Highlights

Searching for a Feedback Vertex Set with the Link-Cut Tree

Luís M. S. Russo

- First application of the link-cut tree data structure to the Feedback vertex set problem.
- Simple simulated annealing algorithm obtained significant results on the extensive PACE 2016 dataset.
Searching for a Feedback Vertex Set with the Link-Cut Tree

Luís M. S. Russo

âINESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal, Portugal

Abstract

In this paper we study the Feedback Vertex Set problem of an undirected graph. This problem has several practical applications and is known to be NP-Complete. We consider this problem from its dual formulation which consists in finding a largest subset of vertexes whose induced subgraph is a forest, i.e., contains no cycles. Our main goal is to explore how the link-cut tree data structure can be used in this context. This data structure makes it possible to identify and resolve cycles quickly. This property is useful for this problem and allows us to support a simple simulated annealing algorithm. We used the PACE 2016 dataset to test the performance of the resulting algorithm. The worst approximation ratio obtained was about 1.21. The algorithm is very fast. The longest execution time was 103 seconds. To justify the importance of escaping local mininums we also present a simple greedy algorithm.

Keywords: Feedback Vertex Set, Link-Cut Tree, undirected graph, Simulated Annealing

1. Introduction

In this paper we propose a new approach to the feedback vertex set (FVS) problem of an undirected graph. Our algorithm is based on simulated annealing (SA) and on the link-cut tree (LCT) data structure. It seems to us that the LCT is particularly well suited for the FVS problem over undirected graphs because it provides an extremely efficient way to detect and handle cycles.

The Feedback Vertex Set (FVS) problem is a well-known computational problem in graph theory that has important applications in diverse areas such as...
as network design, software engineering, bioinformatics, and control theory. The FVS problem is to find a minimum-sized set of vertices in a directed graph such that the removal of these vertices and their incident edges results in a graph without any directed cycles. These sets, also known as feedback vertex sets, have been extensively studied due to their significant impact on the design and analysis of complex systems.

In science, FVSs are related to the study of biological networks, such as gene regulatory networks, protein-protein interaction networks, and metabolic networks. These networks can be modeled as directed graphs, where the feedback vertex sets can represent key nodes or genes that are critical for the regulation and function of the system. Identifying these feedback vertex sets can provide insights into the behavior of the system and can help in the discovery of new drug targets.

In engineering, FVSs are related to the design and analysis of complex systems, such as communication networks, transportation networks, and power grids. In these systems, the removal of a feedback vertex set can lead to the elimination of cycles that cause instability, congestion, or failure. Therefore, identifying and minimizing the size of these sets is crucial for improving the efficiency, reliability, and safety of the system. Related work on this problem is summarized in Section 6.

Overall, the FVS problem is an important research topic that has a wide range of applications in various fields. In this paper, we provide an overview of the FVS problem, its applications, and the state-of-the-art algorithms and techniques for solving it. The overall structure of the paper is the following:

- In Section 2 we give a simple illustration of the problem. We point out the duality between a FVS and a forest of trees in the induced subgraph.

- In Section 3 we explain how to model this problem so that we can apply the simulated annealing technique. We define the notion of state and neighborhood. We also introduce the link-cut tree and point out the efficiency gains that can be obtained by using this data structure. In essence the LCT is much faster at dealing with cycles as it requires only $O(\log V)$ amortized time for this kind of operation. This is much faster than using a depth-first search (DFS), or a breadth-first search (BFS), which require $O(V + E)$ time.

- Section 4 fills in the necessary details. We give a longer more in-depth
exposition of the SA algorithm. We also give a brief description of the LCT. Crucially we explain in detail how to efficiently use this data structure to transform a state $s$ into a neighbor state $s'$. We also discuss several implementation issues related to the SA algorithm and the temperature parameter, Section 4.1. We finish this section by describing a simple greedy algorithm that is used to show the advantage of the SA approach, Section 4.2.

- In Section 5 we use the PACE dataset to do an extensive experimental analysis of our algorithm. Our algorithm obtains good performance. The worst approximation ratio obtained was about 1.21. The algorithm was almost always extremely fast. The longest execution time was 103 seconds. The full experimental results are given in Appendix A.

- Section 6 presents some related work, including a brief description of the PACE solvers we used to compare with our prototype. It also includes references for the LCT and SA.

- Section 7 finishes the paper with some analysis, conclusions and ideas for future work.

An extensive description of the related work is given in Section 6, for now let us give some perspective on the contribution of our work. There is an extensive literature on the FVS problem as it was one of the initial NP-hard problems identified by Karp [1]. An important assessment of existing solutions resulted from the PACE 2016 challenge [2]. We tested our algorithm with this data set and against the two best solvers of the competition. Previous applications of the SA algorithm to the directed version of the FVS problem exist [3, 4]. As far as we know our proposal is the first application of SA the undirected version of the problem. However this is not the main novelty of the paper. The main novelty of the paper is the use of the LCT, which, as far as we know, was never used before in the context of the FVS problem. We used the LCT combined with the SA algorithm as this seemed the simplest way to obtain a working algorithm. We expect that the LCT will probably be used in future FVS algorithms and yield further good algorithms for this problem. To validate the necessity of using the LCT and SA we compared against a simple greedy algorithm, based on the union-find data structure, described in Section 4.2. The greedy algorithm can get stuck very far from the global optimum, although this did not occur in all test.
2. The problem

Consider the undirected graph $G$ in Figure 1. Our goal is to extract a tree from $G$. Actually it need not be a single tree, it may be several disconnected trees, i.e., a forest. The main point is that the resulting object should not contain cycles, such as the one that connects 1, 2, 5 and 4. This is fairly straightforward to do if we can select edges. In this example it is enough to discard the edge $\{1, 2\}$ to break the cycle. Such trees can be obtained as the side product of simple graph search algorithms such as depth-first search (DFS) or breath-first search (BFS). If weights are associated with the edges these trees may be obtained from shortest paths or be minimum spanning trees. Nevertheless we are interested in discarding vertexes, not edges. More precisely we are trying to determine the largest subset $F \subseteq V$, where $V$ is the set of vertexes of $G$, such that the induced subgraph contains no cycles. An induced subgraph is such that if $\{u, v\}$ is an edge of $G$ and $u$ and $v$ are in $F$ then $\{u, v\}$ is necessarily part of the subgraph.

Given this definition is should be clear that we can not have $\{1, 2, 4, 5\} \subseteq F$ since in that case the induced subgraph contains a cycle. Hence at least one of these vertexes must be discarded. The goal is to discard the minimum amount of vertexes so that $F$ becomes the maximum induced forest. Given this focus on cycles most literature describes the dual problem, i.e., the FVS problem. In this problem the goal is to find the minimal set $S$ such that $V \setminus S$ is a forest. Hence a set $S$ is an FVS if for any cycle $C$ of $G$ at least one of its vertexes is in $S$. In our graph we have that the subgraph induced by $F = \{2, 4, 5, 6, 8\}$ is a forest and therefore $S = \{1, 3, 7\} = V \setminus F$ is a FVS. We illustrate this solution in Figure 2 by representing the vertexes in $S$ with dashed lines and the vertexes in $F$ with continuous lines. Likewise the edges in the induced subgraph are drawn with continuous lines, whereas the remaining edges are drawn with dashed lines. By inspecting Figure 2 we can verify that there are no cycles formed exclusively by edges drawn with continuous lines. Therefore $S$ is indeed a FVS. Note also that this is not the minimum FVS for this example, the minimum FVS is simply the vertex 5.

3. The approach

Our approach to the FVS problem is to use simulated annealing (SA). To use this meta-heuristic we need to identify the notions of state and neighbor. For completeness we describe our implementation of SA in Section 4. A state
s will contain information about a feedback vertex set \( S \subseteq V \). A state also contains further information to support efficient transitions, however it is uniquely determined by the set \( S \). Recall the example in Figure 2. In this example the set \( S = \{1, 3, 7\} \) is a FVS. The underlying undirected graph \( G = (V, E) \) has the vertex set \( V = \{1, 2, 3, 4, 5, 6, 7, 8\} \).

