Applied Soft Computing 129 (2022) 109607

Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

Stochastic simulated annealing for directed feedback vertex set

Luís M.S. Russo *, Daniel Castro, Aleksandar Ilic, Paolo Romano, Ana D. Correia

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

ARTICLE INFO

Article history:
Received 17 November 2021
Received in revised form 26 August 2022
Accepted 1 September 2022
Available online 13 September 2022

Keywords:
Feedback vertex set
Maximum directed acyclic graph
Simulated annealing
Cycle detection heuristic
Complexity theory

ABSTRACT

The minimum feedback vertex set problem consists in finding the minimum set of vertices that should be removed in order to make the graph acyclic. This is a well known NP-hard optimization problem with applications in various fields, such as VLSI chip design, bioinformatics and transaction processing. In this paper, we explore the complementary problem in directed graphs, i.e., how to construct the maximum directed acyclic graph (max-DAG). We show that the max-DAG problem is Poly-APX complete, which implies that even trying to obtain approximation algorithms for this problems is likely to be unfeasible. In light of these considerations, we introduce a new algorithmic solution, based on Simulated Annealing (SA), which combines techniques such as kernelization, efficient data-structures, novel heuristics to initialize the search process, a global re-structuring procedure, and a neighbor re-ordering technique to speed-up the local search step. We present an extensive experimental study that validates the key design and implementation choices undertaken in our proposal and compares it to state of the art SA-based solutions Galinier et al. (2013) and Tang et al. (2017). The proposed algorithm provides significant performance gains by obtaining feedback vertex sets up to 13.3× closer to the optimal solution in a wide variety of synthetic and real-world graphs.

Code metadata

Permanent link to reproducible Capsule: https://doi.org/10.24433/CO.1016608.v1.

1. Introduction

In this paper, we study the Feedback Vertex Set (FVS) problem for directed graphs. An FVS consists of vertices that belong to any cycle of the graph. Note that removing an FVS from the graph makes it acyclic. In this work, we focus on determining the maximum sub-graph that is a Directed Acyclic Graph (DAG). As such, we consider the dual formulation of the FVS, in particular the dual of finding the minimum FVS.

This problem has applications in several areas, including program verification [1], bioinformatics and biology [2,3], statistical mechanics [4], deadlock resolution [5], Bayesian inference [6] and VLSI chip design [7]. For example, efficient solutions to this problem are important for hardware synthesis [8], where cyclic combinational relations should not occur among streaming sources and sinks (such as the one illustrated in Fig. 1). Such cycles result in undesirable behaviors or un-synthesizable designs. Recently, a novel approach for modeling automated storage/retrieval systems was proposed by Gharehgozli et al. [9], which relies on high multiplicity asymmetric traveling salesman problem with FVS to sequence the storage and retrieval requests to minimize their completion time.

Our main motivation is the application to concurrency control of transactional systems, such as Transactional Memory (TM) systems [10] and databases [11] — in which case, the induced DAG and underlying topological order can be exploited to ensure that concurrent execution is equivalent to a serial one.

The FVS problem is NP-Hard [12–14] and a survey on approximation algorithms was given by Festa et al. [7]. In this paper, we follow the approach started by Galinier et al. [15] of applying a local search technique in a Simulated Annealing (SA) algorithm. We also consider the improvements by Tang et al. [16]. The approach proposed herein advances the existing SA-based methods by using a new fast heuristic to determine whether a vertex can be safely added to the current configuration without generating cycles. We discuss in detail data structures and optimization techniques that guarantee the efficiency of this approach. This stochastic selection is significantly faster than state of the art approaches that rank all possible transitions.

The code (and data) in this article has been certified as Reproducible by Code Ocean: (https://codeocean.com/). More information on the Reproducibility Badge Initiative is available at https://www.elsevier.com/physical-sciences-and-engineering/computer-science/journals.

* Corresponding author.
E-mail addresses: luis.russo@tecnico.ulisboa.pt (L.M.S. Russo), daniel.castro@ist.utl.pt (D. Castro), aleksandar.ilic@inesc-id.pt (A. Ilic), romano@inesc-id.pt (P. Romano), ana.duarte.correia@tecnico.ulisboa.pt (A.D. Correia).

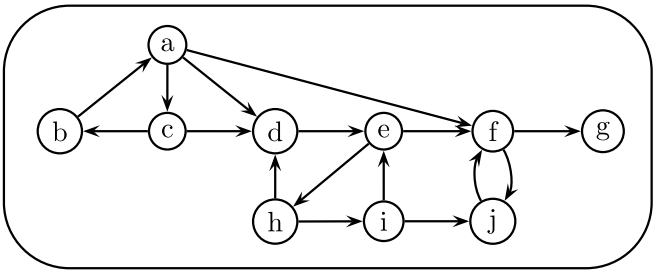
https://doi.org/10.1016/j.asoc.2022.109607

The extensive experimental evaluation shows that the proposed approach is capable of finding feedback vertex sets up to 13.3× closer to the optimal, in a variety of real-world and synthetic graphs when compared to state of the art solutions based on SA.

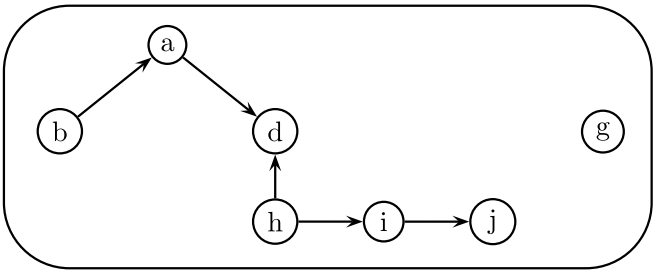
The remainder of this article is structured as follows:

- We start with an initial pre-processing step, Section 3, which separates the initial graph into Strongly Connected Components (SCCs). This acts as a form of kernelization. In some graphs the resulting components are so small that this initial step largely solves the problem, or it makes it possible to apply combinatorial approaches, such as Branch and Bound (BB). We also propose a hybrid algorithm that selects between SA and BB, thus obtaining a result that is the best of both worlds.
- We review the complexity of the underlying problem. In particular, we review its NP-Complete classification, see Theorem 1. The main objective of this revision is to establish that the max-DAG problem is Poly-APX-complete, see Corollary 1. This result implies that even trying to obtain approximation algorithms for this problem is likely to be unfeasible. Instead, we discuss several techniques that obtain good practical results. In particular, we propose a local search procedure and highlight some of its limitations.
- We discuss several data structure design and implementation choices, see Section 5.1. In particular, we show how to represent a DAG such that the local optimization procedure of adding a vertex or removing vertices can be implemented efficiently. Our implementation of this procedure, uses splay trees and several speedup techniques. We also discuss techniques for choosing the initial state and for allocating search time among smaller sub-graphs, which culminates in the aforementioned hybrid algorithm. We also use a global restructuring procedure and a neighbor re-ordering heuristic to boost the performance of the local search procedure.
- We present extensive experimental results in Section 6, based both on real world graphs and on synthetic graphs, which evaluate the relative performance of these algorithms and compare them to state of the art solutions based on SA. The experimental results also validate several of the data structure design and implementation choices. In particular, when compared to the other SA algorithms, our algorithm finds feedback vertex sets up to 13.3× closer to the optimal in a wide variety of graphs. Moreover, we investigate the impact of the temperature parameter in the performance of our SA algorithm. Our results show that choosing relatively cold temperatures tends to yield better results (i.e., quick convergence to global optima) in the majority of the graphs tested. However a hotter temperature setting can be beneficial in dealing with an adversarial class of graphs, which we designed to challenge approaches based on local search methods.

The remainder of this paper is structured as follows: Section 2 formulates the max-DAG problem; Section 3 provides a general overview of the proposed algorithmic contributions; Section 4 gives some background on the SA algorithm, along with a literature survey and an NP-Complete classification of the max-DAG problem; Section 5 discusses an efficient implementation of the proposed approach. The results of extensive experimental campaign are reported in Section 6. Section 7 concludes the paper and elaborates future research directions.



(a) Some input graph



(b) A max-DAG for the graph above

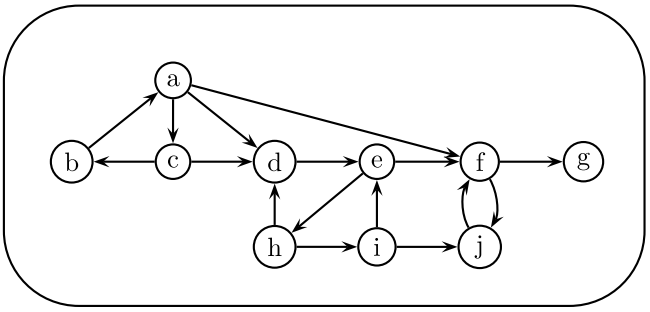
Fig. 1. Illustration of the max-DAG problem.

