertices of T; after adding the valid vertices in T, we remove v from L and we update the function f to all vertices in L. We continue this process while there is a vertex in the list which is not in T.

Example 3. *Consider the graph G of Example 1. In the first step, since there is no vertex in the tree, the list is the same as in Heuristic* 1. $V = [4, 7, 1, 3, 0, 2, 5, 6]$. *Next, in Fig. 3 we illustrate the execution of Heuristic* 2.

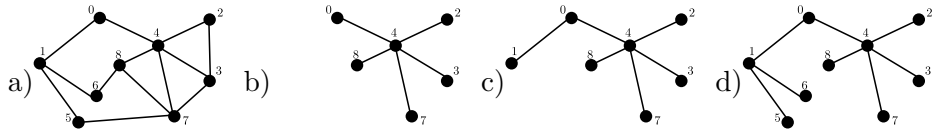

Figure 3: a) Graph G. b) Add vertex 4 and its neighbors. $L = [0, 2, 3, 7, 8]$. c) All neighbors of vertex 3 are in the tree, so $f(3) = 0$. Considering vertex 0, one neighbor is not in the tree, so $f(0) = 1$. Thus, the updated list is $[0, 7, 8, 2, 3]$. Add the valid neighbors of vertex 0 in the tree and in the list. Remove 0 from the list and the updated list is $[7, 8, 2, 3, 1]$. d) applying the function in the list, we have $[1, 7, 8, 2, 3]$. Add the valid neighbors of vertex 0 in the tree. Hence, we have the solution tree.

3.3. Exact algorithms for graph classes

In Sections 3.1 and 3.2 we have presented two simple heuristics, which are not only efficient, but for some graph classes they also return *optimum spanning trees*, i.e. spanning trees such that the factor is equal to the graph stretch index. For instance, for any graph that has a *universal vertex*, i.e. a vertex adjacent to all others, both heuristics return optimum spanning trees.

Although t-admissibility is NP-complete for $t \geq 4$ and even when we restrict to some graph classes such as bipartite graphs, chordal graphs or

planar graphs, and an open problem for $t = 3$, there are several other classes for which we are able to determine the stretch index, such as split graphs, cographs and p-cycle-power graphs [5, 6].

Let a *star* with $n + 1$ vertices be the complete bipartite graph $K_{1,n}$. A *v-centered star* is a star centered on v, that is a universal vertex.

Next, we prove that Heuristic 1 or 2 can be used in order to determine optimum spanning trees for split graphs, cographs and p-cycle-power graphs. The optimum spanning trees of these graphs can be obtained by running one of the previous heuristics after the following modification over the input graph: given a graph G, remove all *pendant vertices*, i.e., the vertices of degree 1. Let G' be the transformed graph; run Heuristic 1 (or Heuristic 2) in the transformed graph G'. Since the corresponding edges of pendant vertices must belong to any solution tree, then for any graph such removal does not change its stretch index. Moreover, this modification intends to avoid "false stars centers", i.e. vertices that could be chosen to be the centers of a star but it may be not a good choice. Since when a vertex of maximum degree in a graph have a large number of pendant as neighbors, its choice as a center may approximate distances uniquely of the pendant vertices.

A graph G is a *split graph* if its vertices can be partitioned into a stable set and a clique. In this case, $V(G)$ is partitioned into a clique X and a stable set Y.

Lemma 3.1. *[5] If G is a split graph, then $\sigma_T(G) \leq 3$. Moreover, $\sigma_T(G) = 2$ if and only if either: i) $d_G(y) = 1, \forall\ y \in Y$, or ii) $\exists\ x \in \bigcap_{y \in Y} N_G(y),\ x \in X$ such that $d_G(y) \geq 2$.*

Lemma 3.1 states that split graphs are 3-admissible and gives the characterization of the 2-admissible ones. A tree 3-spanner can be obtained by setting any vertex $x \in X$ to be x-centered star including each other vertex of X, and next, for each vertex $y \in Y$, choose an edge incident to y, arbitrarily, and make y a pendant in T. Since in the worst case a vertex that belongs to Y is adjacent in G to two leaves of $T \cap X$ and it is not adjacent to x, hence G is 3-admissible. Next, we prove that Heuristics 1 and 2 provide besides a tree 3-spanner of G, a tree 2-spanner for 2-admissible split graphs.

