# Three Is Enough for Steiner Trees

**Emmanuel Arrighi**
University of Bergen, Norway

**Mateus de Oliveira Oliveira**
University of Bergen, Norway

## Abstract

In the Steiner tree problem, the input consists of an edge-weighted graph $G$ together with a set $S$ of terminal vertices. The goal is to find a minimum weight tree in $G$ that spans all terminals. This fundamental NP-hard problem has direct applications in many subfields of combinatorial optimization, such as planning, scheduling, etc. In this work we introduce a new heuristic for the Steiner tree problem, based on a simple routine for improving the cost of sub-optimal Steiner trees: first, the sub-optimal tree is split into three connected components, and then these components are reconnected by using an algorithm that computes an optimal Steiner tree with 3-terminals (the roots of the three components). We have implemented our heuristic into a solver and compared it with several state-of-the-art solvers on well-known data sets. Our solver performs very well across all the data sets, and outperforms most of the other benchmarked solvers on very large graphs, which have been either obtained from real-world applications or from randomly generated data sets.

**2012 ACM Subject Classification** Theory of computation → Theory of randomized search heuristics

**Keywords and phrases** Steiner Tree, Heuristics, 3TST

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.5

**Supplementary Material** *Software*: `https://github.com/AutoProving/3TST`
archived at `swh:1:dir:740b703cb9dceee55ef1571f49d1b58701a2082f`

**Funding** *Emmanuel Arrighi*: Research Council of Norway (Grant no. 274526).
*Mateus de Oliveira Oliveira*: Research Council of Norway (Grant no. 288761) and Sigma2 Network (NN9535K).

## 1 Introduction

In the *Steiner tree* problem, we are given an undirected graph $G$ whose edges are weighted with non-negative values, and a subset of vertices $S$, whose elements are called *terminals*. The goal is to find a minimum-weight tree in $G$ whose nodes span all terminal in $S$. This is a fundamental NP hard problem [14], which has been studied since the seventies [11] and which has found applications in several fields of research such as planning [16], social networks [17], sensor networks [18], community detection [7], VLSI circuit design [13], as well as in numerous applications in industry [6].

Since Steiner tree is an NP-hard problem, most research surrounding this problem has been devoted both to the task of developing heuristics that work reasonably well in practice, and to the task of developing approximation algorithms that provide approximation guarantees within polynomial time. In particular, a short list of heuristic paradigms that have been used to attack the Steiner-tree problem include simulated annealing [19], genetic algorithms [5], logic programming [20] and constraint solving [8]. On the other hand, when it comes to approximation algorithms, the approximation ratio guarantee achievable by algorithms running in polynomial time was gradually improved from 2 [27] to 1.39 [4] in a span of two and a half decades[27, 29, 1, 30, 21, 15, 12, 24, 25, 4]. It is worth noting that unless $P = NP$, the Steiner tree problem in general graphs cannot be approximated within a factor of $1 + \epsilon$ for sufficiently small $\epsilon > 0$ [2].

In this work, we introduce a new heuristic for the Steiner tree problem and show that on large graphs, it outperforms several state of the art algorithms. Our heuristic has two main components. First, we devise a method that can be used to quickly compute a good initial Steiner tree. The second component is based on an improvement procedure that takes a Steiner tree as input and tries to output a lighter Steiner tree. Essentially, this procedure is executed until a specific time limit is up. It is worth noting that our improvement strategy is similar in spirit to an improvement procedure used in a celebrated approximation algorithm due to Robins and Zelikovsky [25].

This improvement procedure can be explained in two high-level steps. First, given a (potentially suboptimal) Steiner tree $T_0$, one appropriately split it into three subtrees $T_1$, $T_2$ and $T_3$ such that, all terminals are contained in $T_1 \cup T_2 \cup T_3$. Then those three subtrees are reconnected together by solving an instance of the Steiner tree problem with three terminals. This gives a new Steiner tree $T_0'$. As the Steiner tree problem with three terminals can be solve exactly and efficiently, the weight of $T_0'$ is at most the weigh t of $T_0$.

We observe that there are two crucial differences between the optimization procedure used in our heuristic and the one used in the algorithm of [25]. The first is that their algorithm is run in a complete graph where for each two vertices $v$ and $u$, the weight of the edge $\{v, u\}$ is the weight of the shortest path between $v$ and $u$ in the original graph, while our algorithm is run without the need to compute shortest paths between all possible pairs of vertices. The second difference is that in Robins and Zelikovsky's algorithm, the split is chosen to be the optimal one, while in our algorithm we replace optimality by a greedy selection strategy. Building the complete graph and looking for the optimum splitting is costly and cannot be done on large graphs. Therefore, the algorithm of [25] cannot handle large real world instances. By doing something that does not need such large structures our approach can handle large instances. As a consequence, our optimization procedure performs especially well on large graphs. We also note that this optimization procedure can also be used to improve the weight of sub-optimal Steiner trees output by other solvers.

To validate our new heuristic, we implement a solver in C++ and benchmark it against several state of the art solvers for the Steiner tree problem on well known data sets. These solvers implement several paradigms, such as genetic algorithms, linear programming algorithms, local search algorithms as well as algorithms with approximation guarantees. The data sets were obtained from a variety of sources, such as established real-world benchmarks for the Steiner tree problem, data sets of common use in the field of road networks, and a synthetic data set where instances are generated at random. Our solver obtained very good results in most data sets. In particular our solver was able to obtain solutions that are on par with those obtained by solvers that employed large scale mixed-linear programming suites such as SCIP [10]. Our solver was also able to handle very large instances, with millions of vertices and edges, while most of the solvers failed in these instances. A detailed exposition of these results can be found in Section 4.

## 2 Preliminaries

In this section, we set notation for basic graph-theoretic concepts used in the description of our algorithm. We let $\mathbb{N}$ denote the set of *natural numbers*. For a finite set $V$, we let $\mathcal{P}(V, 2) = \{\{u, v\} \ : \ u, v \in V, u \neq v\}$ be the set of unordered pairs of elements from $V$.

An *undirected graph* is a pair $G = (V, E)$ where $V$ is a set of *vertices* and $E \subseteq \mathcal{P}(V, 2)$ is a set of *undirected edges*. We may write $V(G)$ to denote the vertex-set of $G$ and $E(G)$ to denote the edge-set of $G$. An *edge-weighted* graph is a graph $G = (V, E)$ together with a *cost function* $\text{cost} : E \to \mathbb{N}$. We let $\text{cost}(G)$ be the sum of the costs of all edges in $G$.

We say that a graph $H$ is a *subgraph* of a graph $G$ if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. For each subset $X \subseteq V(G)$, the subgraph of $G$ induced by $X$ is the graph $G[X]$ with vertex set $X$ and edge set $E(G) \cap \mathcal{P}(X, 2)$.