2. Problem formulation

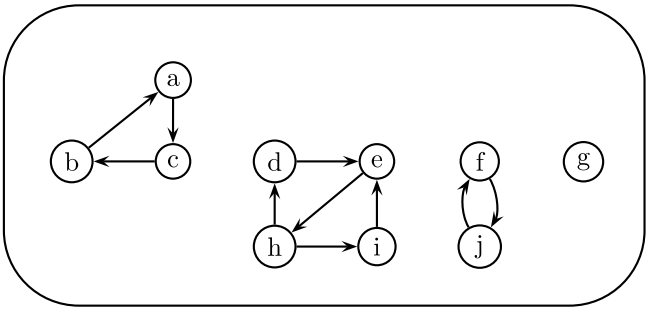
In the context of graph theory the problem described in Section 1 can be modeled as the problem of identifying a set of vertices such that the corresponding induced sub-graph is a DAG. For an introduction, or recapitulation, on the concepts and algorithms of this section see [17–19]. Our problem deals with directed graphs, however in Section 5 we consider a problem which uses undirected graphs. To avoid confusion we use the letter G to represent undirected graphs and G_D to represent directed graphs. A graph is supported by a set of vertices V. In Fig. 1(a) we show an example of a graph G_D with $V = \{a, b, c, d, e, f, g, h, i, j\}$. Furthermore, a graph is characterized by a set of edges E_D. Each edge points from a vertex u to a vertex v. The graph in Fig. 1(a) includes, among others, the edges (a, c), (c, b) and (b, a). This particular sub-set of edges forms a cycle.

Our goal is to select a sub-set of vertices V' such that the induced sub-graph contains no cycles. Hence the above cycle implies that our sub-set V' cannot simultaneously contain the nodes a, b and c. It may, for example, contain a and b, but in that case it cannot contain c. Note that the induced sub-graph G'_D contains all the edges $(u, v) \in E_D$ such that u and v are in V' and (u, v) is in E_D. We are interested in determining the largest such set V'. The graph in Fig. 1(b) illustrates an example of a set of vertices V' that is a maximum for the given graph G_D. In summary, our goal is to find a maximum sub-DAG of a directed graph $G_D = (V, E_D)$, such that the sub-DAG is obtained by removing vertices that are contained in cycles. It may not be immediately clear that the set V' shown in Fig. 1(b) is a maximum, but it is simple to check that it is maximal, i.e., that no vertex can be added to the current configuration without creating a cycle. The configurations that are maximal but not actually a maximum are also known as local maxima.

Note that the set of selected vertices V' is the complement of a FVS [12], i.e., $V \setminus V'$ is a feedback vertex set. The interception between a feedback vertex set $V \setminus V'$ and a cycle of G_D is always non-empty, for any cycle. In particular for the cycle $\{a, b, c\}$ we have that $c \in V \setminus V'$. Another problem that is closely connected to the one we consider is the Maximum Induced Forest (MIF), which instead considers an undirected graph G and searches for



(a) Some input graph



(b) Isolation of SCCs from the top graph

Fig. 2. Illustration of SCC isolation. Original graph G_D on top and graph of isolated SCCs on bottom.

an induced sub-graph that is a tree. This latter problem generally produces smaller solutions. For example if we ignore the directions of the edges in Fig. 1, in both graphs, then the solution $V' = \{a, b, d, g, h, i, j\}$ is also an induced forest. However if we instead assume that the original graph contained the edge (h, d) then the set V' would no longer be an induced forest. This latter set V' is a valid DAG. In our exposition we focus on DAGs instead of forests.

3. Algorithmic approach

Our approach uses mainly concepts related to directed acyclic graphs. We first use an algorithm that identifies SCCs. Recall that for any two vertices u and v in the same SCC there should exist both a path from u to v and a path from v to u. Such an algorithm allows us to, potentially, partition the original graph G_D into a collection of smaller graphs, its SCCs. If the resulting graphs are small enough then even combinatorial algorithms can be applied. This is in essence a simple form of kernelization. In particular we used the algorithm by Tarjan [20]. Other efficient algorithms exist see Sharir [21] and Cormen, Leiserson, Rivest and Stein [22]. These algorithms require only linear time, $O(V + E)$.

This process discards all the edges across different SCCs and keeps only the edges where both vertices belong to the same SCC. An example of this procedure is shown in Fig. 2. The graph in Fig. 2(b) contains only edges inside a component. The edges across SCCs can be safely discarded, as the SCC definition implies that these edges are not involved in cycles.

Another fundamental operation in the algorithms we consider is the identification of cycles. As explained in Section 2 we can identify that a given configuration V' is maximal by testing whether it is possible to add a vertex v to V' so that the resulting induced sub-graph is still a DAG. A Depth First Search (DFS) can be used to determine whether a graph contains a cycle and to identify the nodes involved in the cycle in case it exists. However this procedure requires a complete scan of the graph which is time consuming for an operation that is used extensively in the algorithm. Hence we propose an heuristic approach based on the notion of topological sort of a DAG. As far as we are aware this heuristic is original, and fundamental contribution of our approach.

Such topological orderings can always be obtained from a DAG and consist in arranging the vertices of V' in a line, such that every edge $(u, v) \in E_D$, with $u, v \in V'$, points from left to right. Examples of these orderings are shown in Fig. 3. Figure contains three DAGs. Consider only the vertices labeled 0 to 6 and the edges drawn by dashed arrows. The graphs are drawn according to the topological order, as can be verified by the fact that all the dashed edges point from left to right.

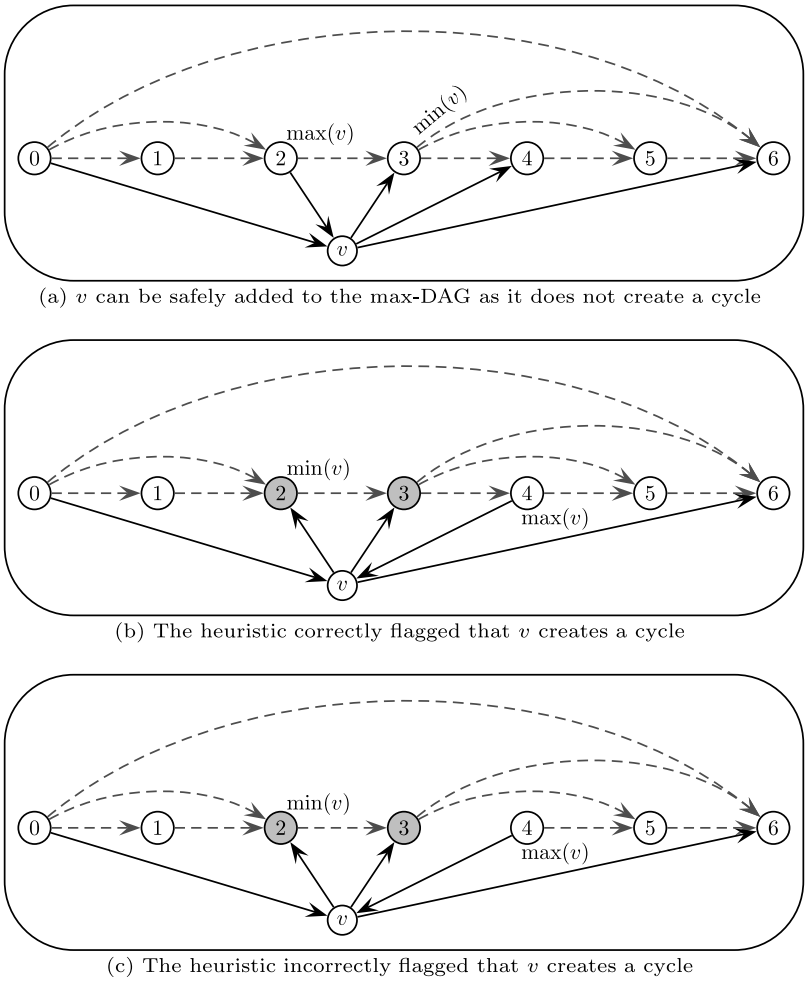
At each step in the algorithm we consider a set of vertices V' as a state. This state must always be a DAG, i.e., the sub-graph induced by V' can never contain a cycle. Our goal is to modify this state until it reaches the global maximum configuration. Hence at each step we choose a vertex v from $V \setminus V'$ and test whether the sub-graph induced by $V' \cup \{v\}$ is still a DAG. If so v can be safely added to V', otherwise v is part of at least one cycle in the induced sub-graph.

If it is safe to add v to V' this process can be executed without further decisions. Otherwise the situation is more complex. If we are running a greedy algorithm then the vertex v is always rejected, which means that it is not added to V'. Conversely the SA approach considers whether it is worth to discard some vertices $I(v)$ of V' so that v can be accepted. This decision depends on how many vertices need to be removed from V'. Hence it is necessary to determine a sub-set $I(v)$ such that the sub-graph induced by $(V' \setminus I(v)) \cup \{v\}$ is a DAG. Ideally $I(V)$ should be as small as possible. Assume for now that we can obtain such a set $I(v)$. The choice the SA procedure must ponder is whether v gets rejected and V' is kept intact or whether v is accepted, in which case V' is replaced by $(V' \setminus I(v)) \cup \{v\}$. The SA decides by considering the value $\Delta E = 1 - |I(v)|$. In each step the algorithm is allowed a limit ℓ that specifies how much energy the current state can afford to change. If $\ell \leq \Delta E$ then the modification is allowed, meaning that the candidate v is accepted, even though this might decrease the size of the current DAG, when $\ell < 0$. Otherwise the candidate v is rejected. The value ℓ is determined by the current temperature value t and a random number, chosen uniformly from the interval $[0, 1]$. Over time the value of the temperature decreases which means that accepting lower values of ΔE becomes progressively unlikely. A precise description is given in Section 4.