Theorem 3.1. *Let G be a split graph without pendant vertices. Hence, Heuristics 1 and 2 return an optimum spanning tree.*

Proof. Consider G with vertex set partitioned into a clique X and a stable set Y such that X has maximum size. If X is not maximum, then there is

a vertex v that belongs to Y that is adjacent to all X's vertices, and then we consider a vertex partition into the clique $X \cup \{v\}$ and the stable set $Y \setminus \{v\}$. Heuristics 1 and 2 start by a tree T rooted in a vertex of maximum degree, such vertex x belongs to X, because otherwise X is not maximum. Therefore, $X \cap T$ is a x-centered star. Moreover all neighbors of x belonging to Y are adjacent to x in T. Both heuristics continue: Heuristic 1 selects a second vertex with largest degree, which is another vertex of X, implying then a adding to T vertices of Y; Similarly, Heuristic 2 also selects a vertex of X, since the unique vertices of G that do not belong to T at this step are vertices of Y, which is a stable set. Heuristic 1 and 2 continue until all vertices of G belong to T, and any choice of the remaining vertices that is made produces in a tree 3-spanner, because $X \cap T$ is a star.

As a consequence of Lemma 3.1 and that G does not have pendant vertices, G is 2-admissible if and only if G has a vertex $x \in X$ such that x is adjacent to each vertex of Y, case ii) of Lemma 3.1, which implies that x is a universal vertex of G. Hence, if G has a universal vertex u, then u is the vertex of maximum degree of G, implying that u is settled by Heuristics 1 and 2 to be center of star in $T \cap G$. □

A *cograph* is a P_4-free graph. A trivial graph is a cograph, and any other can be obtained by disjoint union or join operations of cographs. Given graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, the union of G_1 and G_2 is the graph such that $V = V_1 \cup V_2$ and $E = E_1 \cup E_2$. The join of G_1 and G_2 is the graph such that $V = V_1 \cup V_2$ and $E = E_1 \cup E_2 \cup \{xy \mid x \in V_1, y \in V_2\}$. We can represent the union and join operations of a cograph by a tree representation, called *cotree* [18].

We construct a cotree of a cograph by the following strategy: Given a cograph G, create a leaf node for each vertex of G. If G is disconnected, create the edges between the root of the cotree and internal nodes, one per each connected components of G, and the root is labeled by 0, corresponding to the union operation of the connected components of G. If G is connected, consider its complement graph $\overline{G}$ (since $\overline{G}$ is a cograph and hence if G is connected, $\overline{G}$ is disconnected [18]), create edges between the root of the cotree and internal nodes, one per each connected components of $\overline{G}$, and the root is labeled by 1, corresponding to the join operation of the connected components of $\overline{G}$. After that we apply the above strategy to the new internal nodes until we get to the leaves of the cotree.

Lemma 3.2. *[5] If G is a cograph, then $\sigma_T(G) \leq 3$. Moreover, $\sigma_T(G) = 2$ if and only if G has a universal vertex.*

In [5], the stretch index for cographs was determined by observing struc-

tural properties on their cotrees. Since G must be connected, the label of the root of a cotree is 1, implying that any vertex of G represented as a leaf node of the root of a subtree is adjacent to all vertices of the roots of the other subtrees. A tree 3-spanner T of G is obtained as follows: let f be a leaf node of the root of a subtree, F_1. Since f is adjacent to all vertices of the roots of the other subtrees, set T as a f-centered star making f adjacent to each vertex of all subtrees distinct of F_1; let lf be an edge of the star just obtained. Since the vertex l in G is adjacent to all vertices of F_1, hence we add to T each edge corresponding to a neighbor of l in F_1, except to the edge lf.