Let us consider another state \( s' \). We need to decide if \( s' \) is a neighbor of \( s \). If this is the case then we need to transform \( S \) into \( S' \) by some computationally efficient process. In our example we have \( S' = \{3, 5, 7\} \), which is obtained from \( S \) by inserting the node 5 and removing node 1. This process is illustrated in Figure 3.

Let us dissect the process involved in this transformation and discuss how
to implement it. We find it easier to describe this process as a transformation
from $F$ to $F' = V \setminus S'$, as it maps directly into the data structure we used.
Hence state $s$ will also store the forest $F$. In this example $F = \{2, 4, 5, 6, 8\}$.
We start by selecting a vertex $v$ from $S$. In this case we selected $v = 1$. Next
we consider the subgraph induced by $F \cup \{v\}$. Our main concern is whether
this subgraph still is a forest, i.e., does it contain any cycles? In this example
it does indeed contain a cycle $C$, in particular the nodes involved in the cycle
are 1, 2, 4 and 5. To eliminate this cycle we can simple choose a vertex $v'$ from
$C$. Next we removed $v'$ from $F \cup \{v\}$. In this example we considered
$v' = 5$ and thus reached the set of nodes $F' = (F \cup \{v\}) \setminus \{v'\} = \{1, 2, 4, 6, 8\}$.

If the subgraph $F \cup \{v\}$ does not contain a cycle then it is a neighbor
of $F$. If it contains more than one cycle then it will be necessary to remove
nodes until all the cycles are broken. We illustrate these cases in the following
Figures:

- Figure 4 shows the case where $F \cup \{v\}$ is a forest and therefore it is
  not necessary to remove any node $v'$.
- Figure 5 shows the case where $F \cup \{v\}$ has more than one cycle but it
  is enough to remove only one vertex $v' = 2$.
- Figure 6 shows the same case as Figure 5 but where two nodes are
  removed, $v' = 3$ and $v'' = 1$.

As illustrated in Figure 6 not all neighbors are obtained by removing the
minimum amount of nodes from $F \cup \{v\}$. We remove as many as reasonably
necessary. In this example if we first remove node 3 the resulting forest still
contains a cycle, which can be broken by removing node 1. Note that instead
we could have removed node 2, in which case we should have $S' = \{2, 3, 7\}$.
We do not remove an unnecessarily large amount of vertexes. In the example

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure4.png}
\caption{Transformation of $S = \{3, 5, 7\}$ into $S' = \{5, 7\}$ with $v = 3$.}
\end{figure}
of Figures 5 and 6 the subgraph $F \cup \{v\}$ contains “essentially” two cycles, meaning that it is never necessary to remove more than two nodes in order to break them. Therefore we never consider removing three nodes, which in particularly means that $S' = \{1, 2, 3, 7\}$ is not a neighbor of $S$ in this example. Moreover not all neighbors of $S$ are treated the same way by the annealing process. Recall that the goal of this technique is to minimize the number of vertexes in the resulting FVS. A detailed description of this process is discussed in Section 4.

Hence let us summarize and formalize the notion of neighbor. A state $s'$ is a neighbor of a state $s$ if and only $F' = (F \cup \{v\}) \setminus W$ for some set $W \subseteq F$ of vertexes that get removed. Notice in particular that we do not consider $s$ as a neighbor of itself as node $v$ can not be simultaneously inserted and removed. In general there is no point in removing the node $v$ as the resulting neighbor would contain no new vertexes. At each step the SA process modifies the current state $s$ into some of its neighbors $s'$. The probability that such a transition is accepted depends significantly on the size of the set $W$. Smaller $W$ sets are significantly more likely to get accepted. In the frontier case where $W = \emptyset$ the transition is automatically accepted.

Now let us focus on how to implement the transformation process we
have just described. The first problem is to identify the cycles in $F \cup \{v\}$. Any such cycle can be identified in $O(V + E)$ time by computing a DFS or a BFS, starting at $v$. In fact we can tighten the time bound to $O(V)$. Recall that the search is done on the induced subgraph which is a forest and therefore contains at most $O(V)$ edges. This is a high cost which must be paid even if the resulting cycle is small, essentially because the DFS and BFS algorithms do not know where the cycle is in the graph and may have to explore large portions of it. Instead we propose to use the LCT data structure [5], which can be used to detect cycles much more efficiently in $O(1 + \log V)$ amortized time. This is a much faster approach, which holds even if the resulting cycle is large. Moreover the LCT also supports selecting a node from this cycle in $O(1 + \log V)$ amortized time. Hence with the LCT we can perform the desired transformation much faster. The precise bound is the following: transforming the set $F$ into $F'$ by inserting the node $v$ with degree $n$ and removing the nodes in $W = \{v'_1, \ldots, v'_k\}$ with degrees $n_1, \ldots, n_k$ requires at most $O(1 + (n + n_1 + \ldots + n_k)(1 + \log V))$ amortized time. In the next Section we fill in the necessary details and explain how this process is used in the SA algorithm.

4. The details

In this Section we describe how we combined the SA algorithm and the LCT data structure. The SA algorithm is used to solve optimization problems, i.e., maximization or minimization. We use it as a maximization procedure that tries to determine the largest forest $F$ that is an induced subgraph of $G$. This is equivalent to trying to determine the smallest feedback vertex set $S$ of $G$. Note that $F$ and $S$ are always complementary, i.e., $F = V \setminus S$. This decision is arbitrary but since most of the effort was in implementing the LCT data structure it seems natural to consider it as the most important component of a state. On the other hand this has the side effect that the temperature parameter increases during the process instead of decreasing, as in the actual metallurgic annealing process. This is a minor inconvenience as our description focus essentially on the probability $\bar{p}$ of moving to a neighbor that reduces the current forest.

Let us start with the SA algorithm. At each step the algorithm considers only one state $s$. For our application the state is determined by a set of vertexes $F \subseteq V$ such that the induced subgraph is a forest, i.e., contains no cycles. We use the LCT to store $F$. Moreover each state also stores the
corresponding FVS, i.e., $S = V \setminus F$. The set $S$ is stored in an array, in an arbitrary order. This array is used to choose a vertex $v \in S$ uniformly at random. We then consider the set $F \cup \{v\}$, as described in Section 3.

The SA algorithm assigns to each state $s$ a given energy value $E(s) = |F|$, for this application we use the number of vertexes in the current forest $F$. For example the energy of the forest shown in Figure 2 is $5 = |\{2, 4, 5, 6, 8\}| = |F|$. This is precisely the value we wish to maximize. The larger this value is the better the solution in the current state, i.e., a larger induced sub-forest means a smaller FVS. Whenever our transformation process changes from $s$ to $s'$ and $E(s') > E(s)$ the transition is always accepted. However if $E(s') < E(s)$ this change has a small probability of being accepted by the SA algorithm. More precisely using the energy variation value $d = E(s') - E(s)$ the SA criteria is the following:

- If $d > 0$ then always accept the transition.
- If $d = 0$ then accept the transition with a very high probability.
- If $d < 0$ accept the transition with a very small probability that depends on $d$.

To enforce the probabilistic requirements of the last two cases we use random numbers. In theory we would use a random number $r$ uniformly from $[0, 1]$. In practice we use a random number with 32 bits obtained with the `arc4random` function\(^1\), see the last paragraph of Section 10. The probabilities mentioned become smaller and smaller as the algorithm progresses. The initial probability the final probability and the number of transitions to consider are given as arguments to the SA procedure. Algorithm 4 gives a pseudo-code description of the SA algorithm.