In this section we instead focus on how to effectively determine $I(v)$. This is again an instance of the maximum sub-DAG problem. Even though we can execute a depth first search, or Tarjan's algorithm, to identify which vertices are involved in cycles it is not immediately clear which ones should be removed. Hence, we use an heuristic approach, which is also faster than the $O(|V| + |E_D|)$ time bound required by a DFS. Note that an $O(|E_D|)$ time bound for a single decision of the algorithm is too expensive. The heuristic requires $O(V \log V)$ time, more precisely an $O(d(v) \log V)$ worst case time bound, where $d(v)$ represents the degree of the candidate node.[1]

The trade-off is that to increase speed, the decision is not precise, meaning that it overestimates the set $I(v)$. Crucially though the guarantee that the sub-graph induced by $(V' \setminus I(v)) \cup \{v\}$ is a DAG is always valid. The imprecision means that sometimes candidates are rejected because they were pessimistically analyzed and in reality could have been accepted without decreasing the current solution. We use several approaches to mitigate this consequence. The simplest method, when employing this heuristic in

[1] In directed graphs there is both in-degree and out-degree, respectively the number of edges $(u, v) \in E_D$ and $(v, u) \in E_D$ for some u. By degree we mean the sum of both these values.



(a) v can be safely added to the max-DAG as it does not create a cycle

(b) The heuristic correctly flagged that v creates a cycle

(c) The heuristic incorrectly flagged that v creates a cycle

Fig. 3. Illustration of topological orderings and vertex evaluation for three different graphs.

the SA algorithm, is to choose a higher temperature parameter, as this gives more slack to vertices that would otherwise be rejected. A more efficient method is to reorganize the current DAG so that the decision becomes more precise. This is a costly operation and improving the precision for a node v might decrease the precision for other nodes. Hence we use reorganization sparingly.

The procedure is the following. Since the sub-graph induced by V' is a DAG we keep it sorted, by some topological ordering, i.e., using a bijection $s : V' \mapsto \{0, \ldots, |V'| - 1\}$. Whenever a candidate node v is being evaluated we determine the positions of the incoming and outgoing edges. Precisely we determine the maximum position of the incoming edges and the minimum position of the outgoing edges as follows:

- $\max(v) = \max\{s(u) \mid u \in V', (u, v) \in E_D\}$
- $\min(v) = \min\{s(u) \mid u \in V', (v, u) \in E_D\}$

Whenever $\max(v) < \min(v)$ the candidate vertex v is safe, because v is not involved in any cycle in $V' \cup \{v\}$. Hence v can be safely added to V'. On the other hand when $\max(v) \geq \min(v)$ it may be the case that v is part of some cycle. We do not exactly determine whether it actually is part of some cycle. Instead we focus on identifying a set $I(v)$, of incompatible nodes, such that for $V' \setminus I(v)$ the corresponding values $\max(v)$ and $\min(v)$ respect the desired $\max(v) < \min(v)$ inequality.

A simple choice for $I(v)$ are the vertices u of V' for which $\min(v) \leq s(u) \leq \max(v)$ and $(v, u) \in E_D$. In Section 5.1 we consider a more general processes to obtain $I(v)$, for simplicity we will, for now, restrict our attention to this choice. Fig. 3 illustrates this choice. The edges linking vertices in V' are drawn with dashed gray lines. The edges between nodes of V' and the candidate node v are drawn with solid black lines. The vertices are labeled directly with the $s(v)$ values. Fig. 3 illustrates three examples. In Fig. 3(a) we have that $\max(v) = 2 < 3 = \min(v)$. In this example it is safe to add v to V'. In this case the node v will become position $3 = \min(v)$ in the topological ordering and the nodes in positions that were greater than or equal to 3 are shifted up by one value.

The graph in Fig. 3(b) shows an example of a valid cycle identification. This graph is similar to the top graph except that the edges $(v, 2)$ and $(4, v)$ had their directions swapped. In this middle example we have that $\min(v) = 2 \leq 4 = \max(v)$. In this case the annealing procedure needs to make a choice. We have that $I(v) = \{2, 3\}$, because these nodes are less than or equal to $4 = \max(v)$ and the edges $(v, 2)$ and $(v, 3)$ are part of the graph. If the choice is to include v then the nodes 2 and 3 must be removed before v is added. This consequence is highlighted by the gray background of these nodes.

The graph in Fig. 3(c) illustrates a situation where the heuristic incorrectly flags a cycle. Inserting the vertex v in this graph causes no problems, but the heuristic incorrectly claims there is a problem with the resulting configuration. This graph is in all similar to the middle graph, except that the edge $(3, 4)$ was removed. If in this case the SA algorithm decides to accept v this implies that the vertices 2 and 3 get removed, although this was not necessary. Note that node 4 is not restricted by any of edges in the sub-graph and might just as well be located before

Table 1
Notation summary.

Symbol	Definition
G_D	A directed graph.
G	An undirected graph.
V	The set of vertexes of G. When used in arithmetic expressions, such as $\log V$, it means the size of this set.
E	The set of edges of G.
ΔE	The energy variation in one step of the SA algorithm.
$V' \subseteq V$	A subset of vertexes, usually the current SA state. The size of this set corresponds to the energy of the state.
$u, v \in V$	Some vertexes.
s	Current topological order of V'.
$I(v)$	The set of vertexes of V' that form cycles with vertex v.
T	The current temperature of the SA. Starts at T_{hot} and ends at T_{cold}.
α	Cooling rate that is used to reduce T linearly.
ℓ	The limit on ΔE established by the current random number p.

node 0 in the topological order, in which case the candidate node v would have no problems being accepted and its acceptance would entail no removals. However the selection procedure is not able to reach this conclusion, instead it behaves as in the middle case. This is a necessary trade-off for the fast performance of the decision heuristic we propose. Moreover if the nodes 2 and 3 do get removed they can latter be tested for re-insertion, which would then not identify a cycle and allow the nodes to return to the current state.

4. Background

This section provides some background information about the max-DAG problem. Hence it may be partially, or completely, skipped in an initial read. First we recall how the SA and BB algorithms work, Section 4.1. We then survey the state of the art literature on the directed FVS problem, Section 4.2. We finish this section by giving our complexity analysis of the max-DAG problem, Section 4.3.

A reader familiar with SA can easily skip Section 4.1, as the necessary notation is summarized in Table 1, also a quick look at Algorithm 1 might help. We use no concepts from the BB algorithm in our approach, it is only used as a baseline comparison in the experimental evaluation in Section 6. Our presentation is self contained, hence the survey in Section 4.2 is given for exploratory reasons, and to frame our work in context. It is not required reading to obtain an understanding of our approach. Finally the analysis we present in Section 4.2 gives some further understanding into the nature of the max-DAG problem. In particular we will establish that the problem is NP-complete and Poly-APX-complete. This validates our the need for an approximating probabilistic technique such as SA since it is unlikely that there exists a polynomial time algorithms that can solve or approximate this problem, with a constant or poly logarithmic approximation ratio. However it is a theoretical analysis and can be safely skipped by the pragmatic reader.

4.1. Search recapitulation

The SA algorithm is shown in Algorithm 1. The current state of the algorithm is represented by a sub-set of vertices V' which in the generic algorithm starts off as the empty set $\emptyset$. The initial temperature is set to T_{hot} and the final temperature is set to T_{cold}. The number of iterations to execute is given as argument n. The temperature is stored in a variable T and is decreased linearly. It is possible to update the temperature less frequently, i.e., once in each b iterations, for some batch value b. For simplicity we omit this process. Also choosing appropriate values for the temperature is a tricky business, as seen in Section 6. In our setup we found it simpler to specify temperatures from a limit ℓ and a probability value p and inverting the formula in line 6, i.e., $T = \ell / \log_2((1/p) - 1)$.

At each iteration the algorithm selects a random number uniformly at random from the interval [0, 1]. This value is stored in the variable p, see line 5. This value is used to define the minimum delta value, stored in variable ℓ. A modification to V' is only accepted if it changes the number of vertices by at least ℓ. The SA algorithm uses the notion of energy. For our application the energy is the current number of elements in V'. Let us consider a few values of p to gain some insight into the algorithm. When p is $1/2$ the resulting value of ℓ is 0. This means that modifications to V' for which inserting v implies removing only one other vertex are accepted half the time. Note that this occurs independently of the temperature value T and is a deliberate decision, since we could have chosen to compute ℓ as $-T \times \log_2(p)$ if this was not intended. This is in contrast with the greedy algorithm which never accepts modifications that do not improve the current solution V'.

