Reference manual

Kemény Tamás

Jan-Jul 2018

1 Introduction

This repository contains an implementation of a Parameterized Approximation Scheme (PAS) for the Steiner tree problem for integer edge-weighted undirected graphs. The work is based on the recent paper *Parameterized Approximation Schemes for Steiner Trees with Small Number of Steiner Vertices* by *Pavel Dvořák, Andreas Emil Feldmann, Dušan Knop, Tomáš Masařík, Tomáš Toufar and Pavel Veselý* [1].

The Steiner tree problem:
Given:

- a graph $G = (V, E)$
- positive edge weights $w : E \to \mathbb{R}^+$
- a set $R \subseteq V$ of *terminals* ($V \backslash R$ are called *Steiner vertices*)

Find:
Steiner tree $T \subseteq G$ connecting all terminals of R with minimum total weight $\sum_{e \in E(T)} w(e)$

2 Technique

The main idea of the program is to use the reduction method studied in the mentioned paper to reduce the size of an instance of the Steiner tree problem while outputting approximated edges, until a certain threshold, and then run the classical FPT algorithm by Dreyfus and Wagner [2] to solve the remaining part of the instance. The outputs of the two algorithms are then combined to form a Steiner tree. To describe the reduction method, we first define the ratio of a subgraph $K_{1,k}$ i.e. a star S in the input graph as:

$$ratio(S) := \frac{\sum_{e \in S} w(e)}{|S \cap R| - 1}$$

A single reduction step involves contracting the star with minimal ratio and adding its edges to the output. The effectiveness of this technique is studied in detail in the cited paper, where it is proved that as long as there are sufficiently many terminals left in the graph, these contractions only lose an ϵ-factor compared to the optimum. The idea also works well intuitively as we can see from the above definition that this method favors contracting stars with small edge weights and many terminals. This means the heuristic works to build a Steiner tree of small total weight while greatly reducing the number of terminals of the input graph, which will then be fed to the FPT algorithm that is parameterized by the number of terminals, once it falls below a certain threshold specified by the user.

The program first parses the arguments it is passed in the Main class and constructs an instance of the Graph class from the inputted .gr file. A preprocessing algorithm is then run in order to remove Steiner vertices with degree ≤ 1 and smooth out Steiner vertices with degree 2. If a threshold (characterized by the number of terminals or even the number of Steiner vertices) is initially not passed to the program, the reduction technique is run exhaustively by the Kernel class while constructing a sequence of saves of the original graph after each reduction, so that the parameters of these graphs can be printed and the user may make a more informed decision regarding the input parameters. Once this has been done, the appropriate save is selected and passed to the DreyfusWagner class to obtain the exact solution to that graph. The Steiner tree that is finally outputted is the union of two subgraphs: the edges obtained from the reduction method and the edges computed by the FPT algorithm. The weights of edges that make up these two parts are denoted w_1 and w_2, respectively. The Steiner tree is then printed to standard output in the following format: first the weight of the union of the two subgraphs $w_1 + w_2$, then all the edges of the Steiner tree followed by w_1, the edges corresponding to w_1, w_2, and the edges corresponding to w_2.
Instead of inputting a threshold, the program can also accept an accuracy parameter $\in [0 \ldots 10]$ which serves as a rough indication to the accuracy the user desires. Note that specifying the accuracy or a threshold for the number of Steiner vertices has no ϵ-factor guarantee.
The program may also be passed an option to continuously output the best Steiner tree computed so far, until the optimum is eventually reached. Note that this takes much longer than only computing the optimum Steiner tree.
In addition the user may specify the kind of output desired – for a full list of option see the Usage section.

3 Classes

SteinerGraphEdge An instance of this class represents an edge of the graph. Each edge contains integers representing its two endpoints and its weight. Each edge of the input graph is assigned one unique id, however this id is stored as a singleton list, since after the metric completion of the graph, some edges will represent a path in the input graph rather than a single edge. In this case the id of the edge corresponds to the ids of each edge along the path.