Theorem 3.2. *If G is a cograph, then Heuristic 2 returns an optimum spanning tree.*

Proof. By Lemma 3.2, G is 2-admissible if and only if it has a universal vertex. Since a universal vertex has the highest degree of G and an optimum tree is a star, then Heuristic 2 returns an optimum spanning tree for 2-admissible cographs. For the case that G has $\sigma_T(G) = 3$, in order to obtain a tree 3-spanner T of G observing the cotree of G, any leaf node f of a subtree is chosen as f-centered star of T and all neighbors of f are adjacent to it in T. Hence, Heuristic 2 chooses f as the vertex of G with highest degree and adds to a solution tree T' all its neighbors to be adjacent to it in T'. The next step concerns on choosing a vertex l such that fl is an edge of T, and since any node that belongs to a distinct subtree that f belongs is adjacent to all remaining vertices that do not belong to T', then l is such a chosen node. It implies in a tree 3-spanner of G given by T' after adding $N(l) \setminus N[f]$. □

One may ask if Heuristic 1 also returns an optimum spanning tree for cographs. In fact, if G is cograph and 2-admissible, then Heuristic 1 returns a spanning star. However, if $\sigma_T(G) = 3$, Heuristic 1 eventually returns a tree 4-spanner of G. It can be observed by assuming that T is the star obtained at the first step. If Heuristic 1 considers at the second step a vertex l that does not belong to T, l is adjacent to l' and f'' in T, such that l' does not belong to T and f'' belongs to T and is adjacent to f. Hence, $d_T(l', f') = 4$, for $f' \in N(f)$, by the path l', l, f'', f, f'.

A *p-cycle-power graph*, C_n^p, is obtained from a C_n by adding edges between all vertices with distance at most p in C_n. Such a cycle C_n is denominated *external cycle*.

Lemma 3.3. *[6] For any p-cycle-power graph C_n^p, $p \geq \lfloor \frac{n}{2} \rfloor$, then $\sigma_T(C_n^p) = 2$. Otherwise, if $p < \lfloor \frac{n}{2} \rfloor$, then: i) if $n \equiv x \pmod{p}$, for $x \in \{0, 1\}$, then $\sigma_T(C_n^p) = \lfloor \frac{n}{p} \rfloor$; ii) if $n \not\equiv x \pmod{p}$, for $x \in \{0, 1\}$, then $\sigma_T(C_n^p) = \lceil \frac{n}{p} \rceil$.*

Lemma 3.3 states the stretch index of p-cycle-power graphs, and an optimum spanning tree T of a p-cycle-power graph G is determined as follows: choose an arbitrary vertex v; add $N[v]$ to T; obtain a vertex at distance p to v in the external cycle C_n of G; add $N(v) \setminus V(T)$ to T; continue this strategy until $V(T) = V(G)$.

Theorem 3.3. *If G is a p-cycle-power graph, then Heuristic 2 returns an optimum spanning tree.*

Proof. Note that G is regular, hence Heuristic 2 chooses a vertex v arbitrarily in the first step to obtain a tree T. In the second step, the heuristic takes a vertex already in T with the largest number of neighbors not in T, hence such a vertex is the one at distance p to v in the external cycle C_n, and Heuristic 2 continues as the previous strategy in order to obtain an spanning tree of G, as consequence of Lemma 3.3, Heuristic 2 returns an optimum spanning tree of G. □

Although the proof of Theorem 3.3 establishes that an optimum spanning tree of p-cycle-power graphs can be constructed in a greedy way, Heuristic 1 may not return an optimum spanning tree. For instance, consider C_7^2, where the associated external cycle is $C_7 = v_1, v_2, \ldots, v_7, v_1$. W.l.o.g., in the first step, Heuristic 1 takes v_1 and adds to T vertices v_1, v_2, v_3, v_7, v_6. Now, in the second step, since G is regular, any other one can be chosen. Hence, choosing v_4 (or v_5), it implies in a tree 4-spanner of G. However, by Lemma 3.3, and also by Theorem 3.3, $\sigma_T(C_7^2) = 3$.

4. Computational experiments

We implement: brute force algorithms, the sequential (Section 2.1) and the edge-list-parallelism (Section 2.2); Heuristic 1 (Section 3.1); and Heuristic 2 (Section 3.2).