Our implementation guarantees that p is never 0 or 1 as this would cause issues in the computation of ℓ, moreover its has no impact on the algorithm as these two numbers are a zero measure sub-set of [0, 1]. Still it may occur that the value of ℓ obtained is larger than 1, which is not possible because at most we add one new vertex to V' the vertex v. Hence the number of vertices can never increases by more than 1. Since this is an unreasonable value of ℓ we truncate its value back to 1 if the original computation is larger. Note also that the computation of the value d, in line 6, might yield negative values, in which case transitions that reduce the size of the current solution V' are allowed.

Given a value of ℓ we then execute the procedures described in Section 3. First we choose a vertex v uniformly at random from $V \setminus V'$, in line 10. Then we search for a set of incompatible vertices $I(v)$, such that $\ell \leq \Delta E = 1 - |I(v)|$. This is done as explained in Section 5.1 to avoid using more than $O((2 - \ell) \log V)$ time when no such set exists. In that case the `Verify` procedure fails and no modification is performed. Otherwise the corresponding set $I(v)$ is identified and the current solution V' is updated.

Let us also recall the BB algorithm. Essentially the BB algorithm consists in checking all the possible sets $V' \subseteq V$ to determine if the corresponding induced sub-graph is a DAG, and among the sets with this property the algorithm determines the largest one. Note that this is a big search space, if there are n nodes in V, i.e., $|V| = n$, then there are 2^n possible sub-sets $V' \subseteq V$. This search space is organized in a tree shape to make the search effective and allow for pruning.

The tree organization means that a set V' is only tested if $V' = V'' \cup \{v\}$, for some node v and the set V'' was previously tested and was verified to induce a DAG. This gives an initial pruning because the converse of this property is that if the set V'' does not induce a DAG then the set V' does not need to be tested. The caveat with this heuristic is that the set V' can be decomposed into $V'' \cup \{v\}$ with several different choices of V'' and v. Just because V' is not tested according to one configuration it does mean that it is guaranteed not to be tested. The exact condition is the following: if the set V' is such that for any set V'' such that $V' = V'' \cup \{v\}$, for some vertex v, and the sub-graph induced by V'' is not a DAG then the set V' is never tested by the BB algorithm, because it is guaranteed not to induce a DAG. What we mean by never tested is that the set is not even generated and therefore adds no time at all to the algorithm's execution time.

Note however that considering all the decompositions $V'' \cup \{v\}$ is harder to organize if we focus on the set V'', instead it is easier to focus on the vertex v. To organize the search we order the set

```
input : ⟨G, T_hot, T_cold, n⟩
output: ⟨V'⟩
1  V' ← ∅ ;
2  T ← T_0 ;
3  α ← (T_hot − T_cold)/n ;
4  for i = 1 to n do
5  |  p ← Random(0, 1) ;
6  |  ℓ ← T × log_2((1/p) − 1) ;
7  |  if ℓ > 1 then
8  |  |  ℓ ← 1 ;
9  |  end
10 |  v ← SelectFrom(V \ V') ;      /* Choose a vertex
   |   uniformly at random. */
11 |  if Verify(V', v, ℓ) then
12 |  |  ; /* Executes only when ℓ ≤ ΔE = 1 − |I(v)| */
13 |  |  I(v) ← Identify(V', v) ;  /* Retrieves the set
   |  |   I(v) */
14 |  |  V' ← (V' \ I(v)) ∪ {v} ;      /* Update State */
15 |  end
16 |  T ← T − α ;
17 |  i ← i + 1 ;
18 end
19 return V' ;
```

Algorithm 1: Pseudo code for the simulated annealing algorithm.

V, i.e., have $V = \{v_1, v_2, \ldots, v_n\}$. This order can be arbitrary but is kept fixed during the execution of the algorithm. With this order we can define the order of a sub-set V', which is the value of the largest index i such that $v_i \in V'$. More precisely we say that the set V' has order ℓ if for any $v_i \in V'$ we have that $i \leq \ell$. This order ℓ is related to the depth d of the set V' in the search tree, more precisely if a set V' has order ℓ then all occurrences of V' in the search tree have depth $d \geq \ell$. The depth is used in the search tree to select which vertex to consider next. For example, if V' occurs at some depth d and the induced sub-graph is a DAG then the search further considers the sets V' at depth $d+1$ and the set $V' \cup \{v_{d+1}\}$ at depth $d+1$. This is illustrated with the recursive call in lines 6 and 9 of Algorithm 2.

```
1  BranchAndBound(V', d, b)
2     if |V'| > b then
3  |     b = |V'|
4     end
5     if b < |V'| + n − d − 1 then
6  |     b =BranchAndBound(V', d + 1, b) ;
7     end
8     if b < |V'| + n − d and V' ∪ {v_{d+1}} induces a DAG then
9  |     b =BranchAndBound(V' ∪ {v_{d+1}}, d + 1, b) ;
10    end
11    return b
```

Algorithm 2: Pseudo code for the Branch and bound algorithm.

Another important heuristic is the bound b. The idea is that during the search we keep track of the size of the best solution found so far, in variable b. This allows us to discard parts of the search space that cannot contain solutions that are better than the current best value. If $b < |V'| + n - d$ then the descendants of the current set V' can still obtain a solution that is better than the current best value b, in particular if all the vertices v_i with $i > d$ can be inserted into V' and still induce a DAG. On the other hand if this condition is false than this portion of the search space can be trimmed because not even this extreme case will yield a solution better than the current best bound b. This is the condition we check in line 9. The condition in line 6 uses a similar logic but is a bit tighter because we are considering a solution without the vertex v_{d+1}. As final important observation note that testing the condition that $V' \cup \{v_{d+1}\}$ in line 8 is done by executing a DFS on the induced sub-graph and therefore requires at most $O(|V| + |E|)$ time.

4.2. Related work

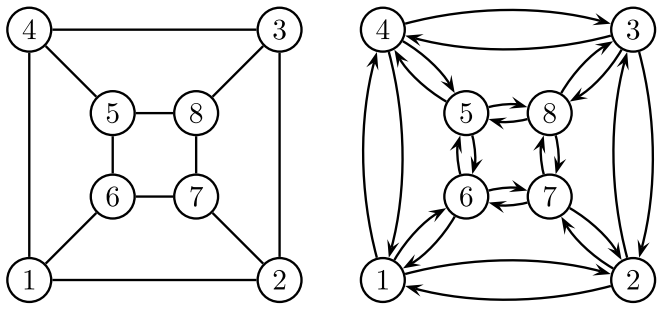
As mentioned in the Introduction in this paper we considered the directed FVS problem, even though our approach explored the complementary max-DAG problem. Hence we will review the history on this problem. Both the directed and undirected versions are known to be NP-Complete [13].

When the graph G is undirected, there are several results about the feedback vertex set problem, such as an exact algorithms for finding a minimum FVS in a graph on n vertices in $O(1.9053^n)$ time [23] and in $O(1.7548^n)$ time [24]. There is also a polynomial time 2-approximation algorithm for the minimum feedback vertex set problem [25]. The first parameterized algorithms, by Bodlaender [26] and Downey and Fellows [27], obtained a $O(17k^4!n^{O(1)})$ running time for the parameterized feedback vertex set problem in undirected graphs.

Bar-Yehuda and Geiger [6] presented several approximation strategies. However they rely either on problem constraints (e.g., see Chitnis et al. [28],Even [29] for parameterized FVS) or only apply to specific families of graphs (e.g., see Lokshtanov et al. [30], Papadopoulos and Tzimas [31],Wang et al. [32] for approximation algorithms for specific graph families).

In the case of directed graphs known results can be divided into exhaustive search approaches and heuristic approaches. The first result that improved the $O(2^n)$ barrier was by Razgon [33], which obtained an $O^*(1.9977^n)$ time bound. Like us their approach focused mainly on the complementary maximum induced DAG problem. This algorithm was improved by Chen et al. [34], that proposed a fixed-parameter algorithm for this problem with an $O(4^k k!n^{O(1)})$ time bound.

As for the heuristic approaches Galinier, Lemamou and Bouzidi [15] were the first to propose the application of simulated annealing to this problem. The version we propose uses the same principles as their solution, which some important variations. In the local optimization strategy by Galinier et al. [15] only two positions are ever considered for the new vertex v before $\min(v)$ or after $\max(v)$. Instead in our solution we show how to scan in between configurations, at no asymptotic extra time, see Section 5.2. Another important contribution of our approach is the insight into the temperature selection. In our experimental results we observed that for this kind of problem cold temperatures obtained much better performance than reasonably hotter ones. Hence whereas Galinier et al. [15] used a geometric cooling schedule we chose a linear one. This allowed us to transition from hot to cold at a slower pace, while avoiding both extreme conditions. In a too hot scenario the algorithm tends to diverge from the optima. A strictly cold search behaves essentially as the greedy algorithm, which obtained a reasonable performance. However the tuned SA algorithm with linear cooling obtained even better performance. Another significant difference is that our approach is stochastic, meaning that we do not evaluate all potential candidate vertices v to determine which one is the best for the next step. Instead a vertex not yet in V' is chosen uniformly at random. The rational for this decision is that a bad vertex that is part of the current configuration will quickly be pressured to leave in future moves. Moreover this allows us to



(a) undirected graph (b) directed graph

Fig. 4. Construction of a directed graph (Fig. 4(b)) from an undirected one (Fig. 4(a)).

focus on the performance of this procedure, for which we use efficient data structures to obtain around $O(d(v)\log V)$ time, or even better if the vertex gets rejected.