A *walk* in a graph $G$ is a sequence of vertices $v_1, \ldots, v_k$ such that for each $i$ in $\{1, \ldots, k-1\}$, $\{v_i, v_{i+1}\} \in E(G)$. A *path* in $G$ is a walk in which all vertices are distinct. We let $\text{dist}(v, v')$ be the minimum number of edges in a path between $v$ and $v'$. We say that $G$ is *connected* if for each two vertices $v_1$ and $v_2$ there is a path between $v_1$ and $v_2$. A *cycle* is a walk $v_1, \ldots, v_k$ such that $v_1 = v_k$ and $v_i \neq v_j$ for $i, j \leq k$ and $i \neq j$. A graph is *acyclic* if it contains no cycle.

A *tree* is a connected acyclic graph $T$. A *rooted tree* is a tree $T$ together with a distinguished vertex $r$. If $T$ is a rooted tree with root $r$, and $v \in V(T)$ is such that $r \neq v$, then the parent of $v$ is the unique neighbour $v'$ of $v$ such that $\text{dist}(r, v') < \text{dist}(r, v)$. Note that the root $r$ does not have a parent. Each neighbour $v'$ of $v$ with $\text{dist}(r, v') > \text{dist}(r, v)$ is called a *child* of $v$. A *leaf* of $T$ is a vertex with no child. A *descendant* of a vertex $v$ is a vertex $v'$ such that any path between $r$ and $v'$ contains $v$. We consider $v$ to be a descendant of itself. The *subtree of $T$ rooted at $v$* is the subgraph of $T$ induced by the set of descendants of $v$.

Given a graph $G$, a spanning tree for $G$ is a tree $T$ such that $T$ is a subgraph of $G$ and $V(T) = V(G)$. Given a connected edge-weighted graph $G$, and a vertex $v \in v(G)$, a *shortest-path tree* for $G$ rooted at $v$ is an edge-weighted spanning tree $T$ of $G$ rooted at $v$ such that for each vertex $u \in V(G)$, the distance between $v$ and $u$ in $G$ is equal to the distance between $v$ and $u$ in $T$.

Let $G$ be an undirected edge-weighted graph and let $S \subseteq V(G)$ be a subset of vertices of $G$ whose elements are called *terminals*. A *Steiner tree* in $G$ is a subgraph $T$ of $G$ such that $T$ is a tree and $S \subseteq V(T)$. We note that $T$ may contain non-terminal vertices. We call the vertices in $V(T) \backslash S$, *Steiner points*. The cost of a tree, $\text{cost}(T)$, is the sum of the costs of its edges.

Let $G$ be a graph and $H$ be a connected subgraph of $G$. The *contraction* of $H$ in $G$, written $G/H$, is the graph obtained from $G$ by first deleting all vertices of $H$, then by adding a new vertex $v_H$, and finally by connecting $v_H$ to a vertex $u \in V(G) \backslash V(H)$ in $G/H$ if and only if there is an edge between $u$ and some vertex from $V(H)$ in $G$. The weight of an edge between $v_H$ and $u$ is the minimum weight of an edge connecting a vertex of $H$ to $u$. We note that in practice, since our graphs are weighted, contraction of a subgraph $H$ will be simulated by simply setting the weights of the edges of $H$ to 0, and therefore, the topology of the original graph remains unchanged.

## 3 Our Heuristics

In this section, we describe the main components of the heuristic used in our Steiner tree solver. There are three main components. A pre-processing component, which simplifies the input graph, a greedy procedure that constructs an initial Steiner tree, an optimization function that takes a given Steiner tree as input and outputs another Steiner tree that is at least as light as the original one. This optimization procedure is then repeated until it stabilizes, or until the time is up. Once the solution can not be improved, our solver starts again with a new starting Steiner tree. It repeats this procedure until it receives a timeout signal. These three components are described in more details below.

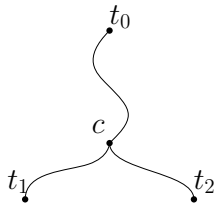


**Figure 1** A Steiner tree with three terminals $t_0$, $t_1$ and $t_2$ is a union of shortest paths between a center vertex $c$ and $t_0$, $t_1$ and $t_2$ respectively.

## 3.1 Preprocessing

During the preprocessing step, we modify the input graph by applying two standard rules [28, 22] with the goal of eliminating redundancies. Once a solution is obtained in the modified graph, this solution can be easily converted into a solution to the original graph. The two preprocessing rules we apply are the following.

1. The first rule removes non-terminal vertices of degree 1 from the graph. These vertices are redundant because if a Steiner tree contains such a vertex, then one can safely delete it from the tree and still obtain a valid Steiner tree.
2. The second rule eliminates non-terminal vertices of degree 2. More precisely, let $u$ be a non-terminal vertex of degree 2 connected to vertices $v_1$ and $v_2$ by edges $e_1 = \{u, v_1\}$ and $e_2 = \{u, v_2\}$ respectively. Then we delete the vertex $u$ and the edges $e_1$ and $e_2$ from the graph. If the graph has an edge of cost $c$ connecting $v_1$ and $v_2$, then we update the cost of this edge to $\min(c, \text{cost}(e_1) + \text{cost}(e_2))$. Otherwise, we just add a new edge $e = \{v_1, v_2\}$ of cost $\text{cost}(e_1) + \text{cost}(e_2)$ to the graph. This rule is repeated until no vertex of degree 2 is left.

This preprocessing step can be done in time $O(n)$ where $n$ is the number of vertices. Note that if a solution to the modified graph contains an edge $e = \{u, v\}$ that is not present in the original graph, then one can obtain a solution to the original graph by replacing each such edge $e$ by a path between $u$ and $v$ in which all internal vertices have degree 2.

## 3.2 Minimum Steiner trees with 2 or 3 terminals

A fact that we will use often both in the construction of an initial Steiner tree and in our optimization procedure is the fact that Steiner trees with two or three terminals can be computed very quickly by using elementary algorithms. Indeed, a Steiner tree with two terminal vertices $t_0$ and $t_1$ is simply a shortest path between these two vertices. On the other hand, it can be shown that if $T$ is a Steiner tree with 3 terminals $\{t_0, t_1, t_2\}$ then there is a *center vertex* $c$ such that $T$ is obtained by taking the union of the shortest paths $p_0, p_1$ and $p_2$ between $c$ and the terminals $t_0$, $t_1$ and $t_2$ respectively. We call $c$ the center of $T$ (Figure 1). We observe that $c$ can be one of the terminals. Therefore, to construct such a Steiner tree, we can iterate through the vertices of $G$ and set as the center the vertex that minimizes the sum of the lengths of the shortest paths $p_0, p_1$ and $p_2$. In this paper, we will call this procedure $\mathsf{3Steiner}(G, t_0, t_1, t_2)$. We note that $\mathsf{3Steiner}(G, t_0, t_1, t_2)$ is a deterministic procedure that produces an optimal Steiner tree with three terminals, and runs in time $O(n + m \log(m))$ where $n$ the number of vertices and $m$ the number of edges.