We also generated graphs of four graph classes, which are Barabási-Albert graphs, Erdös-Rényi graphs, Watts-Strogatz graphs and Bipartite graphs. In Section 4.1, we define the four graph classes and present the parameters used for the graph generations; Section 4.2.1 compares the performance between the sequential, the parallel brute force algorithms, the speedups on the heuristic strategies with respect to the parallel brute-force

algorithm for the Erdös-Rényi graphs by considering $G(n, p)$ graphs from 10 up to 20 vertices. We also present speedups on Heuristic 2 with respect to Heuristic 1 for the four graph classes stated above considering graphs up to 1000 vertices. In Section 4.2.2, we compare the results returned by the heuristics with the smallest-e-cycle lower bound.

4.1. Graphs generator

Barabási-Albert graphs (BA-graphs) are random graphs that have the construction process as follows: initially the graph G is a small clique, for instance with 3 vertices; at each step, a new vertex v of degree k is added in such a way that the neighbors of v are randomly chosen, with probability proportional of the degree of the vertices in $G \setminus v$. BA-graphs model several real applications, like webgraph or social network. The main reason is that new vertices are added to the graph following the preferential attachment process, i.e. new added vertices tend to be adjacent to existing vertices with high degree [2].

Erdös-Rényi graphs (ER-graphs), $G(n, p)$, have n vertices and for each pair of vertices, there is probability p of an edge existence. ER-graphs are well studied in the literature, and there are several known properties that hold for almost all ER-graphs. For instance, given a $G(n, p)$ graph for fixed $p > 0$, it is known that $G(n, p)$ is connected for almost all graphs [3, 12].

Watts-Strogatz graphs (WS-graphs) are graphs with n vertices, which are constructed by first obtaining a cycle graph C_n, and each pair of vertices at a distance at most k in C_n there is a probability p of the existence of an edge. WS-graphs are random graphs inspired on the Small-world experiment developed by Milgram related to the "six degrees of separation" model [13], which says that the relationship between people is a network characterized by paths of short lengths. WS-graphs catch the property of having small short diameter, even considering sparse graphs [20]. Note graph k-cycle-power graphs are particular cases of WS-graphs where the probability p of the existence of an edge of each pair of vertices at a distance at most k in C_n is equal to 1.

Random bipartite graphs are constructed by setting the lengths of the two parts U and V of the vertex partition, and the probability p of the existence of an edge between a vertex of U and a vertex of V. Bipartite graphs are well studied graphs in many problems, in particular, for the t-admissibility problem, it is known that it is NP-complete, for $t \geq 4$ [4].

Table 1 presents the setup parameters to generate all graph classes using NetworkX package on python.

Table 1: The experiments were developed for 10 graphs for each value of $n(G) \in \{10, 20, 30, \cdots, 90, 100, 200, 300, \cdots, 900, 1000\}$.

Graph class	n	m	p	k	**Description of parameters**
BA-graphs	$\|n(G)\|$	$\|n(G)\|0.40$	—	—	n: number of nodes m: degree of each added node
ER-graphs	$\|n(G)\|$	—	0.50	—	n: number of nodes p: probability of an edge
WS-graphs	$\|n(G)\|$	—	0.30	$\|n(G)\|0.40$	n: number of nodes p: probability of an edge with nodes at a distance at most k k: distance of two nodes in C_n
Bipartite graphs	$\frac{\|n(G)\|}{2}$	$\frac{\|n(G)\|}{2}$	0.30	—	n: number of nodes in U m: number of nodes in V p: probability of an edge between a node in U and V

4.2. Comparison between the algorithms

We compare the time between the sequential and the parallel brute-force algorithms, and the solutions given by the heuristics compared to the parallel brute-force one. The comparison of the running time was done by C++ *ctime* library. The test machine has the following configuration: Intel(R) Xeon(R) CPU E5-2640 v3 2.60GHz 32 cores; 128GB of memory.