We used a simple linear update of the temperature value $T$, this produced reasonable experimental results. One important detail in our implementation is the order of the operations in lines 10 and 11. Usually these operations are performed in the reverse order. First the state $s'$ is generated and then it is inspected to decide whether the transition is accepted or not. However we invert this order to increase performance. In our actual implementation the test in line 12 is included into the generation process\(^2\) of line 11. This has

\(^1\)Utility functions from BSD systems (libbsd, -lbsd)
\(^2\)For explanation purposes we chose to show this decision explicitly in Algorithm 4.
1: \{ \bar{p}_S \text{ is the starting probability} \}
2: \{ \bar{p}_E \text{ is the ending probability} \}
3: \{ z \text{ is the number of steps} \}
4: Set \( T_S \) with Equation (1) using \( p = 1 - \bar{p}_S \) and \( \ell = -1.5 \)
5: Set \( T_E \) with Equation (1) using \( p = 1 - \bar{p}_E \) and \( \ell = -1.5 \)
6: \( s := \emptyset \) \{ Initial state \}
7: \( T := T_S \) \{ Set initial Temperature \}
8: \( \partial T := (T_E - T_S)/z \) \{ Set Temperature increment \}
9: \textbf{for} \( z \) iterations \textbf{do}
10: Obtain \( \ell \) using \( T \) and a random \( r \in [0, 1] \)
11: Generate a transition to a neighbor state \( s' \) with \( d = E(s') - E(s) \).
12: \textbf{if} \( \ell \leq d \) \textbf{then}
13: \( s := s' \) \{ Accept the transition \}
14: \textbf{end if} \{ Otherwise implicitly reject the transition \}
15: \( T := T + \partial T \)
16: \textbf{end for}
17: \{ Drift is \( |T_E - T| \) \}

the advantage of reducing the time necessary to compute the transition to \( s' \). As we mentioned in Section 3 the amortized time to compute a transition is \( O(1 + (n + n_1 + \ldots + n_k)(1 + \log V)) \), where \( k = 1 - d \) and \( n \) is the degree of the node that gets inserted and \( n_1, \ldots, n_k \) the degrees of the nodes \( v'_1, \ldots, v'_\ell \) that get removed from the cycles. With this modification the overall amortized time of this transition reduces to \( O(1 + (n + n_1 + \ldots + n_{\ell+1})(1 + \log V)) \) because the process can be aborted as soon as the current iteration tries to remove \( 2 + \ell \) nodes. Note that the \( n_i \) values might be a loose upper bound because the algorithm iterates over the degrees of the vertexes in \( F \), which are bounded by the degrees of those vertexes in \( G \). To understand this process let us discuss exactly how it is computed with the LCT.

A full description of the LCT was provided by Sleator and Tarjan [5]. For our purposes the following overview will suffice. The link-cut tree operates by storing a set of edges, which forms a forest. Some of these edges are grouped into branches. Each tree also contains a designated root node. For the Forest represented in Figure 2 we could have, for example, the node 6 as the root and the nodes 6, 5, 8 corresponding to a branch in the tree. Each branch is stored in a binary search tree (BST), specifically a splay tree. Note that the order of the sequence of nodes is important, in particular nodes are stored by
increasing depth values. The edges inside a branch are stored inside a BST, such as \((5, 6)\) and \((8, 5)\). The edges that link different branches of the same tree are stored in a different way and are known as crossing edges. In this example \((4, 5)\) and \((2, 4)\) are crossing edges. Note that nodes 2 and 4 are also stored in BSTs but these BSTs are singular, i.e., contain no other nodes. This decomposition is altered with the \texttt{Access} operation. This operation reorganizes the BSTs so that the branch from \(v\) to the root becomes part of the current decomposition, where \(v\) is the accessed node. For example if we compute \texttt{Access}(4) in the current example we get the nodes 6, 5, 4 in a single branch and the nodes 2 and 8 in singular BSTs. It is also possible to change the root node with the \texttt{ReRoot} operation.

Let us get back to our application. Consider the transformation illustrated in Figure 3. In this transformation the node being inserted is \(v = 1\). We first consider its neighbors, which corresponds to the sets of nodes \(\{2, 4\}\). For each one of them we first check if \(v\) and its neighbor are already in the same tree. If so, adding the direct connection would result in a cycle. Therefore, in this case, we remove a vertex from the corresponding connecting branch. Otherwise the direct connection is simply added. Let us follow this procedure on the current example. We first check whether 1 and 2 are in the same tree. In the beginning this is not the case, therefore we execute a \texttt{Link} operation with arguments 1 and 2. We then move on to the next neighbor, which is node 4. This time the LCT identifies that nodes 1 and 4 are already in the same tree. Moreover it can isolate the corresponding branch into a BST, in this case 1, 2, 5, 4, assuming 1 is the root node. We now remove one node from this branch uniformly at random, in \(O(1 + \log V)\) amortized time. In this particular example the vertex selected was node 5 = \(v'\). An important nuance in this process is that the node \(v\) is not included in the removal procedure. This means that the probabilities are 1/3 for each of the nodes 2, 5, and 4. The reason for this choice is that removing node \(v\) would lead to a state \(s'\) where \(F' \subseteq F\), i.e., the new forest contained no new nodes.

Let us now consider how this process can be applied to the example in Figure 5. In this case the neighbors of \(v = 5\) are 2, 4, and 6. Assuming this order we first check if 5 and 2 are in the same tree. Since they are not, we issue a \texttt{Link} command with arguments 5 and 2. Next we process neighbor 4. This time querying the LCT indicates that 5 and 4 are in the same tree and the corresponding path is 5, 2, 1, 4. From this path \(v' = 2\) is selected, with 1/3 probability, and removed from the forest. This involves invoking two \texttt{Cut} operations for edges \(\{5, 2\}\) and \(\{2, 1\}\). The last operation involving
this neighbor is a Link command with arguments 5 and 4. Finally when processing the neighbor 6 the first query indicates that 5 and 6 are not in the same tree. Therefore a single Link command for edge \{5, 6\} is enough.

The process for Figure 6 is very similar. The key difference is that we now assume that the neighbors are ordered 2, 6 and 4. Hence we obtain the path 5, 2, 3, 6, from which we remove node \(v' = 3\) by issuing the Cut operations of the edges \{2, 3\} and \{3, 6\}. This process ends with the Link command for \(v = 5\) and 6. When we consider the neighbor 4 we again obtain a path, this time 5, 2, 1, 4. From this path we remove node \(v'' = 1\). In this case the Cut operations apply to edges \{2, 1\} and \{1, 4\}. As before the procedure ends with the Link command.

We can now discuss even more specific details, such as how the value \(\ell\) is computed in line 10.

4.1. Computing \(\ell\)

As discussed earlier each state \(s\) contains a specific energy value \(E(s)\). For our application this value is simply the number of nodes inside the respective forest \(F\). In a given step the algorithm changes from state \(s\) to state \(s'\), likewise the corresponding energy also changes to \(E(s') = E(s) + d\), for some variation value \(d\). If \(d\) is a negative number then vertexes can be removed from \(F\). If \(d\) is positive then vertexes must be added to \(d\). In the state transition process we described the maximal value for \(d\) is 1.

In general the parameter we supply to the algorithm is \(\bar{p} = 1 - p\), but to simplify the mathematical expressions we use \(p\) in the following discussion.

Our goal is to establish a relation between a probability \(p\) and a bound \(\ell\) such that any transition with a variation \(d\), such that \(\ell \leq d\), gets accepted. A simple such relation is the logistic function:

\[
p = \frac{1}{1 + 2^T \ell}
\]

Note that we use a base 2 exponential to match the base of the logarithms. The temperature parameter \(T\) is used to essentially change the base to \(2^T\), which can represent any number, with the appropriate value of \(T\). Figure 7 shows a simple plot of this function for \(T = 1\). Note that the point \(\ell = 0\) and \(p = 0.5\) belongs to this curve. This point can be interpreted as follows: the probability that a random number, selected by the annealng, accepts a transition for which \(0 = \ell \leq d\) is 50%. Likewise the point \(\ell = -1\) and \(p = 2/3\) is also part of this curve and therefore the probability that a random number
accepts a transition for which $-1 = \ell \leq d$ is at least 66%. We can see that as $d$ progresses to $-\infty$ this probability approaches 1.

This is a fairly good function except for two issues. The first one is that the point $\ell = 1$ and $p = 1/3$ is part of the curve. This point refers to transitions for which $1 = \ell \leq d$. In this application we always have that $d \leq 1$. Therefore this point relates directly to transitions for which $d = 1$. Because $1/3$ is slightly larger than 33% we can conclude that this point implies that transitions with $d = 1$ are reject with at least 33% probability. This is undesirable because, as we mentioned, there are no transitions with $d > 1$. This means rejecting transitions which are improving the current solution because they are not “good enough”. However no better transitions exist. This serves no useful purpose as it only amounts to wasted computer time. Moreover the next random number may decide to accept a transition that doesn’t even improve the current solution. Instead we set this probability to 0. This means that transitions for which $d = 1$ are always accepted.