A recent improvement to the SA approach was proposed by Tang, Feng and Zhong [16]. They propose a way to order the candidate vertices so that the vertex v that gets selected is closer to the global optimum. This solution strictly improved the SA of Galinier et al. [15] and was tested in our comparison, see Table 6. Again our stochastic approach with dedicated data structures allowed us to quickly include or discard vertices, therefore we obtained better results.

4.3. Complexity analysis

First we show that the problem at hand is NP-complete. Our proof is essentially a recap of the proof that the feedback vertex set problem is NP-Complete, given by Karp [12]. However we need to recall this reduction for the Poly-APX-complete result of Corollary 1. In this case we consider the sub-DAG problem, which given a graph $G_D = (V, E_D)$ and an integer k consists in determining if there is a set of vertices V' of size k such that the sub-graph induced by V' is a DAG.

Theorem 1. *The Sub-DAG problem is NP-complete.*

Proof. First we show that the Sub-DAG $\in$ NP. For a given graph $G = (V, E_D)$, we consider a sub-set $V' \subseteq V$ of vertices as a certificate. First we check if V' contains exactly k elements, where k is the desired number of vertices specified by the Sub-DAG instance at hand. This requires only $O(V')$ time if V' is given as a linked list or $O(V)$ if it is given as an array of booleans. Verifying that the induced sub-graph is a DAG amounts to checking that it does not contain cycles. This can be solved by executing a DFS on the sub-graph and checking that it does not contain back edges. The resulting algorithm requires $O(V' + E_D)$ time, when the structure that identifies the sub-set V' takes $O(1)$ time to determine if a vertex v is in V' or not. An array of booleans indexed over V suffices. Hence in total a verification algorithm requires $O(V+E)$ time and space, which is polynomial in the size of the input.

Next we show that Max-DAG is NP-Hard. For this goal we use a reduction from the Independent-Set problem. Given an undirected graph $G = (V, E)$ the Independent-Set problem consists in determining whether there exists a set of vertices $V' \subseteq V$, of size k, such that the induced sub-graph contains no edges, i.e., for any $\{u, v\} \in E$ it cannot be that both the vertices exist in V', i.e., either $u \notin V'$ or $v \notin V'$. Given an instance graph $G = (V, E)$, of the Independent-Set problem, we need to build an instance $G_D = (V_D, E_D)$ of the Sub-DAG problem. We illustrate this reduction in Fig. 4. The given undirected graph G is shown in Fig. 4(a). The set of vertices $\{1, 3, 5, 7\}$ is an example of an independent set in this graph.

For this process we need to construct G_D to be a representation of G as a directed graph. In essence for every edge $\{u, v\} \in E$ we will add two edges (u, v) and (v, u) to E_D. The set of vertices is kept unaltered, i.e., $V_D = V$. We illustrate this construction in Fig. 4. The graph in Fig. 4(b) is directed, which is indicated by the arrows at the end of the edges. This construction yields $|V_D| = |V|$ and $|E_D| = 2|E|$ and therefore can be obtained in linear time and space.

To finish the reduction we require that the sub-DAG to determine in G_D should also have size k. Hence to finish this reduction we claim that it is possible to find an independent set of size k for G if and only if it is possible to find a sub-DAG with k vertices in G_D.

We start by showing that if there is an independent set of size k in G then there is a sub-DAG with k vertices in G_D. The reduction is trivial. Given that V' is an independent set of G then V' is also a DAG in G_D, because the sub-graph induced by V' in G_D contains no edges. Assume by contradiction that $(u, v) \in E_D$ is an edge in the sub-graph induced by V', then $u \in V'$ and $v \in V'$. Hence by construction $\{u, v\}$ must also be an edge in E, which is a contradiction because V' should be an independent set but it contains u and v and the edge $\{u, v\}$ exists in E.

Finally we need to show that if we obtain some set of vertices $V' \subseteq V_D$ such that the induced sub-graph of G_D is a DAG then the set V' is an independent set in G. Let $\{u, v\} \in E$ be an edge of G then (u, v) and (v, u) are edges of E_D, which form a cycle. Therefore either $u \notin V'$ or $v \notin V'$, otherwise the DAG contained a cycle. Hence V' is an independent set. □

This complexity condition seems to imply that a pragmatic approach to the problem either consists in studying exponential algorithms or polynomial time approximation algorithms. As we will show in Corollary 1 however even this second hypothesis is infeasible, unless P=NP. This Corollary is the consequence of on two main considerations. On the one hand the Max Independent Set is Poly-APX-complete. On the other hand the reduction employed in Theorem 1 is extremely robust and therefore qualifies as an approximation preserving reduction. Specifically it qualifies as an S-reduction, namely the strictest of several alternatives considered by Crescenzi [35], which preserve several membership conditions. The approximation ratio is equal in the Max Independent Set and in max-DAG, because it is exactly the same sub-set V' from the same set of vertices V, only the set of edges is modified.

Corollary 1. *The max-DAG problem is Poly-APX-complete under PTAS-reductions.*

Proof. Clearly max-DAG is in Poly-APX. An algorithm which simply chooses one vertex v of V runs in polynomial time and is a $|V|$ approximation to max-DAG, because the maximum DAG must be a sub-set V' of V.

To show that max-DAG is Poly-APX-complete we note that the Max Independent Set is Poly-APX-complete under PTAS-reductions, see Bazgan, Escoffier and Paschos [36]. Hence we use the reduction in Theorem 1 to reduce Max Independent Set to max-DAG. Note that S-reductions are also PTAS-reductions. Because the composition of a PTAS-reduction and an S-reduction still is a PTAS-reduction, we only need to confirm that the reduction in Theorem 1 is an S-reduction. Using the notation of Crescenzi [35] we need to describe a function g that transforms solutions of max-DAG into solutions of Max Independent Set, in polynomial time, and preserves the solution measure. Formally

$m_A(x, g(x, y)) = m_B(f(x), y)$. This is trivial because a set of vertices V' that induces a DAG as a sub-graph of the transformed instance $G_D = f(x)$ is also an independent set in the original graph $G = x$, as shown in the proof of Theorem 1. Hence $g(x, y) = V'$, for $y = V'$. Moreover the measure values in these problems are the number of vertices in the set V'. Therefore the necessary equality simplifies to $m_A(x, g(x, y)) = |V'| = m_B(f(x), y)$. □

Even though it is unlikely that Max Independent Set can be approximated to a ratio better than polynomial in polynomial time, polynomial time approximation algorithms do exist for bounded degree graphs. In particular a simple greedy algorithm obtains a $(\delta + 2)/3$ approximation ratio, for a graph with maximum degree δ. Moreover the same algorithm is also guaranteed an approximation ratio of $(d + 2)/2$, where d is the average degree of the graph. See Halldórsson and Radhakrishnan [37]. Hence this algorithm strongly influences our approach to the max-DAG problem.

The algorithm works as follows: we start with an initially empty independent set $V' = \emptyset$ and an instance graph G. At each step we choose the vertex v with minimum degree, i.e., $d(v) = \min\{d(v') \mid v' \in V\}$, where $d(v)$ represents the degree of v. By degree we mean the size of its neighbor set $N(v) = \{u \mid \{v, u\} \in E\}$. The vertex v is added to V' and removed, along with its neighbors, from the set V. Precisely set $V' \leftarrow V' \cup \{v\}$ and $V \leftarrow V \setminus (\{v\} \cup N(v))$. Likewise the set E is cleaned from edges that connect vertices that no longer exist in V. The algorithm terminates when the set V becomes empty. Note that at each step the set V' is an independent set.

Consider a simulated annealing approach to the Independent set problem. A simple algorithm, similar to the greedy one, is the following. Start with an empty set V', but keep the graph G unmodified. At each step a vertex v is allowed into V' if there is no edge $\{u, v\} \in E$ such that $u \in V'$. Such a vertex u is incompatible with v. On the other hand if such incompatible vertices u do exist the annealing algorithm counts how many of them exist. If this number is not too big and the temperature and the random choice allow for it the vertex v may be added into V' anyway. In which case all the incompatible vertices u first need to be removed from V'. Note that like in the greedy algorithm the degree of a node v plays an important role on whether it is allowed in the solution or note. A node v for which $d(v)$ is big is unlikely to be allowed into V' because it is likely to have incompatible nodes in V'. Even if a node with a high degree is allowed into V' at some step it will be pressured out because all its neighbors are incompatible with it. This algorithm is very similar to the one we propose for the max-DAG problem, except that in our case a property violation occurs when the vertex to add is part of a cycle, not just part of an edge.