4.2.1. Performance Analysis

Figure 4 shows the performance results for parallel version using the sequential version as baseline. Overall, the expressive speedups are obtained when running on 8 threads achieving until 2.53 speedup for a graph with $n = 19$. Considering the scenario of using more than 8 threads, the speedups have small variations compared against 8 threads due to the limitation of parallelism of each graph (number of vertices, edges and maximum degree). Thus, we may have better speedup results with more than 8 threads for graphs with larger size. Graphs with smaller size have slowdowns due to thread synchronization overhead being greater than algorithm processing time.

Table 2 presents the performance results for parallel, Heuristic 1 and Heuristic 2 algorithms. The results of the heuristics achieved higher speedup since they are greedy strategies that find only one tree in polynomial time. Heuristic 2 obtained better performance in all graphs, resulting up to 47.44 speedup for a graph with $n = 19$ when compared against Heuristic 1.

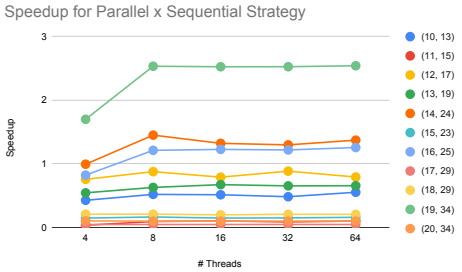

Figure 4: Speedup for parallel strategy when varying the number of threads. Each line corresponds to a graph from Table 2 represented by the pair $(n,\ m)$ with n vertices and m edges.

Table 2: Speedup of parallel algorithm, Heuristic 1 and Heuristic 2 for 11 graphs $G(n,p)$ with m edges, n ranging from 10 to 20 and $p = 0.3$. The baseline for speedups of parallel and heuristics algorithms are the sequential version.

n	**m**	**Speedup Parallel (8 threads)**	**Speedup Heuristic 1**	**Speedup Heuristic 2**
10	13	0.52	99.0	1,349.45
11	15	0.09	1.04	16.3
12	17	0.87	908.13	13,819.96
13	19	0.63	2,619.41	61,531.71
14	24	1.45	34,447.55	848,768.84
15	23	0.161	1.63	42.67
16	25	1.21	23,553.0	692,796.99
17	29	0.05	8,057.82	254,858.68
18	29	0.21	4,155.89	191,137.47
19	34	2.53	9,075,670.64	430,587,532.5
20	34	0.1	108,520.46	4,513,437.36

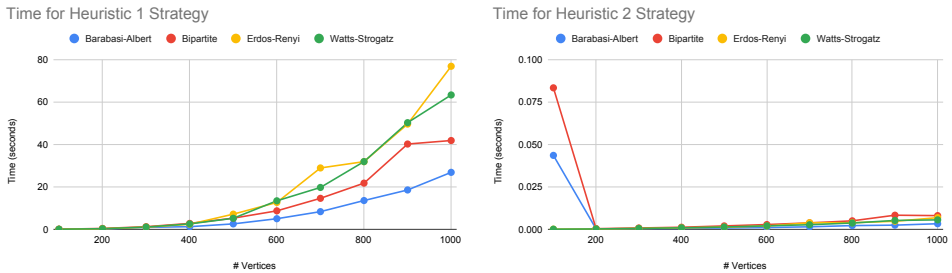

Figure 5: (a) Execution time of Heuristic 1. (b) Execution time of Heuristic 2. The execution time of heuristics strategies when varying the number of vertices for different graph classes. Each line corresponds to a graph class.

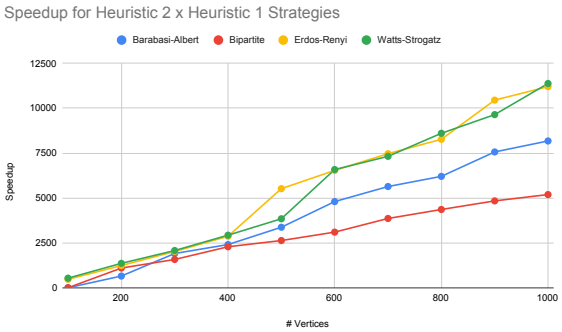

Figure 6: Speedup of Heuristic 2 compared against Heuristic 1 when varying the number of vertices for the four different graph classes. Each line corresponds to one of those graph classes.