## 3.3 Constructing an Initial Solution

Once the preprocessing procedure has been applied, our algorithm proceeds to construct suitable initial solutions. We actually implement two initialization functions. Both functions take as input a triple $(G, S, r)$ consisting of a graph $G$, a set of terminals $S$ and a root vertex $r$ as input, and return a Steiner tree with terminals $S$ rooted at $r$. We note that the root can be an arbitrary vertex in the graph, but in our implementation we always choose this root to be a terminal.

The first function, DetInitialST$(G, S, r)$, is used to construct a reasonable first-solution when our algorithm is executed for the first time. This function is completely deterministic. At each step, the function DetInitialST$(G, S, r)$ maintains the following data:

1. a partial Steiner tree $T$ spanning some of the terminals;
2. a graph $G/T$ obtained from $G$ by contracting $T$ to its root $r$; and
3. a shortest-path tree $D$ for $G/T$ rooted at $r$.

In the beginning, $T$ contains only the root $r$, $G/T = G$, and $D$ is simply the shortest-path tree for $G$ rooted at $r$. After this initialization has taken place, the algorithm enters in a loop, where at each iteration, two new terminals $t_1$ and $t_2$ are incorporated to the tree. Each iteration consists of three steps.

1. First, one applies a function SelectTerminals that selects terminals $t_1, t_2$ that will be added to the tree. This function proceeds as follows. First, it sets $t_1$ as the terminal with *greatest* distance to the root vertex $r$ in the graph $G/T$. Note that that the contraction of $T$ is simulated by setting the costs of its edges to 0 in the graph $G$. Subsequently, a shortest path from $r$ to $t_1$ is contracted, and $t_2$ is selected as the terminal with the *greatest* distance to $r$.
2. Once the terminals $t_1$ and $t_2$ have been determined, one calls the function 3Steiner to compute the minimum Steiner tree $T'$ in $G/T$ with respect to the terminal set $\{r, t_1, t_2\}$.
3. Finally, the two trees $T$ and $T'$ are *merged*. This merging process consists in taking the spanning tree of the union $T \cup T'$.

The three steps above are repeated until a Steiner tree spanning at least $|S| - 1$ terminals in $S$ has been obtained. Suppose that some last terminal $t \in S$ is not spanned by $T$, and that $v$ is a vertex in $T$ of minimum distance[1] to $t$. Then the tree $T$ is updated by incorporating to it the shortest path between $v$ and $t$. The algorithm described above is specified more formally in Algorithm 1. The procedure DetInitialST$(G, S, r)$ has time complexity $O(|S| \cdot (n + m \log(m)))$, where $n$ the number of vertices, $m$ the number of edges of $G$.

Once we have a starting Steiner tree, we will improve it by applying the optimization procedure described in Subsection 3.4. Since this optimization procedure may converge to a local minimum, we will repeat the optimization process with respect to several initial Steiner trees. Nevertheless, from this point on, each initial Steiner tree will be selected using a much cheaper procedure, which we call RandomInitialST$(G, S, r)$. This procedure simply selects random path between some terminal $t_1$ in $S$ and the root vertex $r$. Subsequently, it selects a random path between some terminal $t_2$ and some vertex in the first path, then a random path between some terminal $t_3$ and some vertex in the previous paths and so on, until all terminals have been selected. Each random path is selected by performing a random walk in the graph starting at the terminal to be added.

[1] More precisely $\text{dist}(v, t) = \min_{u \in T} \text{dist}(u, t)$, where the distance function is computed with respect to $G$.

**Algorithm 1** DetInitialST$(G, S, r)$.

**Input**: An edge weighted graph $G$, a set of terminals $S$, a vertex $r$
**Output**: A Steiner tree in $G$ connecting all terminals in $S$
$T \leftarrow r$
**while** *there are two terminals in* $S$ *not spanned by* $T$ **do**
  $G' \leftarrow G/T$
  $D \leftarrow \mathsf{ShortestPathTree}(G', r)$
  $t_1, t_2 \leftarrow \mathsf{SelectTerminals}(G', D)$
  $T' \leftarrow \mathsf{3Steiner}(G', r, t_1, t_2)$
  $T \leftarrow \mathsf{SpanningTree}(T \cup T')$
**end**
**if** *some terminal* $t \in S$ *is not spanned by* $T$ **then**
  Let $v$ be the vertex of $T$ with smallest distance to $t$
  Set $T \leftarrow T \cup p$ where $p$ is a shortest path between $v$ and $t$ in $G$
**end**
**return** $T$

## 3.4 Optimization Procedure

Once the preprocessing stage has been completed, and an initial Steiner tree has been computed using the procedure described in the previous subsection, our algorithm applies an optimization procedure that takes a Steiner tree $T$ rooted at a *terminal vertex* $r$ as input, and outputs a Steiner tree $T'$, also rooted at $r$, with equal or smaller weight than $T$. This optimization procedure is repeated until the time is up or until it has stabilized. Alternatively, the procedure can be halted by an external algorithm even if it has not stabilized. In this case the best Steiner tree computed so far is given as the result.

Intuitively, this optimization procedure works in two stages. In the first stage, we split the Steiner tree $T$ into three subtrees $T_1$, $T_2$ and $T_r$, where $T_1$ and $T_2$ are rooted at vertices $v_1$ and $v_2$ respectively, and $T_r$ is rooted at $r$. Subsequently, we select a vertex $v_r$ among the leaves of $T_r$ and reconnect the three subtrees by finding a suitable Steiner tree with respect to terminals $\{v_1, v_2, v_r\}$.

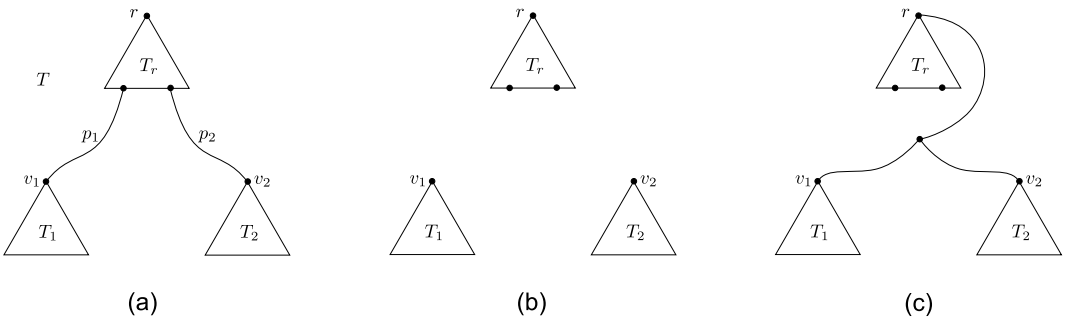


**Figure 2** (a) A Steiner tree $T$, a pair $\{v_1, v_2\}$ of vertices in $\mathsf{SelectCut}(T)$, and Steiner paths $p_1$ and $p_2$. (b) The internal vertices of $p_1$ and $p_2$ are removed. This results into three trees $T_r$, $T_1$ and $T_2$. (c) $\{r, v_1, v_2\}$ are connected using an optimal 3-terminal Steiner tree obtained using $\mathsf{3Steiner}(G, r, v_1, v_2)$ function.