SteinerGraphVertex Each instance has information about the vertex and its neighbouring vertices. A SteinerGraphVertex contains a boolean isTerminal, a list of neighbouring vertices sorted increasingly by the weight of the incident edge, a set of neighbouring terminals and a list of sums of weights of edges to neighbouring terminals up to an index. The latter list is precomputed once reading of the input graph is finished and recomputed when the list of neighbouring terminals changes. This way during calls to the bestStar method, the appropriate sum may be fetched and divided by the number of terminals in the star minus one for a fast computation of the minimum ratio.

SteinerGraph The SteinerGraph class stores all data necessary to represent an instance of the Steiner problem i.e.: a set of vertices, edges, terminals and a weight function. Furthermore the class implements methods to modify the graph, such as adding and removing edges or modifying Steiner vertices into Terminals.

Kernel The Kernel class contains methods implementing the graph operations the reduction method relies on, such as bestStar, contractStar and computeClosure:

1. Pair< Double, List <SteinerGraphEdge> > **bestStar**(Integer center)
   The bestStar method computes the star with the best possible ratio centered at a given vertex and returns a Pair whose first element is a double representing the minimum ratio and second element is a list of the corresponding edges of the star.

2. void **contractStar**(List<SteinerGraphEdge> starEdges, Integer center)
   The contractStar method is equivalent to successively performing edge contractions for every edge in starEdges, however to prevent loops and to make things faster the routine does the following:
   (a) introduce a new vertex, mark it as terminal
   (b) for each vertex in starEdges gather its neighbourhood (excluding vertices in starEdges) in a set
   (c) add a new edge between the new vertex and each vertex in the set
   (d) remove starEdges from the graph.
3. void **computeClosure**(Integer upperbound)
   The computeClosure method finds the metric closure of the graph up to some upperbound L, that is for every pair of non-adjacent vertices a connecting edge is added to the graph with weight equal to the minimum distance between the vertices. This is done by running Dijkstra's algorithm starting from each vertex of the graph. During a Dijkstra procedure if the top of the heap exceeds the upper bound, the computation is stopped and the state of the routine is saved (min-heap, list of visited nodes) so that we may continue once the upper bound is incremented.

The Kernel class however does not specify what order star contractions and the metric completion should take place, instead it serves as a helper class for **FastKernel** and **SlowKernel**.

FastKernel The FastKernel class inherits from the Kernel class. It performs star contractions until no more stars are available, then computes the metric completion of the graph up to a certain threshold L. This is done by running Dijkstra from every terminal and adding edges between non-adjacent vertices with edge weights equal to the shortest path length between the two vertices if the distance is $\leq$ L. The threshold L is doubled each time no contractible star is found and computation of the metric completion is continued. This technique significantly reduces the number of edges that have to be added to the graph.

SlowKernel The SlowKernel class also inherits from the Kernel class, but instead of the the incremental completion of the graph however, this method computes the whole completion of the graph and then contracts a single star.

DreyfusWagner This class provides an implementation of the classical Dreyfus and Wagner algorithm [2]. The algorithm solves the problem exactly using dynamic programming, the idea is to compute a tree spanning a larger and larger subset of the set of terminals, until we have the optimum Steiner tree. The table can be computed by the following recurrence relation:
For every $D \subseteq R$ of size at least 2, and every $v \in V(G) \backslash R$

$$T[D, v] = \min_{\substack{u \in V(G) \backslash R \\ \emptyset \neq D' \subsetneq D}} \{T[D', u] + T[D \backslash D', u] + dist(v, u)\}$$

Observe that entries $T[D, v]$ where $D = \{u\}$ is a singleton set can be filled in trivially by $dist(v, u)$. The weight of the optimum Steiner tree can be read from the bottom row $T[K, v]$ and the Steiner tree can be reconstructed by backtracking from the optimum entry in the table through a tree of parent pointers. Note that this algorithm has exponential time and space complexity.

ReadInput The ReadInput class reads a given file containing the description of an instance of the steiner problem and returns an instance of the SteinerGraph class. The specified file must be in the .gr format.

For documentation on more classes, see the Javadocs (via doc/index.html).