Figures 5(a) and (b) depict running time (in seconds) of Heuristics 1 and 2 considering the graph classes discussed in Section 4.1, respectively. Heuristic 2 concludes the executions much faster than Heuristic 1, as it can be viewed by the speedup for Heuristic 2 compared against Heuristic 1 in Figure 6. The main reason is that Heuristic 1 checks at each time the degrees of all vertices in the graph, differently of Heuristic 2, that uses a local search between the vertices of the current spanning tree.

4.2.2. Quality Analysis

In Section 3.3, we have presented a modification on the proposed heuristics by removing all pendant vertices of the input graphs. Such removal does not affect the value of the stretch index we aim to obtain and so we can determine optimum spanning trees for the graph classes presented.

Next, we present the results of the random graphs generated by comparing the heuristics of Sections 3.1 and 3.2 with the implemented smallest-e-cycle lower bound, proposed in Section 2.1 (i.e. the smallest-e-cycle lower bound is $\max_{\forall e \in E(G)}\{\ l(e)\}$, where $l(e)$ is the length of the smallest cycle minus 1 containing e). Although we preprocess the input by removing pendant vertices (as discussed in Section 4.1) and so, Heuristic 1 and 2 are applied to graphs without pendant vertices, when considering the random BA-graphs and WS-graphs, we have that the input graphs do not have pendant vertices, by construction. When considering random $G(n,p)$ graphs (also regarding random bipartite graphs), the removal of pendant vertices does not substantially change the input graph. Because in almost all random graphs, the expected number of graphs with pendant vertices is given by the following formula: $n(n-1)p(1-p)^{n-2}$ [10]. A wide study on pendant vertices in random graphs was developed by Palka [14, 15].

Table 3: Analysis of Heuristic 1 for the presented graph classes. In each cell, the bottom (the top) value is the average (the standard deviation) between the difference of the returned value of H1 and the smallest-e-cycle lower bound. For each graph class, there were computed 10 graphs for each n vertices. $Av(m)$ is the average of the number of edges of all graphs with n vertices. The lower bounds computed of all graphs with $n \in \{100, 200, \cdots, 1000\}$ are equal to 2 for three first graph classes, and equal to 3 for the last one (bipartite graphs).

Class \ n / $Av(m)$	100 1.9k	200 7.9k	300 17.1k	400 30.4k	500 47.8k	600 68.8k	700 93.7k	800 122.2k	900 154.9k	1000 191.2k
Barabási-Albert	0.30 1.90	0.00 2.00	0.00 2.00	0.00 2.00	0.00 2.00	0.00 2.00	0.00 2.00	0.00 2.00	0.00 2.00	0.00 2.00
Erdös-Rényi	0.70 2.90	0.70 3.10	0.67 3.50	0.75 3.80	0.64 4.30	1.19 4.00	0.89 3.8	0.67 3.50	0.75 4.20	0.78 4.30
Watts-Strogatz	0.80 3.60	0.89 5.00	0.63 5.00	0.60 5.20	0.75 4.80	0.53 4.90	0.80 5.40	1.11 5.40	1.04 5.10	0.94 4.90
Bipartite	1.33 5.20	1.67 6.20	1.26 6.00	1.10 6.20	1.60 7.20	1.28 7.40	1.83 7.20	1.74 7.60	1.20 6.40	1.61 7.00

Tables 3 and 4 present the average (and the standard deviation) of the difference between the returned value of Heuristic 1 and Heuristic 2, respectively, and the smallest-e-cycle lower bounds. The computational experiments were made for BA-graphs, ER-graphs, WG-graphs and bipartite graphs. For each graph class, there were computed 10 graphs for each n, with $n \in \{100, 200, \cdots, 1000\}$.

For graphs belonging to the four random graph classes with $n \in \{10, 20, ..., 90\}$, the average between the difference of the returned value of Heuristic 1 and the smallest-e-cycle lower bound is 2.36, with standard deviation of 1.49. Whereas, the average between the difference of the returned value of Heuristic 2 and the smallest-e-cycle lower bound is 1.94, with standard deviation of 1.25.