5. Methodology

In this section we will explain how to efficiently implement the decision heuristic described in Section 3 and also a global re-organizing procedure that mitigates the limitations of the decision heuristic.

5.1. DAG data structures

Let us now discuss which data structures can be used to implement the decision heuristic described in Section 3. As explained in that section we aim to maintain a representation of the set V' of vertices, such that the induced sub-graph is a DAG. As the annealing algorithm proceeds vertices get added and removed from V'. Moreover for the local search heuristic procedure it is necessary to keep this set ordered, by some topological order of the DAG. Given these constraints we choose to store the vertices of V' in a Binary Search Tree (BST). In particular we used splay trees in our implementation. Any binary search tree provides the operations necessary for our algorithm, but a BST with worst case $O(\log V)$ guarantees would be ideal, such as AVL's or Red–Black trees. Splay trees might suffer from the occasional worst case bad performance, but they provide amortized time guarantees of several desirable time bounds. Given that our algorithm performs a sequence of operations the amortized time bounds are sufficient. Single operation worst case guarantees are not crucial. Storing the sequence of nodes in a BST has the added advantage that whenever a node is added or removed the new positions of the nodes are obtained automatically. To map from a vertex of G to the corresponding node in the splay tree we simply store the corresponding node in the structure that represents a vertex. Likewise each node may also store a pointer to its corresponding vertex.

Consider again the example at the top of Fig. 3. In this case we have that $\max(v) = 2 < 3 = \min(v)$ and therefore it is safe to add the node v to the set V''. This consists in inserting the vertex v into the splay tree so that it becomes the new vertex at position 3. Hence the previous vertex at position 3 becomes the vertex at position 4, the previous vertex at position 4 becomes the vertex at position 5 and so on. This is obtained with the insertion operation in $O(\log n)$ amortized time in splay trees. To determine the current position of a vertex it is enough to store the current sub-tree size in each node of the splay tree.

Whenever a node needs to be removed from the set V' it is removed from the splay tree. Consider for example the cases in the middle and bottom of Fig. 3. In these cases the nodes at position 2 and 3 need to be removed from the splay tree. After this operation, and before the node v is inserted, the node that was at position 4 becomes the node that is at position 2, the node that was at position 5 becomes the node that is at position 3 and so on.

We can now focus on the exact procedure we use for the local search heuristic. According to the description in Section 3 we need to determine $\min(v)$ and $\max(v)$. To determine $\min(v)$ we need to visit the neighborhood of v, i.e., all the vertices $u \in V'$ such that there exists an edge $(v, u) \in E_D$. Actually we only need the neighborhood in the sub-graph induced by V'. However, because our graph G_D is represented in an adjacency list, we actually end up traversing the complete neighborhood of v to obtain the necessary nodes u. For each node u that is a neighbor of v, i.e., for each edge (v, u) we first check if $u \in V'$, by checking if u is currently in the splay tree. If so, we then determine its position in the topological order. This is actually the size of the left sub-tree of node u, since looking up node u involves a splay operation that pulls it to the root of the tree. In fact looking up u is slightly tricky, because the splay tree is ordered by the topological order. Therefore this process is not computed as a search over the tree, instead we keep an array that for each node $u \in V$ keeps the information of whether the node is in the tree or not, and if so where is it in the tree. To be absolutely precise each node of the splay tree is represented by a `struct node`, that contains pointers to the left and right sub-trees and to the parent tree. These structs are allocated all at once, at the beginning of the algorithm. They are kept in an array that contains exactly $|V|$ elements, each one corresponding to a node $u \in V$. We also keep a pointer to the `root` of the splay tree. If a given `struct` is pointed to by the `root` pointer or has a non `NULL` parent pointer then it is in V', because it is part of the splay tree. Otherwise it is not in V', because it is not part of the splay tree. In either case the `struct` is allocated. Whenever the `struct`, corresponding to node u, is in the splay tree, we splay it to the root of the tree and use the size of the resulting left sub-tree to determine its position in the topological order. Note that this representation wastes a bit of

space, since full `struct node` space is used even for nodes that are not in the splay tree. On the other hand we save one level of indirection that would exist in an alternative that used an array of single pointers to these `struct node` elements, which would be `NULL` when the node was not in the splay tree. Saving a level of indirection usually yields better cache performance and the space penalty in this case is not too steep, particularly when V' is large.

Hence we can determine $\min(v)$ by computing the minimum of all the position values obtained in the procedure we just described. Likewise we need to compute a similar process to obtain the value $\max(v)$. This time we need to determine the nodes $u \in V'$ such that $(u, v) \in E_D$. The process is essentially similar to the one we described for $\min(v)$, except that this time v is a neighbor of u and not the other way around. This means that we also must keep an adjacency list representation of the transposed graph G^T in memory. This representation is created when the algorithm starts, along with the adjacency list representation of the original graph G_D. Hence, during a verification procedure this information is readily available. Algorithm 3 gives a succinct description of this computation.

```
1  GetMax(G^T, V', v)
2      M = −∞
3      for u ∈ N^T(v) do
4          if u ∈ V' and s(u) > M then
5              M = s(u)
6          end
7      end
8      return M
9
10 GetMin(G, V', v)
11     m = +∞
12     for u ∈ N(v) do
13         if u ∈ V' and s(u) < m then
14             m = s(u)
15         end
16     end
17     return m
```

Algorithm 3: Pseudo code to compute the $\max(v)$ and $\min(v)$ values. Recall that s represents the current topological ordering of V'.

Simulated-annealing optimizations. Note that when $\max(v) < \min(v)$ there is no alternative to the previous procedure. Therefore there is little opportunity to obtain a better bound than $O(V \log V)$ time or more precisely $O(d(v) \log V)$, where $d(v)$ is the degree of v, obtained by summing both the in-degree and the out-degree. However for most verification procedures the resulting situation is $\min(v) \leq \max(v)$. In these cases we need to determine the set $I(v)$ described in Section 3. This is straightforward to obtain by first computing $\max(v)$ and then computing $\min(v)$. The computation of $\min(v)$, iterates over all the nodes $u \in V'$ such that an edge (v, u) exists. For each such node we determine its position $s(u)$. To determine $\min(v)$ we compute the minimum among the $s(u)$ values. Since $\max(v)$ is already known it is possible to determine which of the nodes u are actually in $I(v)$. Whenever a node $u \in V'$ is considered in the computation of $\min(v)$, because an edge (v, u) exists, we also check if $s(u) \leq \max(v)$. If this condition is verified we have that $u \in I(v)$. Hence we can determine $I(v)$ in the same $O(d(v) \log V)$ time bound.

The case when $\min(v) \leq \max(v)$ can be very frequent (e.g., when we have already identified a maximal solution or are very close to it), hence it is worthwhile to try to improve it. Notice that computing a large set $I(v)$ only to have it be rejected by the simulated annealing process amounts to redundant work. The annealing decision depends only on the size of $I(v)$. At each step the simulated annealing algorithm determines a value ℓ, which depends on the current temperature and a random value p, chosen uniformly from $[0, 1]$. Whenever $\ell \leq \Delta E = 1 - |I(v)|$ the transition is accepted, otherwise it is rejected. Hence accepted transitions usually occur for negative values of ℓ, in fact, the largest value of ℓ that is significant for this problem is 1. Any other positive values of ℓ results in rejected transitions. Therefore our generation procedure never generates positive values larger than 1. Generating the value ℓ is fast, i.e., $O(1)$ time. Therefore we generate the ℓ value before computing the set $I(v)$. This has the advantage that we do not need to compute the complete set $I(v)$. As soon as we identify enough elements in $I(v)$ so that the inequality with ℓ is no longer valid, we can abort the identification procedure and the corresponding transition. Hence our goal is to obtain an $O((2 - \ell) \log V)$ time bound for each verification process that fails. Our method is not guaranteed to obtain this bound all the time. We improve the best case time to this bound and show that our solution is effective in practice.

Our technique involves several nuances, which we will now explain and simultaneously obtain more general selections of incompatible sets $I(v)$. First we store the necessary nodes in BSTs, sorted by their position in the topological order. This makes it easy and efficient to obtain the $\min(v)$ and $\max(v)$ values, or their current best approximations. Precisely we consider the set of nodes $\text{out}(v) = \{u \in V' | (v, u) \in E_D\}$ and $\text{in}(v) = \{u \in V' | (u, v) \in E_D\}$, which can be used to obtain the $\min(v)$ and $\max(v)$ values respectively. Consider the middle example in Fig. 3. In this case we have $\text{out}(v) = \{2, 3, 6\}$ and $\text{in}(v) = \{0, 4\}$, yielding $\min(v) = 2$ and $\max(v) = 4$ respectively. As before if we have $\max(v) = 4$ computed first then identifying $I(v)$ can be obtained by searching for 4 in the BST that contains the representation of $\text{out}(v)$. This allows us to divide the set $\text{out}(v)$ in two, the set $I(v) = \{2, 3\}$ of elements that are smaller than or equal to 4 and the set $\{6\}$ which contains safe nodes larger than 4. This computation is supported in BSTs, either by simply counting the number of elements less than or equal to 4, or by actually splitting the tree in two. We also use splay trees to store the $\text{in}(v)$ and $\text{out}(v)$ sets, which provides a simple implementation of both these operations.