We can manipulate the above expression to obtain that $\ell = \log((1/p) - 1)/T$, which leads to our second issue. In general this expression yields a real number. However the energy variation $d$ is always an integer. This means that somehow this expression is transformed to an integer. According to our previous exposition a transition is only accepted if $\ell \leq d$ this essentially
means that the expression is rounded towards $-\infty$. However this is an unbalanced choice, compared with rounding to the nearest integer. Recall that the point $\ell = 0$ and $p = 0.5$ means that transitions for which $d \geq 0$ are rejected with probability 50%. This is undesirable because it means that transitions with $d < 0$ are accepted with 50% probability, which is excessively high. Moreover this problem cannot be solved with different temperature values, as this point always exists in the function, no matter the value of $T \neq 0$. Recall that in our summary of the SA algorithm we separated transitions into three categories $d < 0$, $d = 0$ and $d > 0$. The rounding towards $-\infty$ approach groups $d = 0$ and $d > 0$ together. To fix this issue we compute $\ell$ by rounding $\log((1/p) - 1)/T$ to the nearest integer. An illustration of this procedure is shown in Figure 8. With this decision the point with $\ell = 0$ now has $p \approx 0.586$. Meaning that transitions for which $0 = \ell \leq d$ are accepted with more than 58% probability and therefore transitions for which the values of $d$ are negative are now accepted in less than 42% of the time. Moreover this probability is now sensitive to the temperature parameter $T$ and can therefore be made arbitrarily small. With this approach the $d = 0$ and $d > 0$ cases are no longer grouped together and in particular we get the nicely balanced property that the probability that a transition with $d < 0$ is accepted is equal to the probability that a transition with $d > 0$ is demanded. Note
that demanded is not the same as accepted, because, as explained above, transitions with \( d > 0 \) are always accepted.

This leads us to consider how to choose the temperature value, which is a fundamental parameter of the algorithm and is usually determined experimentally. However simply trying to guess a numerical value is likely to be error prone and not very informative. Instead we again manipulate the logistic expression. This time we solve it for \( T \). The result is given in Equation 1 and can be used to calculate a temperature from given \( p \) and \( \ell \) values.

\[
T = \log((1/p) - 1)/\ell
\]  

(1)

Consider determining a temperature for which a transition with \( 0 = [-0.5] = \ell \leq d \) is accepted with probability 80%, where \( d \) is the corresponding energy variation. We are using the notation \([−0.5] = 0\) to represent rounding to the nearest integer. The result should be even in case of ties, as in the example. To compute \( T \) we can substitute \( \ell = 0.5 \) and \( p = 0.8 \) into Equation (1), which yields \( T = 4.0 \). Note that using this logistic function the higher the value of \( T \) the stricter the acceptance criteria is. This means that, contrary to the classical annealing, the \( T \) values increase as the algorithm progresses. In other words the initial temperature is cooler than the finishing temperature. Note also that according to the previous discussion the value of \( \ell \) is always assumed to be an integer in lines 10 and 12 of Algorithm 4. It is only in the context of Equation (1) that we use decimal values for \( \ell \).

A final remark concerning the use of the expression \( \ell = \log((1/p) - 1)/T \) is how to use it to compute the value of \( \ell \). In theory we could use a random number \( r \) uniformly from \([0,1]\) and use it as the value of \( p \). In practice we use a random number \( R \) with 32 bits obtained with the \texttt{arc4random} function. Hence \( R \) is selected uniformly in the interval \([0,M]\), where \( M \) is the maximum value that can be represented in the computer. For example, in a 32 bit computer, \( M = 2^{32} - 1 \). We can now use \( p = R/(M+1) \) to compute the value of \( \ell \). To do this computation observe that \( \log((1/p) - 1) = \log(M+1-R) - \log(R) \) and that due to the two’s complement representation of negative numbers the value \( M+1-R \) can be obtained by interpreting the integer \(-R\) as unsigned. Hence the desired computation can be easily performed without using the value \( M \). Two special cases must be handled separately. If \( R = 0 \) then the expression is not valid. This falls into the very “strict” region, where only transitions with \( d >> 1 \) should be accepted. As we have seen these overly strict decisions are solved by setting \( \ell = 1 \). On the other extreme we have
that our procedure for obtaining \( p \) never yields \( p = 1 \) because \( R < M + 1 \). Again the expression is undefined for \( p = 1 \) but intuitively the value of \( \ell \) would tend to \(-\infty\), which meant accepting any transition, no matter how many elements got removed from the current solution. We actually prefer not to include this process and instead use random restarts when it is deemed necessary.

### 4.2. Greedy algorithm

We considered a greedy version of the algorithm we proposed. This allows us to properly evaluate the advantage of the SA algorithm. In particular the greedy algorithm will be much faster than SA but will obtain mostly local minimums whereas the SA algorithm can take longer to execute but most of the time approaches the global minimum value.

The greedy algorithms works as follows. We first select a random permutation of the vertexes \( V \). We then consider the vertexes \( v \in V \) in this order. We start with \( F = \emptyset \). At each step if \( F \cup \{v\} \) is still a forest, i.e., contains no induced cycles, then \( v \) is accepted into \( F \). Otherwise it is rejected and we consider the next vertex in \( V \). By construction the resulting set \( F \) is a forest and therefore \( V \setminus F \) is an FVS. To implement this greedy algorithm we could use the LCT data structure as before. However this variation never uses the \texttt{Cut} operation, which means that it can be implemented with a union-find data structure. An efficient implementation of the union-find data structure is faster than the LCT. It requires only \( O(\alpha(V)) \) amortized time per operation, see Tarjan and van Leeuwen [6], where \( \alpha \) is the inverse Ackermann function. This means that the execution time of the greedy algorithm is bounded by \( O(V + E\alpha(V)) \). The pseudo-code for this procedure is given in Algorithm 4.2.

Besides the union-find data structure we use an array \( A \) to verify that inserting \( v \) into \( F \) still yields a forest. The union-find data structure is also known as a disjoint sets data structure. The idea is that the structure represents a family of disjoint sets of elements of \( V \). The \texttt{Find}(\( v \)) operation returns the element that represents the set that contains \( v \). The representing elements are also in \( V \). Therefore the array \( A \) is assumed to be indexed over \( V \). The \texttt{Union}(\( v, v' \)) operation unites the sets that contain the elements \( v \) and \( v' \). The algorithm works by testing whether it is safe to add each element \( v \in V \) to \( F \). For each such vertex \( v \) the \texttt{for} cycle in line 6 checks if all the neighbors \( v' \) of \( v \) are in distinct sets. If so then it is safe to add \( v \) to \( F \), moreover the \texttt{for} cycle in line 14 then unites \( v \) and all its neighbors into the
1: Initialize union-find
2: Initialize all entries of array $A$ to nil \{A is indexed by elements in $V$\}
3: $F = \emptyset$
4: Permute the elements in $V$
5: for all $v \in V$ in the permutation order do
6: \hspace{1em} for all $v' \in V$ with $\{v, v'\} \in E$ do
7: \hspace{2em} if $A[\text{Find}(v')] = v$ then
8: \hspace{3em} Skip to the next iteration of the cycle in line 5
9: \hspace{2em} else
10: \hspace{3em} $A[\text{Find}(v')] = v$
11: \hspace{2em} end if
12: \hspace{1em} end for
13: \hspace{1em} $F = F \cup \{v\}$
14: \hspace{1em} for all $v' \in V$ with $\{v, v'\} \in E$ do
15: \hspace{2em} Union($v, v'$)
16: \hspace{1em} end for
17: end for

same set. Otherwise $v$ is not added to $F$ and the algorithm proceeds to the next element of $V$.

5. Experimental results

We tested the SA and greedy algorithms using the PACE 2016 dataset\(^3\). The initial description of this dataset was given by Kiljan and Pilipczuk [2]. We used the top two solvers from the competition to obtain the optimal values for some of the tests and reference execution times. We included both the public and private datasets. By analyzing the dataset we identified that tests 4, 5 and 116 of the hidden dataset contained vertexes with self-loops. We discarded test 5, as it contained too many such nodes. For tests 4 and 116 we edited the corresponding files to remove those vertexes.