Before describing the details of the procedure, we need to define the concept of a *relevant vertex*. Let $T$ be Steiner tree of $G$ a rooted at a vertex $r$. We say that a vertex $v \in V(T)$ is *relevant* for $T$ if $v$ is a terminal or if $v$ has at least 2 children in $T$. A path $p$ in $T$ is a *Steiner path* if the two endpoints of $p$ are relevant for $T$ and if the remaining vertices of $p$ are Steiner points of degree 2 in $T$. Note that each middle vertex of a Steiner path has a unique child. Let $v$ and $v'$ be relevant vertices. We say that $v'$ is a *relevant child* of $v$ if these two vertices are the endpoints of a relevant path in $T$ and if $\text{dist}(r, v) < \text{dist}(r, v')$.

The algorithm starts by applying a simple routine that prunes the input Steiner tree. More precisely, this routine processes the input tree by removing every Steiner point that does not have a terminal as descendant. Such Steiner points do not connect the root to any terminal, and therefore can be safely removed. The resulting tree is still a Steiner tree and every leaf is a terminal.

Subsequently, the algorithm executes a procedure $\mathsf{Improve}(G, S, T)$ that takes a graph $G$, a set of terminals $S$ and a tree $T$ as input, and tries to modify $T$ with the goal of reducing its cost by proceeding as follows.

1. First, we construct a set $\mathsf{SelectCut}(T)$ containing all pairs of the form $\{v_1, v_2\}$ where both $v_1$ and $v_2$ are relevant vertices, and the unique path connecting $v_1$ to $v_2$ in $T$ has no relevant vertex, other than possibly the root (which is always relevant).
2. Now, for each pair of vertices $(v_1, v_2)$ in the list $\mathsf{SelectCut}(T)$ built in the previous step, we call a function $\mathsf{Cut}(T, v_1, v_2)$ that cuts the tree above each of the vertices $v_1$ and $v_2$. More precisely, for each $i \in \{1, 2\}$, one deletes from $T$ the internal vertices of the *unique* Steiner path $p_i$ that starts at $v_i$ that is contained in the unique path between $v_i$ and $v_r$ in $T$. Such Steiner paths $p_1$ and $p_2$ are always guaranteed to exist because the root is a relevant vertex. This process splits the original tree into three disjoint subtrees $T_r$, $T_1$, $T_2$ (Figure 2.(b)).
3. Subsequently, the algorithm contracts each of the three subtrees into a single vertex. More precisely, $T_r$ is contracted to $r$, $T_1$ is contracted to $v_1$, and $T_2$ is contracted to $v_2$. We let $G'$ be the contracted graph. We note that in practice, the contraction of a subtree is simulated by setting the cost of each edge of the subtree to 0.
4. Finally we apply the subroutine $\mathsf{3Steiner}(G', r, v_1, v_2)$ to computes an optimal 3-terminal Steiner tree with terminal set $\{r, v_1, v_2\}$. This tree, together with the three subtrees $T_r$, $T_1$ and $T_2$ give rise to a graph $T_r \cup T_1 \cup T_2 \cup \mathsf{3Steiner}(G', r, v_1, v_2)$ whose weight is at most the weight of the input tree $T$. The algorithm then returns a spanning-tree of this graph.

A summary of the algorithm is provided below (Algorithm 2). Let $t$ be the number of terminals in the graph, and $\Delta$ be the maximum degree of the graph $G$. We note that relevant vertices are either terminals or have degree at least 2, therefore, there are at most $O(t)$ relevant vertices. And for each relevant vertices, the algorithm generates at most $\Delta^2$ pair of vertices. Therefore, the time complexity of the function $\mathsf{Improve}(G, S, r)$ is $O(t \cdot \Delta^2 \cdot (n + m \log(m)))$, where $n$ is the number of vertices and $m$ is the number of edges in $G$.

## 4 Experimental results

We have implemented our heuristic algorithm in C++ and compared it with six state-of-the art solvers for the Steiner tree problem, including solvers that competed at the PACE challenge 2018 [3]. We refer to our solver as 3TST, an acronym for *3-Terminal Steiner Tree*. The remaining solvers in our benchmark are named according to the surnames, or initials of their respective authors. These solvers are listed below.

**Algorithm 2** Improve$(G, S, T)$.

**Input:** An edge weighted graph $G$, a set of terminals $S$ and a Steiner tree $T$ in $G$ spanning $S$
**Output:** A Steiner tree $T'$ of cost at most cost$(T)$ spanning $S$.
**for** $\{v_1, v_2\} \in$ SelectCut$(T)$ **do**
  $G' \leftarrow G$
  **if** $v_1 \in T$ *and* $v_2 \in T$ **then**
    $(T_r, T_1, T_2) \leftarrow$ Cut$(T, v_1, v_2)$
    $G' \leftarrow G'/T_r/T_1/T_2$ ($G'$ is obtained by contracting $T_r, T_1$ and $T_2$)
    $T \leftarrow$ SpanningTree$(T_r \cup T_1 \cup T_2 \cup$ 3Steiner$(G', r, v_1, v_2))$
  **end**
**end**
**return** $T$

1. Grandcola's Solver[2] implements a local search algorithm.
2. HTKME Solver[3] combines a star contraction algorithm from [9] with several auxiliary heuristics.
3. HGSSB Solver[4] performs a shortest path heuristic followed by a local optimization step.
4. RCLG Solver[5] implements an evolutionary algorithm.
5. KR Solver[6] reduces the Steiner tree problem to a linear programming problem.
6. AO solver[7] is based on a local optimization heuristic.

We used the original implementation of each of these solvers in our benchmark, without any modification in the code. We benchmarked all the solvers on different data sets, some of which are well established datasets for the Steiner tree problem (PACE2018 dataset [3], Vienna dataset), and some of which are well known datasets in the field of networks (Urban Road Networks set [23], Network repository [26]). Finally, we also compared the solvers on synthetic data sets obtained by generating random $d$-regular graphs for distinct values of $d$.