An additional optimization that we use is the following. Assume that we know that $\max(v) = 4$, and that we are in a situation where the simulated annealing has decided that $\ell = 0$. In this case the transition will be rejected because $0 = \ell > -1 = 1 - 2 = 1 - |I(v)| = \Delta E$. Hence the order in which we process $\text{out}(v)$ matters. If we obtain $\text{out}(v)$ as $6, 3, 2$ then only after we receive the number 2 can we reject the transition. On the other hand if we receive $\text{out}(v)$ as $2, 3, 6$ we can reject the transition immediately after receiving the number 3 and therefore there is no need to process the number 6. In this case it is only one number but in general this approach reduces the number of vertices processed from $d(v)$ to $2 - \ell$. For this early termination to be effective we need two conditions. We need to search for $\max(v)$ in the intermediate configuration of $\text{out}(v)$, i.e., this search is executed every time a new node u of $\text{out}(v)$ is identified. We need the nodes in $\text{out}(v)$ to be obtained according to the current topological order of the V', s. This second condition amounts to keeping the adjacency list of a node v sorted according to the topological order, s. In general we cannot guarantee this requirement, since this order potentially changes with every step of the annealing algorithm and updating all the corresponding adjacency lists would amount to more work than what this optimization saves. Hence our approach is simple and pragmatic. Each time a verification operation is executed in node v a prefix of its adjacency list is traversed, namely to obtain $\text{out}(v)$. This prefix could be the whole list. At the end of the verification procedure this prefix of the adjacency list is sorted,

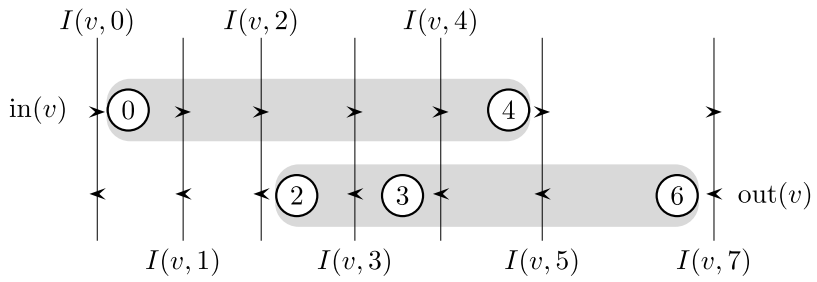


Fig. 5. Schematic representation of incompatibility sets.

according to the current topological order, and is used to overwrite the current configuration of this prefix in the adjacency list. Hence at the end of each verification procedure the corresponding prefix of the list is guaranteed to be in the correct order. This optimization guarantees that in a subsequent verification of node v the corresponding adjacency list is obtained according to the topological order. This reduces the amount of nodes that are processed in the second verification from $O(d(v))$ to $O(2-\ell)$, provided the topological order has not changed much meanwhile and that the second ℓ value does not imply scanning a larger prefix. Even thought this optimization is not always effective it a can be implemented with no extra time overhead and it does improve the performance in practice, see Section 6.

Note that the adjacency list of v will contain nodes u that are not in V', i.e., there are nodes $u \in V \setminus V'$ for which edges $(v,u) \in E_D$ exist. These nodes are placed at the end of the prefix reordering we described. Also note that an inorder traversal of the BST containing the partial configuration of out(v) provides the nodes in the topological ordering which is necessary for the prefix overwrite. Hence no extra sorting is actually necessary.

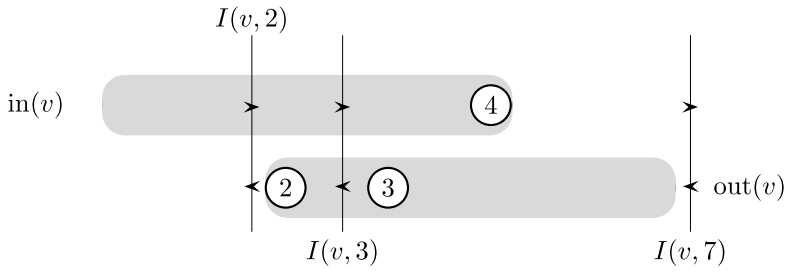
Thus far in our explanation we assumed that the whole set in(v) is processed before the set out(v) is considered. This is not actually the case, as such a procedure would not actually obtain $O((2-\ell)\log V)$ time, because of its dependency on the in-degree of v. Note that to obtain this bound we also need to reorganize a prefix of the adjacency list of the transposed graph G^T. This time to keep it in reverse topological order. This is, likewise, done by using the reverse inorder sequence of the BST storing the in(v) set.

5.2. Determining the set $I(v)$

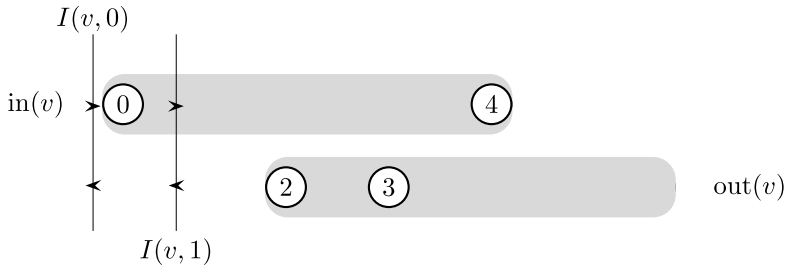
We will now describe the general procedure we use to determine the set $I(v)$. Consider again the situation in the middle graph of Fig. 3. In case the value ℓ is 0 the set $I(v) = \{2,3\}$ is rejected by the verification procedure, because it contains two elements and the maximum size allowed is one. However this rejection is excessive. We could have chosen $I(v) = \{4\}$, which contains only one element and therefore would be accepted by the verification procedure. Recall that in this example we have out(v) = $\{2,3,6\}$ and in(v) = $\{0,4\}$. This means that we need to consider a wider range of incompatibility sets. Given an integer k we consider the set $I(v,k)$, which consists of the nodes $u \in$ out(v) such that $s(u) < k$ and the nodes $u \in$ in(v) such that $k \leq s(u)$. Recall that in our exposition the nodes are being identified by their $s(u)$ values. Hence we have that in our running example $I(v,7) = \{2,3,6\}$, where these nodes are obtained from out(v). The incompatibility set we have considered so far is $I(v, \max(v)+1) = I(v,5) = \{2,3\}$. Consider instead the set $I(v,2) = \{4\}$. This set contains only elements from in(v) and is smaller. Clearly we could also have considered the sets $I(v,4) = \{2,3,4\}$ and $I(v,3) = \{2,4\}$, which actually contain vertices from both sets out(v) and in(v). However these incompatibility sets have more than one element. A schematic representation of these sets is shown in Fig. 5.

We need a general procedure to determine if there is some set $I(v,k)$ such that $\ell \leq \Delta E = 1 - |I(v,k)|$. Our procedure works by interleaving the iteration of the adjacency lists of v in G_D and in G^T, i.e., we alternate between considering the nodes u because there is an outward edge of v, $(v,u) \in E_D$, and because there is an inward edge into v, $(u,v) \in E_D$. Whenever appropriate these vertices are added to the corresponding BST, the one for out(v) and the one for in(v) respectively. In general the search proceeds while there is at least one value of k for which $\ell \leq \Delta E = 1 - |I(v,k)|$. Note that we can compute the value $|I(v,k)|$ in $O(\log V)$ amortized time using splay trees. Hence the verification procedure keeps track of an interval $[a,b]$, such that any value of k for which the desired condition holds must belong to this interval. This interval is initialized with $a = 0$ and $b = |V|$. This matches the fact that initially the BSTs are empty and therefore all current $I(v,k)$ are also empty. Whenever the process considers a node $u \in V'$ that is a neighbor of v, i.e., $(v,u) \in E_D$, then the value of b may need to be decreased. For example suppose that $\ell = 0$ in the middle graph of Fig. 3 that we are currently considering. Assume that we first consider the nodes 3 and 4, which are added to the BSTs corresponding to out(v) and in(v) respectively. At this point the values of a and b are still the initial values. Therefore the corresponding interval is still $[0,7]$. Now assume that the algorithm needs to process the node $u = 2$. At this point the set $I(v,7)$ contains, at least, the vertices 2 and 3. Therefore the value of b needs to decrease. To decrease this value we consider all the currently known positions of out(v). In this case we have $\{2,3\}$. Therefore b is reduced to 3. The number of elements known to be inside $I(v,4)$ is computed. This value is two, therefore we need to further reduce b to 2. The number of elements known to be inside $I(v,2)$ is one. Hence $b = 2$. Now assume that we need to process the node $u = 0$. After this new node is added to in(v) the number of elements known to be in $I(v,a) = I(v,0)$ is at least two. Therefore we need to increase the value of a. Increasing an index is slightly different from decreasing, because this time we consider the set of known positions in in(v) increased by 1, i.e., in this example $\{1,5\}$. Hence we increase a to 1. Again we verify that the set $I(v,1)$ is known to have at most one element and therefore a is kept at value 1. Hence at this point our current search interval $[a,b]$ is $[1,2]$. The search can still proceed because this interval is not empty. Hence assume that the search now processes node $u = 6$. This node is added to the BST corresponding to out(v), but has no impact in the value of $I(v,2)$. Hence the value b is kept unaltered. An illustration of the procedure we just described is shown in Fig. 6. Fig. 6(a) shows the reduction of position b and Fig. 6(b) the increase of position a.