The tests were executed in a server equipped with an Intel(R) Xeon(R) CPU E7- 4830 @ 2.13GHz. and 252 GiB of RAM running Linux 5.10.0-13-amd64. The SA and greedy algorithms are written in C and compiled with gcc 10.2.1 with optimization flags -O9. The first solver was written in java and

\footnote{https://github.com/ckomus/PACE-fvs.git}
compiled with javac 1.8.0_292 and compiled according to the instructions given by the authors\(^4\). The second solver was written in C++ and compiled with g++ 10.2.1 using the cmake system given by the author\(^5\).

We allocated two hours for each test and for each prototype. The complete dataset contained 229 test graphs. From these at least one of the PACE solvers obtained the optimal value for 169 graphs, within the two hour limit. For the remaining 60 graphs neither of the two solvers obtained a solution. In this Section we present some sample results. The complete set of graphs is given in Appendix A.

Figure 9 shows the results for test 048 of the public dataset. For this particular test the size of the minimal FVS is 119, this bound was obtained by the first solver in about 1 hour and 45 minutes. This minimum bound is represented by the horizontal thick green line in the plot. The key label of this MIN line indicates the value of the minimum, i.e., 119. The \(y\) axis represents the size of the current FVS. The \(x\) axis indicates how much time was used by the SA and greedy algorithms. As can be seen, in this particular

\(^4\)https://github.com/wata-orz/fvs.git

\(^5\)https://bitbucket.org/marcin-pilipczuk/fvs-pace-challenge.git
example the SA algorithm took around 12 milliseconds to obtain an FVS of size 127, i.e., an approximation ratio of around 1.07 times the optimal value. The execution of the greedy algorithm is even faster than the SA algorithm. This is represented by the gray lines, which are much more “vertical”. We execute the greedy algorithm several times and choose the best overall result, this is indicated by the gray horizontal line, which in this case obtains a minimal FVS of size 173. We use several sampling techniques to avoid overloading the plot. For example we do not show all the executions of the greedy algorithm, as this would result in too many gray lines. Instead we show one iteration out of 8. This skip value \((sk = 8)\) is given in the key’s text of the Greedy algorithm. The other value that is given in this text \(bp = 2\) indicates that this line does not contain values for every vertex \(v\), instead we sample the greedy algorithm’s execution time after processing every other vertex.

For the SA algorithm the plot contains points. Besides indicating the time and the size of the FVS in the current state the color of each point indicates the current temperature of the algorithm, but this value is mapped directly to the corresponding probability \(\bar{p}\) of accepting a transition that decreases the size of the current FVS. Naturally the red temperatures are hotter and the blue temperatures are cooler. We execute the SA algorithm for three probability ranges, from \([0.05, 0.01]\), \([0.01, 0.005]\) and \([0.005, 0.001]\). The results are overlaid in the plot in this order, i.e., red colors appear over blue ones. Moreover we repeat the execution of the SA algorithm three times for each one of these ranges. As in the greedy algorithm we use a batch size parameter \(bp\), in this case \(bp = 32\). This indicates that we only sample one information point in every 32 iterations. Moreover for this transition batch we keep the temperature value constant. This is a common technique in SA algorithms to reduce the amount of computation spent on updating temperatures. Besides the information points themselves we also use black lines to indicate the overall minimum value obtained in a given execution. Note that this minimum value might not correspond to an information point as this value is kept internally by the algorithm and checks every single state that it reaches. In other plots we are able to observe several such black lines, for example in the plot of the public test 026 shown in Figure 10.

Let us also consider some harder tests. For the test 27 of the hidden dataset the SA algorithm needed around 20 seconds to stabilize, whereas neither of the two exact solvers found the optimal solution within the two hour limit. Hence we do not actually know how far from the optimal solution
was the solution computed by the SA. The plot for this test is shown in Figure 11.

In the examples where the solvers found the size of the minimum FVS we compared those values against the size of the FVS found by the SA and the greedy algorithm. The worst approximation ratio was about 1.21 for the hidden test 93, where the optimal solution was 135 and SA obtained 164 and $1.21 \times 135 \approx 164$. The plot for this test is shown in Figure 12.

To give further insight into the approximation ratios of SA and greedy we show this data in bar charts. The green bar represents the size of the minimum FVS. The red bar the size of the FVS obtained by SA and the gray bar the size of FVS obtained by the repeated greedy algorithm. The bars are overlaid so the green bar obstructs the red one in such a way that the visible part of the red bar corresponds to the extra amount of the size of its FVS compared to the optimal value. Likewise the same happens between the red and gray bars. All the bars are scaled so that the green bar has size 1 as the goal is to illustrate the approximation ratio. The tests are ordered so that the results are in decreasing value of the SA approximation ratio. A blue line indicates the 2 approximation ratio guaranteed by the algorithm of Bafna, Berman, and Fujito [7]. In Figure 13 we show these results for the first 50 tests, the remaining results are shown in the appendix.
Figure 11: Results for hidden test 27.

Figure 12: Results for hidden test 93.
Figure 13: Approximation Ratios for SA and greedy.
The last set of data that we need to present concerns the time performance of the SA algorithm. The time difference between the SA algorithm and the exact solver can be huge, as in the example of the public test 048. However this is not always the case. Sometimes it is the case that the exact solvers are faster than the SA algorithm. We show, some, of these results in Table 1. Each entry in the table corresponds to a test input. To explain how the information is organized let us consider the first entry, i.e., 041-pub. The time the SA algorithm used in this test was 61 seconds. The fastest exact solver used 0.18 seconds to solve this problem. In this particular case it was solver 1 whereas solver 2 also solved the problem and in 0.41 seconds. The table contains only the time of the fastest solver. Hence the resulting slowdown ratio is about $335 \approx 61/0.18$. The entries in the table are sorted by this value, so this was the absolute worst ratio for SA. To indicate that SA was the slowest algorithm we wrote the name of the entry in red. Likewise once the ratio drops below 1 the exact solvers become slower than the SA algorithm and we write the corresponding labels in green to indicate this fact. The first sample for which the ratio was smaller than 1 was 7-hid. As can be seen the time ratio drops fairly quickly in the table, the last shown value is 0.05. Moreover this table contains only 74 entries, which means that for the remaining 95 tests the ratio was even smaller. On top of that there are also the 60 tests for which neither of the exact solvers finished within 2 hours. Note that the longest any SA test took was 103 seconds. Therefore these 60 tests have a time ratio that is guaranteed to be smaller than 0.02.

6. Related work

In this section we briefly survey the related work on the undirected FVS problem and the simulated annealing meta-heuristic and the link cut tree data structure. The FVS problem is NP-hard; for directed graphs Karp showed its NP-completeness even if graphs are unweighted [1], and essentially the same transformation shows that it is equally hard for undirected graphs.