For each graph considered in our benchmark, we run each solver with a time limit of 30 minutes. When the time limit was reached, each solver received a Unix signal SIGTERM, and had 30 seconds to output a solution before being killed. This is the same experimental setting used in the PACE challenge 2018, whose theme was the Steiner tree problem. Each solution is associated with a score, which is defined as the relative distance of the solution to the best solution found during the whole experiments. If the value of the solution is $v$ and the best solution is $b$ then the score is the ratio $\frac{v-b}{b}$. The score of a solver on a data set is the sum of the scores over all graphs in the data set. With this measure, the lower the score the better is the performance of the solver. In particular, a solver that gets a score of 0 in a given instance is the best solver on that instance.

For some instances of some data sets, some solvers did not output a feasible solution. In these cases, we assigned a default value for the instance. To avoid penalizing excessively a solver on such instances, we have defined the default value as the weight of a Steiner tree obtained by computing a minimum spanning tree of the input graph and subsequently by pruning this tree in such a way that each leaf is a terminal.

[2] http://www.dil.univ-mrs.fr/~gcolas/sgls.c
[3] https://github.com/goderik01/PACE2018
[4] https://github.com/maxhort/Pacechallenge-TrackC/
[5] https://github.com/HeathcliffAC/SteinerTreeProblem
[6] https://github.com/dRehfeldt/scipjack/
[7] https://github.com/SteinerGardeners/TrackC-Version1

**Table 1** Summary: ratios for all algorithms and all data sets. The smallest the value the better is the solver on a given data set. A value of 0 means that the solver was the best in all instances of the data set. Values in bold are the smallest values on the dataset among all the solvers. The superscript number in the column of our solver give the rank of our solver on that data set. For example 0.0397[2] means that our solver is the second best solver on the data set. (*) means that for some instances, the solver did not output a feasible solution. NC means that the solver could not find a solution for any of the instances of the data set.

| Set/Solvers | AO | Grandcolas | HGSSB | HTKME | KR | RCLG | 3TST |
|---|---|---|---|---|---|---|---|
| PACE 2018 | 1.3756 | 1.6682 | 3.6119 | 0.6589 | 0.1994 | **0.1755** | 0.7130[3] |
| Geo Original | 0.2579 | 0.4889 | 1.0904 | 0.1032 | **2.2048e-05** | 0.0707 | 0.0397[2] |
| Geo Prepro. | 0.0661 | 0.0639 | 0.9802 | 0.0723 | **0.0009** | 0.0543 | 0.0472[2] |
| I Simple | 0.0081 | 0.0354 | 0.2454 | 0.0054 | **0.** | 0.0102 | 0.0056[3] |
| I Advanced | 0.0294 | 0.0393 | 0.4100 | 0.0146 | **0.** | 0.0128 | 0.0199[4] |
| 3-regular | 0.3081 | 0.1656 | 1.1526 | 0.2324 | **0.** | 0.0610 | 0.1252[3] |
| 4-regular | 0.3027 | 0.2385 | 1.1380 | 0.3453 | 5.0182* | **0.0616** | 0.1414[2] |
| 5-regular | 0.4151 | 0.2642 | 1.0573 | 0.4176 | 9.8697* | **0.0123** | 0.1325[2] |
| 6-regular | 0.3472 | 0.1668 | 1.1315 | 0.4133 | 8.2781* | **0.0304** | 0.1729[3] |
| 7-regular | 0.4414 | 0.2909 | 1.0943 | 3.2680* | 8.7525* | **0.0332** | 0.2179[2] |
| 8-regular | 0.4854 | 0.2840 | 1.1815 | 1.1467* | 3.1947* | **0.0997** | 0.3123[3] |
| 9-regular | 0.3554 | 0.2205 | 1.0640 | 2.9180* | 2.4698* | **0.1008** | 0.2762[3] |
| 10-regular | 0.6159 | 0.2950 | 1.2266 | 4.5041* | 4.0639* | **0.0942** | 0.3585[3] |
| 20-regular | 0.5437 | 0.2385 | 8.4318* | 13.2870* | 6.4683* | **0.0671** | 0.4591[3] |
| City Road | 4.5950* | 3.2082* | 10.8455* | 0.5466 | **0.** | 5.3683* | 0.9224[3] |
| Big Road | 4.9210* | NC | NC | 4.1762* | 3.4190* | NC | **0.4513***[1] |

Each solver that uses a random procedure has an option to choose a particular seed with the goal of making a computation deterministic, and therefore reproducible. We used the same seed for all experiments (seed = 10). This seed was chosen before experiments were run. All our experiments were executed on Core™ i7-4770S computers with 16 Gb of RAM running Ubuntu™ 16.04.

In all figures and tables, our implementation is called 3TST. Table 1 summarises all experiments. This table gathers the sum of ratios obtained by each algorithm on each data set. The symbol (*) following an entry in the table is used to indicate that for some graphs in the data set, the solver did not output a feasible solution. *NC* means that the solver did not output a feasible solution for any of the graph in the data set. We can see that our implementation (3TST) obtains good results in most data sets. Additionally, it is worth noting that our implementation is the one that could find feasible solutions more often in the Big Road Networks data set, which contains graphs with millions of nodes.

## PACE-Challenge

Graphs of the PACE Challenge 2018 dataset were selected by the organizers of the competition from the hard instances of the well known Steinlib and Vienna data sets. The average number of vertices is $27K$, the average number of edges is $48K$, and the average number of terminals is 1114, with a median at 360.5. Finally, most of these instances have treewidth above 40.

Figure 3 shows the score of each solver on each instance. Instances are sorted by increasing number of vertices. On the first half of the instances, our implementation provide decent solution but not as good as RCLG, KR or HTKME which are among the four first in the PACE Challenge 2018. And Figure 4 show a focus on the second half without the HGSSB solver for clarity because it has quite large ratio compare to the other solvers. On those larger instances with smaller average degree, our implementation is very good and almost on par with KR which is the best solver on this part of the data. We note that the implementation of KR is based on the SCIP Optimization Suite, a state-of-the-art tool for mixed integer programming [10]. We also note that the maximum ratio of our solver on these instances was 0.008, while in most instances this ratio was much smaller.

### Vienna set

Graphs in the Vienna set were generated from real-world telecommunication networks at the University of Vienna. This dataset is split into several types of instances. We realized our benchmark in the so called $I$-Instances sub-dataset, which contains 85 instances representing deployment areas from various Austrian cities, but they also include rural areas with smaller population density and very sparse infrastructure. The underlying graphs contain between 7K and 178K nodes, 9K and 239K edges, and between 38 and 4991 terminals. I-instances are available after simple preprocessing that eliminates non-terminal nodes of degrees 1 and that contracts non-terminal nodes of degree 2.

Figure 5 shows the score of each solver on each instance of the I simple preprocessed instances data set without the HGSSB solver for clarity because it has quite large ratio compare to the other solvers. Instances are sorted by increasing number of vertices. We can see a similar behaviour as for the PACE Challenge instances. On small instances, our implementation gives decent solutions and show its strength on larger instance where it give very good solutions. The instances of this data set are small enough so that the KR Solver, which is base on an exact solver, manage to give the best solution in all case. On such data set solutions given by KR Solver can be seen as the ground truth.