Note that even comparing the heuristics with a natural lower bound (smallest-e-cycle lower bound), which is eventually much less than the exact stretch index of a graph, the heuristics perform good results, since the tested graphs have large number of vertices, have average number of edges close to the half of the number of all possible edges, and belong to distinct random graph classes.

Table 4: Analysis of Heuristic 2 for the presented graph classes. In each cell, the bottom (the top) value is the average (the standard deviation) between the difference of the returned value of H1 and the smallest-e-cycle lower bound. For each graph class, there were computed 10 graphs for each n vertices. $Av(m)$ is the average of the number of edges of all graphs with n vertices. The lower bounds computed of all graphs with $n \in \{100, 200, \cdots, 1000\}$ are equal to 2 for three first graph classes, and equal to 3 for the last one (bipartite graphs). The graphs are the same as Table 3.

Class \ n / $Av(m)$	100 1.9k	200 7.9k	300 17.1k	400 30.4k	500 47.8k	600 68.8k	700 93.7k	800 122.2k	900 154.9k	1000 191.2k
Barabási-Albert	0.46 1.30	0.30 1.90	0.00 2.00	0.00 2.00	0.00 2.00	0.00 2.00	0.30 2.10	0.30 2.10	0.40 2.20	0.49 2.40
Erdös-Rényi	0.00 3.00	0.77 3.00	0.50 3.50	0.49 3.40	0.87 3.20	0.66 3.40	0.46 3.70	0.54 4.10	0.40 4.20	0.77 4.00
Watts-Strogatz	0.60 3.20	0.92 3.40	0.54 3.90	0.54 4.10	0.54 3.90	0.49 4.60	0.92 4.50	0.54 4.10	0.46 4.70	0.49 4.40
Bipartite	0.92 4.60	1.08 5.80	1.74 5.60	1.20 5.60	1.50 5.60	1.50 6.40	0.90 6.00	1.40 5.80	1.34 5.00	1.56 7.40

5. Concluding remarks

We develop, implement and compare exact (sequential and parallel) and heuristic algorithms for t-admissibility (all implemented algorithms are available in [8]). Previous works presented computational results on the edge-weighted version of the t-admissibility problem. In [1], the results were obtained by using a time limit of 600s per instance, and in [17], the results were obtained by just comparing the proposed heuristics. Hence, studies regarding parallelism approaches are very important in order to have better practical evaluation.

Next steps are to implement solutions proposed in Section 2.2 to do not generate repeated trees and to have a uniform distribution on the number of spanning trees per thread, and to compare these implementations with the sequential and with the edge-list-parallelism; to find other graph classes for which at least one of the heuristics gives the exact solution; to implement and analyze other heuristics trying to provide some smart criteria for the choices of vertices. Tables 3 and 4 suggest that H2 produces better solutions than the ones returned by H1. However, there are examples for which H1 is better than H2, as the graph depicted in Fig. 7.

One may ask if the brute-force strategy presented could be optimized by not considering isomorphic spanning trees. Although there are isomor-

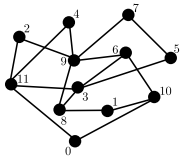

Figure 7: Graph G whose $\sigma_T(G) = 4$. H1 returns a tree 4-spanner, but H2 returns a tree 5-spanner.

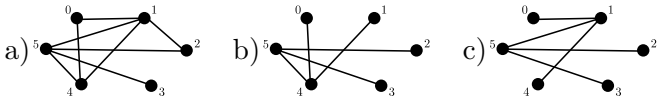

Figure 8: a) Graph G. b) Spanning tree T of G with factor equal to 3, by $d_T(1,2) = 3$. c) Spanning tree of G isomorphic to T with factor equal to 2, since vertex 0 and 4 are not adjacent to 2 or 3.

phic trees that have same factor, being isomorphic trees is not a sufficient condition of having the same factor. An example can be seen in Fig. 8. Hence, another open problem is to characterize isomorphic spanning trees of a graph G with the same factor.

In Section 2.2, we have developed a parallel strategy executed at thread level, and as discussed, the number of trees obtained per thread is not uniformly distributed. A way to obtain a uniform distribution on the number of trees per thread can be further developed by creating the parallelism based on the edges of a cycle C from an input graph, as described below.