The algorithm that obtained first place in the PACE 2016 challenge was proposed by Iwata, Wahlström, and Yoshida [8] [9]. The authors designed a linear-time kernelization algorithm that, given an undirected graph $G$ of $m$ edges and an integer $k$, computes a graph $G'$ and an integer $k'$ in $O(k^4 m)$ time such that (1) the size of the graph $G'$ is $O(k^2)$, (2) $k' \leq k$, and (3) $G$ has a FVS of size at most $k$ if and only if $G'$ has a FVS of size at most $k'$. The size of the kernel is $2k^2 + k$ vertices and $4k^2$ edges, which is smaller than
| 041-pub | 335.00 ≈ 6.1e+01 / 1.8e-01 | 108-hid | 138.03 ≈ 1.5e+01 / 1.1e-01 |
| 045-pub | 74.64 ≈ 3.1e+01 / 4.1e-01 | 107-hid | 70.14 ≈ 7.2e+00 / 1.0e-01 |
| 038-pub | 46.26 ≈ 1.6e+01 / 3.6e-01 | 130-hid | 30.94 ≈ 1.8e+00 / 5.9e-02 |
| 001-pub | 22.09 ≈ 3.0e+00 / 1.4e-01 | 074-pub | 19.96 ≈ 1.7e+01 / 8.7e-01 |
| 080-pub | 14.18 ≈ 2.5e+00 / 1.8e-01 | 073-pub | 14.14 ≈ 3.4e+00 / 2.4e-01 |
| 129-hid | 11.56 ≈ 6.4e-01 / 5.5e-02 | 64-hid | 8.75 ≈ 2.0e+00 / 2.3e-01 |
| 45-hid | 6.44 ≈ 1.6e+00 / 2.5e-01 | 3-hid | 4.86 ≈ 1.5e-01 / 3.0e-02 |
| 037-pub | 4.24 ≈ 7.7e+01 / 1.8e+01 | 65-hid | 3.73 ≈ 6.2e-01 / 1.7e-01 |
| 019-pub | 3.56 ≈ 3.0e-01 / 8.5e-02 | 128-hid | 3.44 ≈ 9.3e-02 / 2.7e-02 |
| 60-hid | 3.29 ≈ 7.4e+00 / 2.2e+00 | 057-pub | 2.01 ≈ 4.2e-01 / 2.1e-01 |
| 66-hid | 1.60 ≈ 1.3e+00 / 8.1e-01 | 59-hid | 1.50 ≈ 7.7e+00 / 5.2e+00 |
| 58-hid | 1.20 ≈ 7.6e+00 / 6.3e+00 | 39-hid | 1.18 ≈ 1.4e+01 / 1.2e+01 |
| 085-pub | 1.07 ≈ 5.0e-02 / 4.7e-02 | 7-hid | 0.69 ≈ 3.8e-01 / 5.5e-01 |
| 118-hid | 0.61 ≈ 3.8e-02 / 6.3e-02 | 002-pub | 0.59 ≈ 2.9e-02 / 4.9e-02 |
| 032-pub | 0.47 ≈ 8.7e-01 / 1.8e+00 | 38-hid | 0.46 ≈ 1.3e+01 / 2.9e+01 |
| 12-hid | 0.42 ≈ 1.2e-02 / 2.9e-02 | 049-pub | 0.37 ≈ 1.5e-02 / 3.9e-02 |
| 62-hid | 0.35 ≈ 7.5e+00 / 2.2e+01 | 127-hid | 0.33 ≈ 4.9e-03 / 1.5e-02 |
| 97-hid | 0.31 ≈ 3.4e-02 / 1.1e-01 | 61-hid | 0.31 ≈ 8.0e+00 / 2.6e+01 |
| 106-hid | 0.30 ≈ 7.5e+03 / 2.5e-02 | 37-hid | 0.27 ≈ 4.6e-03 / 1.7e-02 |
| 031-pub | 0.27 ≈ 6.2e-03 / 2.3e-02 | 014-pub | 0.24 ≈ 1.2e+00 / 5.1e+00 |
| 126-hid | 0.21 ≈ 4.9e-03 / 2.3e-02 | 006-pub | 0.20 ≈ 2.1e-02 / 1.1e-01 |
| 113-hid | 0.20 ≈ 4.7e+03 / 2.4e-02 | 124-hid | 0.18 ≈ 3.3e-03 / 1.8e-02 |
| 36-hid | 0.16 ≈ 1.8e-02 / 1.1e-01 | 116-hid | 0.16 ≈ 5.9e-03 / 3.7e-02 |
| 83-hid | 0.16 ≈ 3.1e+03 / 2.0e-02 | 4-hid | 0.15 ≈ 3.5e-03 / 2.3e-02 |
| 88-hid | 0.15 ≈ 5.7e+03 / 3.8e-02 | 070-pub | 0.14 ≈ 2.7e-03 / 1.9e-02 |
| 122-hid | 0.13 ≈ 3.3e-03 / 2.6e-02 | 81-hid | 0.12 ≈ 5.2e-03 / 4.2e-02 |
| 69-hid | 0.12 ≈ 1.2e+02 / 9.5e-02 | 112-hid | 0.12 ≈ 2.4e-03 / 2.0e-02 |
| 022-pub | 0.11 ≈ 1.8e+01 / 1.7e+02 | 042-pub | 0.11 ≈ 2.2e-03 / 2.1e-02 |
| 18-hid | 0.10 ≈ 3.0e+03 / 2.9e-02 | 072-pub | 0.10 ≈ 2.3e-03 / 2.2e-02 |
| 117-hid | 0.10 ≈ 1.8e+03 / 1.7e-02 | 077-pub | 0.10 ≈ 3.2e-03 / 3.3e-02 |
| 024-pub | 0.09 ≈ 5.4e+03 / 5.8e-02 | 098-pub | 0.09 ≈ 1.4e+03 / 1.6e-02 |
| 21-hid | 0.08 ≈ 4.1e+03 / 4.9e-02 | 091-pub | 0.08 ≈ 4.8e-03 / 5.9e-02 |
| 120-hid | 0.08 ≈ 1.1e+03 / 1.4e-02 | 015-pub | 0.08 ≈ 3.6e-03 / 4.8e-02 |
| 065-pub | 0.08 ≈ 1.5e+03 / 2.0e-02 | 115-hid | 0.07 ≈ 1.6e-03 / 2.2e-02 |
| 062-pub | 0.06 ≈ 1.6e+03 / 2.6e-02 | 114-hid | 0.06 ≈ 1.4e-03 / 2.3e-02 |
| 16-hid | 0.06 ≈ 8.3e+03 / 1.4e-01 | 003-pub | 0.06 ≈ 1.8e-03 / 3.0e-02 |
| 093-pub | 0.06 ≈ 8.2e+03 / 1.4e-01 | 007-pub | 0.05 ≈ 2.3e-03 / 4.2e-02 |

Table 1: Time results and ratio between SA and exact solvers.

24
the previous best of $4k^2$ vertices and $8k^2$ edges. Thus improving the size and the running time simultaneously. Note that under the assumption that NP $\not\subseteq$ coNP/poly, FVS does not admit an $O(k^{2-\epsilon})$-size kernel for any $\epsilon > 0$.

The kernel used a $k$-submodular relaxation, which is a technique for obtaining efficient FPT algorithms for various problems. The dual of $k$-submodular relaxation of FVS can be seen as a half-integral variant of A-path packing, and to obtain the linear-time complexity, the author used an efficient augmenting-path algorithm.

The algorithm that obtained second place was implemented by Marcin Pilipczuk and it was an algorithm designed for the challenge. The algorithm involved the following techniques:

**Basic reduction rules** Reducing loops and vertexes of degree at most two; contracting two adjacent undeletable vertices; Adding to the solution a vertex adjacent to an undeletable vertex with a multiple edge; if the maximum degree of a yet undecided vertex drops to three, a polynomial-time solution is invoked;

**Basic branching** The basic branching algorithm is a branch first on a double edge, if present, and if not, on the highest degree yet undecided vertex.

**Initial solution** First the algorithm finds an approximate solution using a simple heuristic, and then runs branching in $2^k$ time, with $k$ being the size of the solution found by the heuristic.

**Extra Processing** A degree-3 vertex incident to a double edge can be made undeletable; A simple reduction of false twins; Branching on independent 2-connected components.

**Tree decompositions** Compute an approximation of the treewidth. If this number is small a standard dynamic programming algorithm is used.

The author reported that this prototype outperformed a previous algorithm [10]. We did not use the remaining prototypes, as they solved fewer instances. Descriptions and links to the respective implementation can be found in the challenge’s web page

---

Considering approximation algorithms the first nontrivial approximation ratio of $2 \log n$ for unweighted graphs appeared in the early work of Dirac and Erdős [11], where they studied the number of (vertex) disjoint cycles in a graph. It was later improved to $\sqrt{\log n}$ by Monien and Schulz [12], who considered and compared various approaches to the problem. Later Bar-Yehuda, Geiger, Naor, and Roth [13] showed that the smallest cardinality FVS (i.e., unweighted version) can be approximated within a constant factor of 4. Moreover, they considered the weighted version as well and obtained a performance ratio of $\min\{4 \log n, 2d^2\}$, where $d$ is the maximum vertex degree of a graph.

Currently the best FVS approximation ratio is 2, proposed by Bafna, Berman, and Fujito [7]. This algorithm was implemented by several of the solvers. An implementation of this algorithm requires at most $O(E \log V)$ time (or $O(V^2)$ in dense graphs), which makes it very efficient. In fact it implies that a fast alternative to the greedy algorithm exists. We still use the greedy algorithm in our experimental setup as the goal is to determine how relevant is the SA heuristic. Hence we compare it with a less flexible version, the greedy algorithm.