The description of a file in the .gr format as specified by and used in *The PACE 2018 Parameterized Algorithms and Computational Experiments Challenge* [3] is the following:

> The file starts with a line 'SECTION Graph'. The next two lines are of the form 'Nodes #nodes' and 'Edges #edges', always in that order, where #nodes is the number of vertices and #edges is the number of edges of the graph. The #edges next lines are of the form 'E u v w' where u and v are integers between 1 and #nodes representing an edge between u and v of weight the positive integer w. The following line reads 'END' and finishes the list of edges.
>
> There is then a section Terminals announced by two lines 'SECTION Terminals' and 'Terminals #terminals' where #terminals is the number of terminals. The next #terminals lines are of the form 'T u' where u is an integer from 1 to #nodes, which means that u is a terminal. Again, the section ends with the line 'END'.

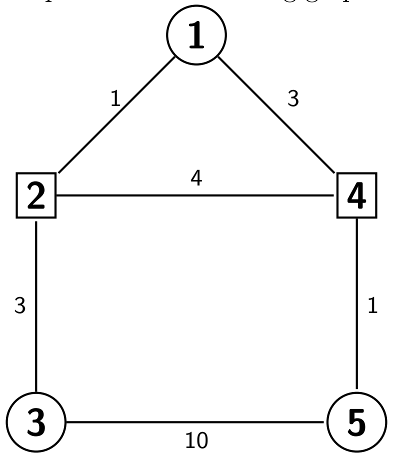
Example: instance000.gr

```
SECTION Graph
Nodes 5
Edges 6
E 1 2 1
E 1 4 3
E 3 2 3
E 2 4 4
E 3 5 10
E 4 5 1
END

SECTION Terminals
Terminals 2
T 2
T 4

END
```

Represents the following graph:



Sample instances can be downloaded from a link in the Downloads section in the README.md of this repository. Files instances[000-199].gr are also accredited to *The PACE 2018*.

4 Usage

Run the executable jar with the following java command:
`java -jar steiner-tree-pas.jar -in <input filename>.gr`

List of parameters:

`-ui [input|output]` - display input or output file in interactive ui

`-silent` - mute standard output message

`-long` - more detailed standard output

`-s p` - run FPT algorithm on at least `p` Steiner vertices

`-t r` - run FPT algorithm on at least `r` Terminals

`-a [0..10]` - instead of using options `-s` or `-t` we may specify a number between 0 and 10 as a rough indication of the accuracy with which we want to run the approximation scheme (0 being the least and 10 the most accurate)

`-c` - continuously find better approximations, until the optimum is found

`-out` - write computed Steiner tree into **`<input filename>-tree.gr`**

`-slow` - use slower, but possibly more accurate reduction method

Examples:

- **`java -jar steiner-tree-pas.jar -in instances/instance001.gr -ui input`**
  Display instance001.gr in the ui
- **`java -jar steiner-tree-pas.jar -in instances/instance001.gr -ui output -a 10`**
  Compute optimum and display Steiner tree in the ui
- **`java -jar steiner-tree-pas.jar -in instances/instance019.gr -c -long -out`**
  Compute exhaustively the best feasible solution for instance019.gr with detailed standard output and keep writing the Steiner tree into **`instance019-tree.gr`**

5 Dependencies

For development the following dependencies are needed:

JDK 8+ - http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Maven - https://maven.apache.org/install.html

References

[1] Pavel Dvořák, Andreas Emil Feldmann, Dušan Knop, Tomáš Masařík, Tomáš Toufar, and Pavel Veselý: Parameterized Approximation Schemes of Steiner Trees with Small Number of Steiner Vertices. v3 abs/1710.00668 URL https://arxiv.org/abs/1710.00668

[2] Dreyfus, S.E., Wagner, R.A.: The Steiner problem in graphs. *Networks*, 1(3):195–207, 1971. doi: 10.1002/net.3230010302.

[3] The PACE 2018 Parameterized Algorithms and Computational Experiments Challenge. In *13th International Symposium on Parameterized and Exact Computation (IPEC 2018)*, 2018+. URL https://pacechallenge.wordpress.com/pace-2018/.