### $d$-regular graphs

We generated random $d$-regular graphs using the random generator from the python package Networkx. The number of vertices were chosen uniformly at random from the range $[10000; 200000]$. The weights on the edge follow a normal distribution with mean uniformly chosen from the range $[2000; 10000]$ and standard deviation uniformly chosen from the range $[200; 2000]$. Negative weights were set to 0. The number of terminals was chosen uniformly at random between 2% and 10% of the number of vertices. Terminals were chosen uniformly at random from the vertices. We generated 10 graphs for each $d$ in the set $\{3, 4, 5, 6, 7, 8, 9, 10, 20\}$.

Figure 6 and Table 1 show the evolution of the ratio for each solver with respect to the degree $d$ of the vertices of the graphs. The best solver in these datasets was the solver RCLG, which implements a genetic algorithm. The ratios obtained by our solver (3TST) alternated between the second best and third best. This ratio varied from 0.1252 for 3-regular graphs, to 0.4591 for 20-regular graphs. We note that starting from $k = 4$, the solver KR, which reduces the Steiner tree problem to mixed integer-programming, started failing to give feasible solutions for some instances.

**Table 2** Big road networks: ratios for all algorithms on the big road networks data set. The smallest the value the better is the solver on a given data set. A value of 0 means that the solver was the best the instance. The superscript number in the column of our solver give the rank of our solver on that instance. For example $0.0891^{2}$ means that our solver is the second best solver on the instance. NC means that the solver could not find a solution for the instance.

| Set/Solvers | AO | Grandcolas | HGSSB | HTKME | KR | RCLG | 3TST |
|---|---|---|---|---|---|---|---|
| Instance 1 | NC | NC | NC | NC | NC | NC | $0.^{1}$ |
| Instance 2 | NC | NC | NC | 0. | NC | NC | $0.0891^{2}$ |
| Instance 3 | NC | NC | NC | NC | NC | NC | $0.^{1}$ |
| Instance 4 | NC | NC | NC | NC | 0. | NC | $0.1219^{2}$ |
| Instance 5 | NC | NC | NC | NC | 0. | NC | NC |

### City Road Networks

This well known data set contains graphs associated with road networks for 80 of the most populated urban areas in the world. As the original graphs were not connected we filtered each instance by taking only the largest connected component of each graph. Since these graphs do not come originally with information about terminal nodes, we selected these terminals at random. First, we selected a number $r$ uniformly at random in the range between 2% and 10% of the number of vertices. Subsequently, we selected $r$ distinct vertices uniformly at random among the vertices of the graph. The graphs contain between $2K$ and $685K$ nodes, $3K$ and $924K$ edges and between 246 and 53275 terminals

Figure 7 shows the score of each solver on each instance of the City Road Networks set. Instances are sorted by increasing number of vertices. We can see that on the first half of the instances, almost all solvers manage to give really good solution. As the size of the instances grow, the solver KR, which reduces Steiner tree to mixed-integer programming, starts to be the dominant best solver. Nevertheless, our solver still outputs solutions with a very good ration (of at most 0.2).

### Big Road Networks

In this data set was used to push the solvers to their limits. We selected 9 unweighted road networks with more than 1 million nodes. As in the previous data set, the number of terminals was chosen uniformly at random between 2% and 10% of the number of vertices. Since no solver could output a feasible solution for the 4 largest graphs we only show results for the remaining five. These graphs contain between $1087K$ and $6686K$ nodes, $1541K$ and $7013K$ edges, and between $52K$ and $661K$ terminals

Table 2 shows the ratio of each solver on each of these five instances. On this data set only KR, RCLG and our algorithm (3TST) managed to output some solution for some of the instances. On the 9 graphs, KR output 2 solutions, HTKME 1 solution, and our implementation 4 solutions. This dataset highlights one of the strengths of our solver, which is the ability to handle very large instances and still give good solutions, when compared with other solvers.

## 5 Conclusion

In this work, we introduced a simple combinatorial heuristic algorithm for the Steiner tree problem. Our heuristic is similar in spirit to the classic approximation algorithm of Robin and Zelikovsky [25], that works by replacing sub-trees of a prospective solution with Steiner trees

on a small set of terminals. In our case, we use a routine that splits a prospective solution Steiner tree into three disjoint subtrees, and that reconnects these subtrees by taking the union with a 3-terminal Steiner tree, where the terminals are the roots of the subtrees. We note that one distinguishing feature of our algorithm is that it is well suited for large graphs, since it does not require the book-keeping of the distances between all pairs of vertices in the graph. Indeed we almost only need to keep track of of the edges of a slightly pre-processed version of the input graph, where non-terminal vertices of degree 1 are removed, and edges containing non-terminal vertices of degree 2 are contracted.

Our experimental results have shown that our algorithm fits well the category of a general purpose Steiner tree heuristic, since it was able to obtain good solutions in all benchmarked datasets when compared with other solvers. We note that the best solver in some datasets was built upon a state-of-the art mixed-integer programming package. In some other datasets, the best solver was based on genetic algorithms. On the other hand, our algorithm essentially consists in the application of a single simple replacement routine that is applied multiple times until the time limit is reached. Still the solutions obtained by our solvers were very competitive, often being the second best in the benchmarks and with a very small ratio $(v - b)/b$ where $v$ is the weight of our solution and $b$ the weight of the best solver. It is also worth noting that our algorithm was able to handle graphs with millions of vertices, while most of the other solvers failed in all these big instances. Finally, it is worth noting that one possible application of our Steiner-tree improvement sub-routine is as a black-box that can be used to improve the solution output by other solvers.

## References


1 Piotr Berman and Viswanathan Ramaiyer. Improved approximations for the steiner tree problem. *J. Algorithms*, 17(3):381–408, 1994.

2 Marshall W. Bern and Paul E. Plassmann. The steiner problem with edge lengths 1 and 2. *Inf. Process. Lett.*, 32(4):171–176, 1989.

3 Édouard Bonnet and Florian Sikora. The PACE 2018 Parameterized Algorithms and Computational Experiments Challenge: The Third Iteration. In *Proc. of IPEC 2018*, volume 115, pages 26:1–26:15, 2019.

4 Jaroslaw Byrka, Fabrizio Grandoni, Thomas Rothvoß, and Laura Sanità. Steiner tree approximation via iterative randomized rounding. *J. ACM*, 60(1):6:1–6:33, 2013.

5 Goutam Chakraborty. Genetic algorithm approaches to solve various steiner tree problems. In *Steiner Trees in Industry*, pages 29–69. Springer, 2001.