A wide range of reduction rules for the FVS problem was proposed by Bodlaender and van Dijk [14]. The authors showed that the iterated application of these rules yields an $O(k^3)$ kernel for the problems. This result was further improved to an $O(k^2)$ kernel by Thomassé [15].

The link-cut tree data structure played a fundamental role in our algorithm. This data structure was proposed by Sleator and Tarjan [5]. Our implementation is based on splay trees, as latter described by Sleator and Tarjan [16]. There are other data structures that can be used to efficiently detect cycles in undirected graphs, such as Euler tour trees [17]. The Euler tour tree data structure is a tree representation that encodes the structural information of a rooted tree into a sequence of integers. Specifically, it performs a depth-first traversal of the tree and assigns a unique integer value to each vertex based on the order in which it is visited. The resulting sequence is known as the Euler tour sequence, which captures the nesting of the tree nodes and their relationships to each other. This representation allows the ETT to detect that there already is a path between a pair of nodes of the represented tree. However this path is not necessarily the smallest path, instead it will contain all the nodes in the sub-tree between the two given nodes. In essence this means that the ETT is equally efficient at detecting cycles but is not suitable to identify the nodes of the cycle. Hence link-cut trees are
better suited for our application, since our procedure involves identifying the corresponding cycle and removing a vertex from it.

Simulated annealing is a probabilistic technique for approximating the global optimum of a given function. Specifically, it is a meta-heuristic. It is often used in discrete problems. The inspiration for this technique comes from annealing in metallurgy, a technique involving heating and controlled cooling of a material to alter its physical properties. The first application of this technique to the traveling salesman problem was proposed by Kirkpatrick, Gelatt, and Vecchi [18], Černý [19]. An Application of SA to the FVS problem was proposed by Galinier, Lemamou, and Bouzidi [3] and further improved by Tang, Feng, and Zhong [4] and Russo, Castro, Ilic, Romano, and Correia [20]. These algorithms are similar to the one we are proposing. At each step they also consider modifying the current configuration by inserting a new vertex into the current solution $V'$. However these approaches are for the version of the problem over directed graphs. This means that it is not possible to determine the corresponding cycles by using something like the link-cut tree. Therefore the removal process in these algorithms is not as efficient as in our approach. We have also used the LCT in combination with SA before in the context of data compression [21].

7. Conclusions and future work

In this paper we proposed an approach to the feedback vertex set problem of an undirected graph. Our algorithm is based on the simulated annealing meta-heuristic. A fundamental part of this algorithm is the use of the link-cut tree data structure. In fact the application of this data structure to the FVS problem was the most fundamental idea we wish to convey. It seems to us that the LCT is particularly well suited for the FVS problem over undirected graphs because it provides and extremely efficient way to detect cycles. We expect that the LCT will be incorporated into other algorithms related to this problem, namely in combinatorial algorithms that search for an exact solution. We plan to look into this possibility in future works.

We tested our algorithm experimentally, using the dataset from the PACE 2016 challenge. The results we obtained showed that in general our algorithm is very efficient. Most of the time the algorithm obtained the optimal solution, or was very close to the optimal solution. The worst approximation ratio obtained was about 1.21. Most of the time the algorithm was extremely fast. The longest execution time was 103 seconds. For a small set of the tests the
exact solvers were faster than the SA algorithm, but for the vast majority
of cases the SA algorithm was faster, see Table 1. Our prototype was very
simple, in the sense that it did not use any reduction rules of the initial
graph, such as the ones by Bodlaender and van Dijk [14]. Clearly including
such optimizations is likely to further improve the approximation ratio.

We also proposed a greedy algorithm, based on the union-find data struc-
ture. The purpose of this algorithm was not to be competitive with state of
the art solutions. In particular there were several tests for which the greedy
algorithm had an approximation ratio larger than 2. Instead this algorithm is
used to justify the necessity of an SA approach, as the greedy algorithm gets
stuck in local optimum solutions. Still for some tests the greedy algorithm
did obtained good approximation ratios quickly. In a future work we plan
to investigate the viability of using the union-find data structure in existing
algorithms for the FVS.

Acknowledgments

The work reported in this article was supported by national funds through
Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020,
project NGPHYLO PTDC/CCI-BIO/29676/2017.

Appendix A. Complete results

Here we present the results from the experimental setup that were too
extensive for the main text of the article.

Figures A.14, A.15 and A.16 show the remaining approximation ratios.
Like in Figure 13 the tests are ordered so that the results are in decreasing
value of the SA approximation ratio. Furthermore when this ratio is 1 the
tests are ordered by the approximation ratio of the greedy algorithm.

Tables A.2 and A.3 show the remaining time ratios. The remaining Fig-
ures show the performance of the SA and greedy algorithms for each of the
tests in the PACE dataset.
Figure A.14: Approximation Ratios for SA and greedy.
Figure A.15: Approximation Ratios for SA and greedy.
Figure A.16: Approximation Ratios for SA and greedy.
| 020-pub  | 0.05 ≈ 2.0e-03/3.8e-02 | 076-pub  | 0.05 ≈ 3.9e-03/7.7e-02 |
| 123-hid  | 0.05 ≈ 1.2e-03/2.3e-02 | 84-hid   | 0.05 ≈ 1.1e-03/2.2e-02 |
| 87-hid   | 0.05 ≈ 5.2e-03/1.1e-01 | 018-pub  | 0.05 ≈ 2.5e-02/5.1e-01 |
| 095-pub  | 0.05 ≈ 9.9e-04/2.1e-02 | 099-pub  | 0.05 ≈ 6.4e-04/1.4e-02 |
| 059-pub  | 0.05 ≈ 4.1e-03/8.8e-02 | 096-pub  | 0.04 ≈ 9.0e-04/2.0e-02 |
| 125-hid  | 0.04 ≈ 1.1e-03/2.4e-02 | 083-pub  | 0.04 ≈ 9.2e-04/2.1e-02 |
| 040-pub  | 0.04 ≈ 1.5e+01/3.7e+02 | 99-hid   | 0.04 ≈ 1.9e-03/4.8e-02 |
| 17-hid   | 0.04 ≈ 8.3e-04/2.3e-02 | 86-hid   | 0.03 ≈ 1.8e-03/5.1e-02 |
| 092-pub  | 0.03 ≈ 3.1e-03/9.0e-02 | 030-pub  | 0.03 ≈ 9.6e-04/2.8e-02 |
| 2-hid    | 0.03 ≈ 1.1e-03/3.3e-02 | 63-hid   | 0.03 ≈ 9.4e+00/2.8e+02 |
| 73-hid   | 0.03 ≈ 3.6e-03/1.1e-01 | 028-pub  | 0.03 ≈ 7.6e-04/2.3e-02 |
| 111-hid  | 0.03 ≈ 8.2e-04/2.5e-02 | 85-hid   | 0.03 ≈ 6.4e-04/2.0e-02 |
| 70-hid   | 0.03 ≈ 2.0e-01/6.3e+00 | 119-hid  | 0.03 ≈ 6.2e-04/2.0e-02 |
| 121-hid  | 0.03 ≈ 6.2e-04/2.0e-02 | 005-pub  | 0.03 ≈ 3.4e-03/1.2e-01 |
| 90-hid   | 0.03 ≈ 1.1e-02/4.2e-01 | 050-pub  | 0.02 ≈ 5.4e-04/2.3e-02 |
| 009-pub  | 0.02 ≈ 5.0e-03/2.2e-01 | 046-pub  | 0.02 ≈ 2.5e-03/1.1e-01 |
| 15-hid   | 0.02 ≈ 1.1e-03/5.0e-02 | 060-pub  | 0.02 ≈ 5.2e-03/2.5e-01 |
| 044-pub  | 0.02 ≈ 2.7e-03/1.3e-01 | 012-pub  | 0.02 ≈ 7.6e-03/3.8e-01 |
| 75-hid   | 0.02 ≈ 1.7e-03/1.0e-01 | 77-hid   | 0.02 ≈ 3.3e-03/2.0e-01 |
| 069-pub  | 0.02 ≈ 2.7e-02/1.6e+00 | 74-hid   | 0.02 ≈ 1.3e-03/8.1e-02 |
| 19-hid   | 0.01 ≈ 2.0e-03/1.4e-01 | 57-hid   | 0.01 ≈ 2.1e-03/2.2e-01 |
| 71-hid   | 0.01 ≈ 5.6e-02/6.1e+00 | 56-hid   | 0.01 ≈ 2.1e-03/2.3e-01 |
| 025-pub  | 0.01 ≈ 1.2e-01/1.5e+01 | 72-hid   | 0.01 ≈ 7.0e-04/8.3e-02 |
| 026-pub  | 0.01 ≈ 5.6e-03/6.8e-01 | 20-hid   | 0.01 ≈ 6.3e-03/7.9e-01 |
| 98-hid   | 0.01 ≈ 2.4e-02/3.1e+00 | 011-pub  | 0.01 ≈ 9.0e-03/1.2e+00 |
| 017-pub  | 0.01 ≈ 2.3e-02/3.5e+00 | 023-pub  | 0.01 ≈ 2.6e-02/4.3e+00 |
| 80-hid   | 0.01 ≈ 3.2e-02/5.7e+00 | 76-hid   | 0.01 ≈ 1.4e-02/2.7e+00 |
| 6-hid    | 0.01 ≈ 9.1e-02/1.8e+01 | 033-pub  | 0.00 ≈ 1.8e-02/3.9e+00 |
| 087-pub  | 0.00 ≈ 1.9e-02/4.1e+00 | 027-pub  | 0.00 ≈ 1.3e-03/3.0e-01 |
| 68-hid   | 0.00 ≈ 4.0e-03/9.4e-01 | 056-pub  | 0.00 ≈ 2.4e-01/5.7e+01 |
| 029-pub  | 0.00 ≈ 1.4e-03/3.5e-01 | 084-pub  | 0.00 ≈ 3.4e+00/8.9e+02 |
| 013-pub  | 0.00 ≈ 8.5e-03/2.5e+00 | 100-pub  | 0.00 ≈ 2.7e-02/9.0e+00 |
| 035-pub  | 0.00 ≈ 1.6e-02/6.5e+00 | 67-hid   | 0.00 ≈ 1.2e-02/5.5e+00 |
| 79-hid   | 0.00 ≈ 1.7e-02/9.2e+00 | 067-pub  | 0.00 ≈ 7.1e-03/4.2e+00 |
| 008-pub  | 0.00 ≈ 2.7e-01/1.9e+02 | 22-hid   | 0.00 ≈ 1.5e-02/1.2e+01 |
| 097-pub  | 0.00 ≈ 2.6e-03/2.3e+00 | 034-pub  | 0.00 ≈ 7.0e-03/7.4e+00 |
| 1-hid    | 0.00 ≈ 3.4e-03/3.7e+00 | 021-pub  | 0.00 ≈ 2.5e-01/2.7e+02 |