We create $|C|$ spanning subgraphs of G, in such a way that each subgraph does not have an edge of C. Thus, after that, perform the edge-list-parallelism in each subgraph. Note that in here we may have two threads generating a same tree. In fact, let e and e' be two edges in a cycle so that one thread is responsible for trees obtained by removing e, and another thread is responsible for trees obtained by removing e'. Since in both threads eventually both edges do not exist in a spanning tree, we can have equal trees in the two threads.

A possible way to prevent repeated trees and to have a distribution across threads more uniform can be, after choosing an induced cycle C of G, remove i edges, for $i = 1, \ldots, |C|$, and consider the remaining edges of C as mandatory. Now, perform the edge-list-parallelism in each obtained graph. Considering that, by construction, all trees generated are distinct, because i edges of C belong to a single thread.

Acknowledgments

This work was partially supported by Brazilian agency CAPES (Finance Code 001) and by FAPERJ. The authors thank the Computer Architecture and Microelectronics Laboratory, LAM-PESC/COPPE/UFRJ, for making computers available for the computational experiments.

References

[1] E. Álvarez-Miranda and M. Sinnl. Mixed-integer programming approaches for the tree t*-spanner problem. *Optimization Letters*, pages 1–17, 2018.

[2] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[3] B. Bollobás and B. Béla. *Random graphs*. Number 73. Cambridge university press, 2001.

[4] L. Cai and D. Corneil. Tree spanners. *SIAM J. Discrete Math.*, 8(3):359–387, 1995.

[5] F. Couto and L. Cunha. Tree t-spanners of a graph: Minimizing maximum distances efficiently. In *Lecture Notes in Computer Science*, pages 46–61. Springer, 2018.

[6] F. Couto and L. Cunha. Hardness and efficiency on minimizing maximum distances for graphs with few P4's and (k,l)-graphs. In *Electron. Notes Theor. Comput. Sci.*, volume 346, pages 355–367, 2019.

[7] F. Couto and L. Cunha. Hardness and efficiency on minimizing maximum distances in spanning trees. *Theor. Comp. Sci.*, 2020.

[8] F. Couto, L. Cunha, D. Juventude, and L. Santiago. Implemented algorithms and computational analysis. `https://github.com/leandro-santiago/spanner_tree_generator`, 2021.

[9] F. Couto, L. Cunha, and D. Posner. Edge tree spanners. In *18th Cologne-Twente Workshop on Graphs and Combinatorial Optimization*, volume AIRO Springer series, pages 1–12, 2020.

[10] M. R. Dale. *Applying graph theory in ecological research*. Cambridge University Press, 2017.

[11] Y. Emek and D. Peleg. Approximating minimum max-stretch spanning trees on unweighted graphs. In *15th ACM-SIAM Symposium on Discrete Algorithms*, SODA, pages 261–270, 2004.

[12] P. Erdos, A. Rényi, et al. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[13] S. Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.

[14] Z. Palka. On pendant vertices in random graphs. In *Colloquium Mathematicum*, volume 45, pages 159–167. Institute of Mathematics Polish Academy of Sciences, 1981.

[15] Z. Palka. The distribution of degrees in random graphs. In *Graph Theory*, pages 161–169. Springer, 1983.

[16] A. Saha, M. Pal, and T. K. Pal. An optimal parallel algorithm to construct a tree 3-spanner on interval graphs. *International Journal of Computer Mathematics*, 82(3):259–274, 2005.

[17] K. Singh and S. Sundar. Artifical bee colony algorithm using problem-specific neighborhood strategies for the tree t-spanner problem. *Applied Soft Computing*, 62:110–118, 2018.

[18] J. P. Spinrad. *Efficient Graph Representations.: The Fields Institute for Research in Mathematical Sciences.*, volume 19. American Mathematical Soc., 2003.

[19] S. Sundar. A steady-state genetic algorithm for the tree t-spanner problem. In *Soft Computing: Theories and Applications*, pages 387–398. Springer, 2019.

[20] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *nature*, 393(6684):440–442, 1998.

[21] D. B. West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 1996.