6 Xiuzhen Cheng, Yingshu Li, Ding-Zhu Du, and Hung Q Ngo. Steiner trees in industry. In *Handbook of combinatorial optimization*, pages 193–216. Springer, 2004.

7 Mung Chiang, Henry Lam, Zhenming Liu, and H. Vincent Poor. Why steiner-tree type algorithms work for community detection. In *Proc. of (AISTATS 2013)*, volume 31, pages 187–195, 2013.

8 Diego de Uña, Graeme Gange, Peter Schachte, and Peter J Stuckey. Steiner tree problems with side constraints using constraint programming. In *Proc. of the 30th AAAI Conference on Artificial Intelligence*, 2016.

9 Pavel Dvorák, Andreas Emil Feldmann, Dusan Knop, Tomás Masarík, Tomas Toufar, and Pavel Veselý. Parameterized approximation schemes for steiner trees with small number of steiner vertices. In *Proc. of (STACS 2018)*, volume 96, pages 26:1–26:15, 2018.

10 Ambros Gleixner, Leon Eifler, Tristan Gally, Gerald Gamrath, Patrick Gemander, Robert Lion Gottwald, Gregor Hendel, Christopher Hojny, Thorsten Koch, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Jan Merlin Viernickel, Stefan Vigerske, Dieter Weninger, Jonas T.

Witt, and Jakob Witzig. The SCIP Optimization Suite 5.0. ZIB-Report 17-61, Zuse Institute Berlin, December 2017. URL: `http://nbn-resolving.de/urn:nbn:de:0297-zib-66297`.

11 S Louis Hakimi. Steiner's problem in graphs and its implications. *Networks*, 1(2):113–133, 1971.

12 Stefan Hougardy and Hans Jürgen Prömel. A 1.598 approximation algorithm for the steiner problem in graphs. In *Proc. of the 10th Symposium on Discrete Algorithms (SODA 1999)*, pages 448–453, 1999.

13 Rostam Joobbani. *An artificial intelligence approach to VLSI routing*, volume 9. Springer Science & Business Media, 2012.

14 Richard M. Karp. Reducibility among combinatorial problems. In *Proc. of Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103, 1972.

15 Marek Karpinski and Alexander Zelikovsky. New approximation algorithms for the steiner tree problems. *J. Comb. Optim.*, 1(1):47–65, 1997.

16 Emil Keyder and Hector Geffner. Trees of shortest paths vs. steiner trees: Understanding and improving delete relaxation heuristics. In *Proc. of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 1734–1739, 2009.

17 Theodoros Lappas, Kun Liu, and Evimaria Terzi. Finding a team of experts in social networks. In *Proc. of 15th International Conference on Knowledge Discovery and Data Mining (KDD 2009)*, pages 467–476, 2009.

18 Sookyoung Lee and Mohamed F. Younis. Recovery from multiple simultaneous failures in wireless sensor networks using minimum steiner tree. *J. Parallel Distrib. Comput.*, 70(5):525–536, 2010.

19 M Lundy. Applications of the annealing algorithm to combinatorial problems in statistics. *Biometrika*, 72(1):191–198, 1985.

20 Mohamed El Bachir Menai. A logic-based approach to solve the steiner tree problem. In *IFIP International Conference on Artificial Intelligence Applications and Innovations*, pages 73–79. Springer, 2009.

21 Hans Jürgen Prömel and Angelika Steger. Rnc-approximation algorithms for the steiner problem. In *Proc. of the 14th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1997), Proceedings*, volume 1200 of *LNCS*, pages 559–570, 1997.

22 Daniel Rehfeldt, Thorsten Koch, and Stephen J. Maher. Reduction techniques for the prize collecting steiner tree problem and the maximum-weight connected subgraph problem. *Networks*, 73(2):206–233, 2019. `doi:10.1002/net.21857`.

23 Urban Road Networks. Urban road network data, January 2016. `doi:10.6084/m9.figshare.2061897.v1`.

24 Gabriel Robins and Alexander Zelikovsky. Improved steiner tree approximation in graphs. In *Proc. of the 11th Symposium on Discrete Algorithms (SODA 2000)*, pages 770–779, 2000.

25 Gabriel Robins and Alexander Zelikovsky. Tighter bounds for graph steiner tree approximation. *SIAM J. Discrete Math.*, 19(1):122–134, 2005.

26 Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL: `http://networkrepository.com`.

27 Hiromitsu Takahashi. An approximate solution for the steiner problem in graphs. *Math. Japonica.*, 6:573–577, 1990.

28 Eduardo Uchoa, Marcus Poggi de Aragão, and Celso C. Ribeiro. Preprocessing steiner problems from VLSI layout. *Networks*, 40(1):38–50, 2002. `doi:10.1002/net.10035`.

29 Alexander Zelikovsky. An 11/6-approximation algorithm for the network steiner problem. *Algorithmica*, 9(5):463–470, 1993.

30 Alexander Zelikovsky. Better approximation bounds for the network and euclidean steiner tree problems. *University of Virginia, Charlottesville, VA*, 1996.

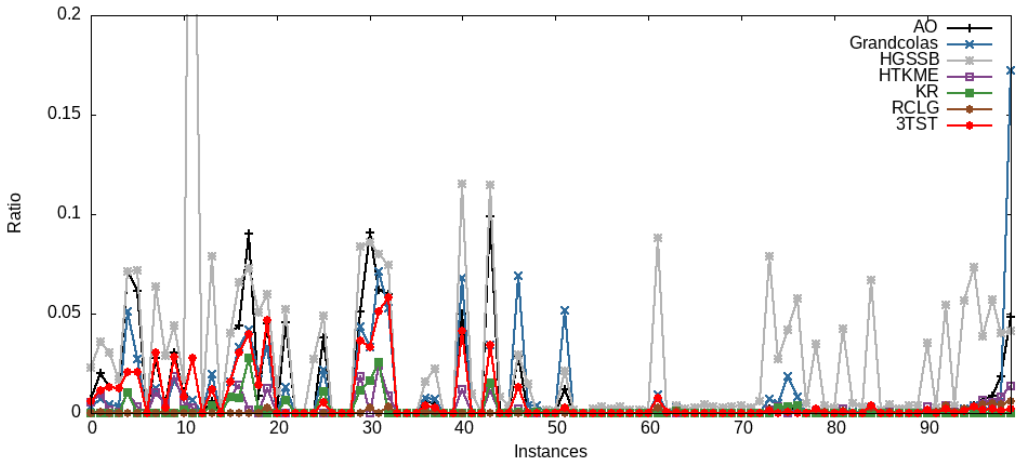


**Figure 3** PACE Challenge: Show the ratio obtained by each solver on each instance of the PACE Challenge data set. Instances are sorted by increasing number of vertices.