Table A.2: Time results and ratio between SA and exact solvers.
Table A.3: Time results and ratio between SA and exact solvers.

<table>
<thead>
<tr>
<th></th>
<th>FVS</th>
<th></th>
<th>Greedy(sk=8,bp=2)</th>
<th>SA-MIN</th>
<th>SA(bp=32)</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>086-pub</td>
<td>0.00≈3.0e-03/3.4e+00</td>
<td>010-pub</td>
<td>0.00≈3.6e-02/4.2e+01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>052-pub</td>
<td>0.00≈1.8e-01/7.5e+02</td>
<td>23-hid</td>
<td>0.00≈6.9e-03/3.2e+01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>95-hid</td>
<td>0.00≈1.6e-02/1.3e+02</td>
<td>066-pub</td>
<td>0.00≈1.0e-02/1.2e+02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>051-pub</td>
<td>0.00≈3.3e-03/5.1e+01</td>
<td>043-pub</td>
<td>0.00≈2.6e-02/5.6e+02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>82-hid</td>
<td>0.00≈1.0e-02/2.2e+02</td>
<td>41-hid</td>
<td>0.00≈3.4e-03/8.4e+01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>51-hid</td>
<td>0.00≈2.0e-02/5.8e+02</td>
<td>91-hid</td>
<td>0.00≈6.1e-03/1.9e+02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35-hid</td>
<td>0.00≈8.3e-02/3.9e+03</td>
<td>93-hid</td>
<td>0.00≈1.3e-02/6.9e+02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>102-hid</td>
<td>0.00≈5.2e-02/3.8e+03</td>
<td>109-hid</td>
<td>0.00≈2.5e-03/3.1e+02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53-hid</td>
<td>0.00≈2.1e-02/3.3e+03</td>
<td>047-pub</td>
<td>0.00≈1.6e-03/2.5e+02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>081-pub</td>
<td>0.00≈2.9e-02/6.2e+03</td>
<td>24-hid</td>
<td>0.00≈1.6e-02/3.7e+03</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>048-pub</td>
<td>0.00≈1.1e-02/6.3e+03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Greedy (sk=8, bp=2)
SA-MIN
Min
063-pub

FVS

Greedy(sk=8,bp=2)  SA-MIN  SA(bp=32)

064-pub

FVS

Greedy(sk=8,bp=2)  SA-MIN  SA(bp=32)

065-pub

FVS

Greedy(sk=8,bp=2)  SA-MIN  Min  SA(bp=32)

54
093-pub

094-pub

095-pub

64
9-hid

Greedy (sk=128, bp=2)  SA-MIN  SA (bp=512)

10-hid

Greedy (sk=128, bp=2)  SA-MIN  SA (bp=1024)

11-hid

Greedy (sk=64, bp=2)  SA-MIN  SA (bp=512)
30-hid

Greedy\( (sk=128,bp=2) \)  \hspace{1cm} \text{SA-MIN} \hspace{1cm} \text{SA}(bp=512)

31-hid

Greedy\( (sk=64,bp=2) \)  \hspace{1cm} \text{SA-MIN} \hspace{1cm} \text{SA}(bp=512)

32-hid

Greedy\( (sk=64,bp=2) \)  \hspace{1cm} \text{SA-MIN} \hspace{1cm} \text{SA}(bp=256)
33-hid

Greedy (sk=64, bp=2) | SA-MIN | SA (bp=256)

34-hid

Greedy (sk=32, bp=2) | SA-MIN | SA (bp=256)

35-hid

Greedy (sk=8, bp=2) | SA-MIN | SA (bp=128) | Min

77
0 20 40 60 80 100 120

0.002
0.004
0.008
0.016
0.032

0 0.5 1 1.5 2 2.5 3 3.5

Greedy(sk=8,bp=2) SA-MIN SA(bp=16) Min

57-hid

0 1 2 3 4 5 6 7 8 9

FVS

Greedy(sk=256,bp=2) SA-MIN SA(bp=1024) Min

58-hid

0 1 2 3 4 5 6 7 8 9

FVS

Greedy(sk=256,bp=2) SA-MIN SA(bp=1024) Min

59-hid

0 1 2 3 4 5 6 7 8 9

FVS

85
66-hid

Greedy (sk=256, bp=2)  SA-MIN  SA (bp=512)
Min

67-hid

Greedy (sk=8, bp=2)  SA-MIN  SA (bp=32)
Min

68-hid

Greedy (sk=8, bp=2)  SA-MIN  SA (bp=32)
Min
87-hid

Greedy (sk=8, bp=2)  SA-MIN  SA (bp=32)  Min

88-hid

Greedy (sk=8, bp=2)  SA-MIN  SA (bp=32)  Min

89-hid

Greedy (sk=8, bp=2)  SA-MIN  SA (bp=32)  Min
FVS

114-hid

Greedy(sk=8,bp=2)  time (ms)  SA-MIN  SA(bp=32)  Min

115-hid

Greedy(sk=8,bp=2)  time (ms)  SA-MIN  SA(bp=32)  Min

116-hid

Greedy(sk=8,bp=2)  time (ms)  SA-MIN  SA(bp=32)  Min
123-hid

Greedy (sk=4, bp=2)  SA-MIN  SA (bp=8)  Min

124-hid

Greedy (sk=4, bp=2)  SA-MIN  SA (bp=16)  Min

125-hid

Greedy (sk=8, bp=2)  SA-MIN  SA (bp=32)  Min
References