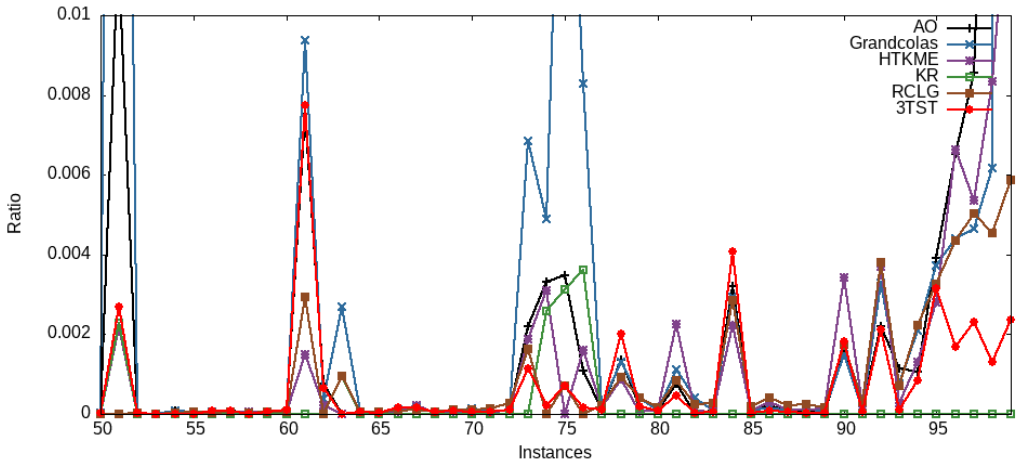


**Figure 4** PACE Challenge: Show the ratio obtained by each solver on the 50 largest instances of the PACE Challenge data set. Instances are sorted by increasing number of vertices.

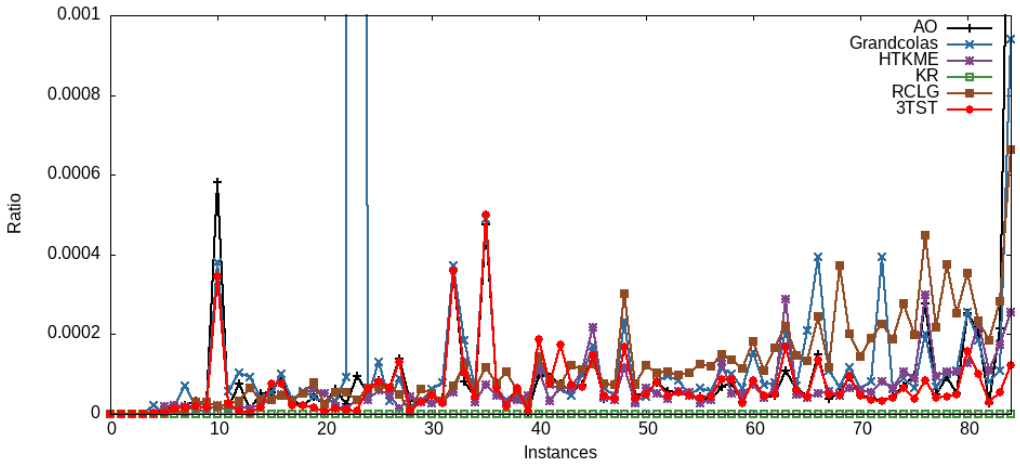


**Figure 5** I simple: Show the ratio obtained by each solver on each instance of the I simple preprocessed instances data set. Instances are sorted by increasing number of vertices.

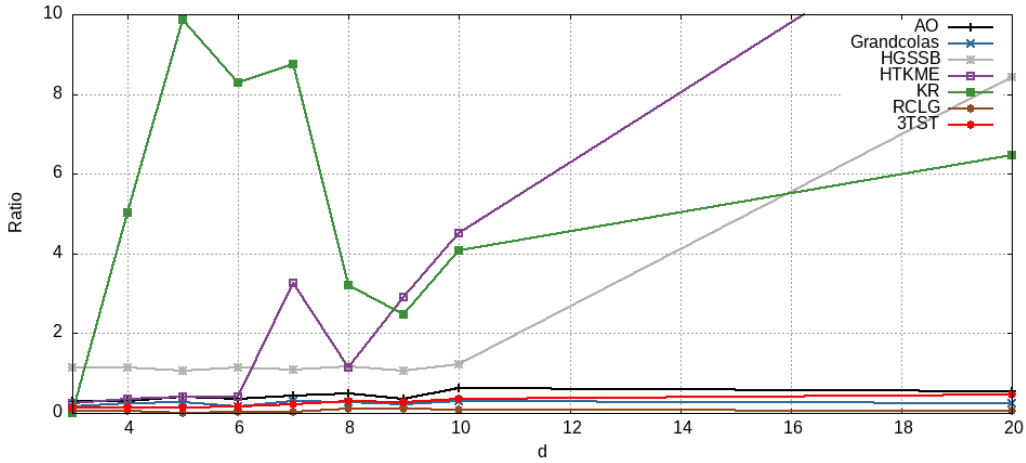


**Figure 6** $d$-regular: Show the ratio obtained by each solver on $d$-regular random graph. Show the evolution of the ratio with respect to increasing values of $d$.

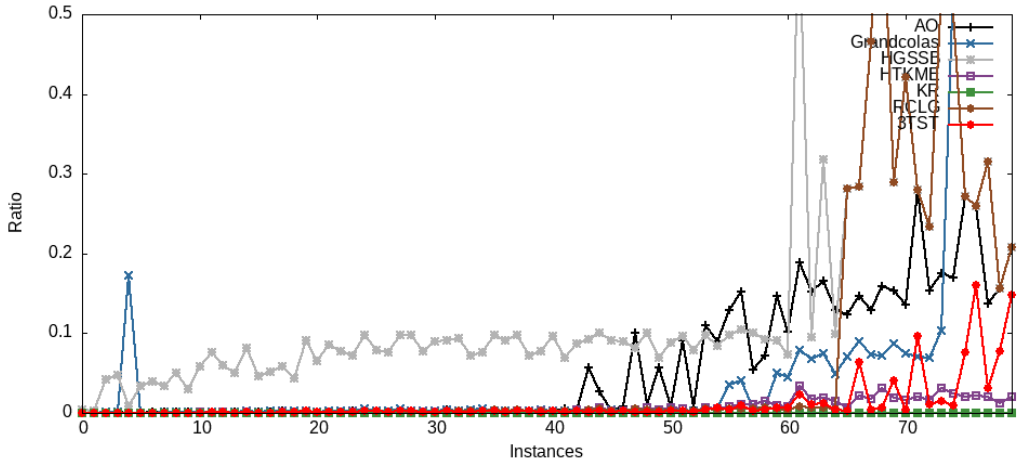


**Figure 7** City road networks: Show the ratio obtained by each solver on each instance of the City road networks data set. Instances are sorted by increasing number of vertices.