At this point there are no more nodes to consider. Therefore the search terminates. The search ended with $1 = a \leq b = 2$. Hence the verification process accepts the transition and node v will be added to V'. However an incompatibility set must first be removed from V'. We can only use $I(v,1)$ or $I(v,2)$, in general we can use any integer in $[a,b]$. To maximize the size of the sub-DAG it is best to minimize the number of nodes that need to be



(a) Reducing the value of b to 2



(b) Increasing the value of a to 1

Fig. 6. Schematic representation of range reduction $[a, b]$. Fig. 6(a) shows how to reduce value b and Fig. 6(b) how to increase value a.

removed. Hence we search for some integer $k \in [a, b]$ such that $|I(v, k)|$ is as small as possible. We use the increasing operation described in the previous paragraph. The pseudo-code for this process in given in Algorithm 4.

This scan requires $O(d(v) \log V)$ time instead of $O((1 + b - a) \log V)$. On the other hand whenever the interval $[a, b]$ becomes empty during the verification procedure, because at some point we have $b < a$, the verification procedure terminates immediately by rejecting the transition. When all the known nodes in in(v) are greater than all the known nodes in out(v) it is only necessary to process $O(2-\ell)$ nodes to reach this rejecting verdict. This is the reason our optimization technique tries to maintain the adjacency lists of G sorted according to the topological order and the adjacency lists of G^T in reverse topological order.

```
1  Verify(V', v, ℓ)
2      a = −∞
3      b = +∞
4      while a ≤ b and
5      there are unprocessed vertexes in N(v) or N^T(v) do
6          Pick the next u in N(v) and update out(v)
7          or pick the next u in N^T(v) and update in(v)
8          while |I(v, a)| > 1 − ℓ and a ≤ b do
9              a = Next element in in(v)
10         end
11         while |I(v, b)| > 1 − ℓ and a ≤ b do
12             b = Previous element in out(v)
13         end
14     end
15     if a < b then
16         return k ∈ [a, b] that minimizes |I(v, k)|
17     end
18     return False
```

Algorithm 4: Pseudo code to determine $I(v, k)$.

5.3. Optimizing the initial state

In this section we describe a simple algorithm to select the initial state of the annealing process. We adapt an approximation algorithm for Vertex Cover. Usually the initial state of the annealing process is an empty set. Still it is possible to start the process at any other set V', provided the corresponding induced sub-graph is a DAG. Hence it is possible to use a simple and fast algorithm to choose this initial configuration, which the annealing process further improves. In Section 4.3 we showed that the max-DAG problem is Poly-APX-complete, see Corollary 1. This fact results from a reduction of the Independent Set problem. Therefore we are not focusing on an approximation algorithm for the max-DAG, but for its complement the directed feedback vertex set problem. The complement of an Independent Set is a Vertex Cover. Polynomial time algorithms that obtain a 2 approximation to VC are known, see Cormen, Leiserson, Rivest and Stein [22] or Garey and Johnson [13] or Papadimitriou and Steiglitz [38]. We adapt this algorithm to pick an initial state for our annealing algorithm.

Given a graph $G_D = (V, E_D)$ and an ordering of the vertices $s : V' \mapsto \{0, \ldots, |V'| - 1\}$ we can classify edges as forward or backward. An edge $(u, v) \in E_D$ is a backward edge iff $s(u) \geq s(v)$. Otherwise it is a forward edge. Recall that s is a bijection. Notice that this classification is not an intrinsic to an edge, because it is influenced by the ordering. Different orderings provide different classifications.

For our objectives the consequence of this classification is that the graph G_D is a DAG iff there is some ordering such that no edge is backward. Hence our approximation approach consists in removing vertices from G_D so that the resulting graph has no backward edges.

Consider the set of backwards edges $B_s \subseteq E_D$ and let V'' be a vertex cover of the graph composed of these edges, i.e., the graph given by (V, B_s). Note that the notion of vertex cover is usually defined on undirected graphs so assume that after we classify the edges its direction is ignored. Precisely, by a set cover we mean a set of vertices V'' such that for any backward edge $(u, v) \in B_s$ at least one of the vertices must be in V'', i.e., $u \in V''$ or $v \in V''$.

Our observation is that the sub-graph induced by $V \setminus V''$ is a DAG. This can easily be established. If this graph contained a cycle then at least some edge (u, v) in that cycle would be a backward edge and therefore either u or v would be in V'', which would cut the cycle.

We use a 2 approximation algorithm to obtain V'' in $O(V+E)$ time. This does not guarantee that we obtain a 2 complement approximation of the max-DAG problem. The reason being that the vertex cover we obtain depends on s. Hence to obtain a 2 complement approximation we would need to check all possible such orderings, which is impractical. Instead we use the ordering provided by Tarjan's algorithm. We show in our experimental study (Section 6.3) that the resulting algorithm can be far from a 2 complement approximation. Still the resulting max-DAG approximation is usually good.

Another approach for selecting the initial state for V' is to execute a BB algorithm with a fixed number of iterations, see Section 4.1 for a brief description of this algorithm. We refer to this algorithm approach as the hybrid algorithm in Section 6. This approach has a worst complexity bound than the heuristic algorithm we have just described. However, as we will show in the next section, it is very effective in practise for two reasons. First and most importantly the underlying SCCs are most of the time small, which makes approaches such as BB feasible. This has the added benefit that, contrary to SA, the BB algorithm may actually terminate. The SA algorithm never knows whether it has reached the global optimum, whereas BB does, given enough time. For small SCCs this is very likely to occur. The second important consequence of this hybrid approach, is that it provides a more efficient time allocation among SCCs, because all SCCs get an initial short slot of time. If the SCC is simple enough, this slot of time may actually be enough for BB to find an optimal solution, hence, the underlying BB algorithm terminates early, avoiding hogging the CPU unnecessarily. Otherwise the time bound is exceeded and the SCC passes to the SA phase.

5.4. Ordering reset

Resetting the search state is a technique that is sometimes combined with simulated annealing searches to avoid getting stuck in local maxima. In general a new random state is chosen and the procedure continues from there. A global reset where the current solution is discarded did not proved to be useful, in the graphs we tested. When the SA algorithm got stuck in a local maxima, because these maxima where highly probable, the restarts did not obtain better results. Otherwise the SA algorithm did not get stuck and therefore it was not necessary to employ restarts. Still we did found ordering resets to be useful in mitigating the shortcomings of the local search procedure described in Section 3. As mentioned in that section we keep a topological ordering s of the DAG induced by V'. It is this ordering s that we reset. Hence after a certain amount of operations we use an algorithm to determine another topological ordering of V'. This algorithm is a modified DFS and therefore requires at most $O(V+E)$ time. This reset is executed only after $2\times(V+E)$ vertices are tested. This means that essentially the reset operations amortize to a constant time overhead for each vertex v that is tested.

The reordering algorithm is the standard algorithm that sorts the vertices by decreasing finishing times, the only modification that we perform was to try to choose the restarts close to regions of the graph that have already been discovered. There is some randomness involved in the process, namely to avoid always obtaining the same ordering. In particular we randomly choose to compute the DFS on G or in G^T and do not always use the same order for the neighbors of a vertex.

6. Experimental evaluation

In this section we evaluate the previously described algorithmic techniques. More precisely, this study aims to answer the following main questions:

1. How does the temperature parameter affects the performance of the SA-based approach (Section 6.2)?
2. How effective is the heuristic proposed in Section 5.3 in identifying a high quality initial solution (Section 6.3)?
3. How does the graph complexity affects the performance of combinatorial approaches, and for which graph sizes do SA-based approaches outperform them (Section 6.4)?
4. How does the proposed SA approach perform in a wide variety of both synthetic graphs (Section 6.2, Sections 6.3 and 6.4) and real-world graphs (Sections 6.5 and 6.6)?
5. How does the proposed SA algorithm compares against alternative algorithmic approaches (Section 6.6)?

To answer these questions we developed two baselines. The first one is a BB approach (see Section 4 for implementation details). Despite BB does not exploring the entire set of solutions, as it prunes paths that are deemed to not improve the current solution, it is a combinatorial approach and still requires exponential time. The second baseline is a Greedy approach, which, unlike our SA algorithm presented in Section 3, never explores configurations that reduce the quality of the current solution and, as we will show further in our evaluation, is thus prone to converge to local optima instead of the global optimum.

Furthermore we compare the solution we proposed against two state-of-the-art approaches also based on SA, i.e., Galinier et al. [15] and Tang et al. [16].

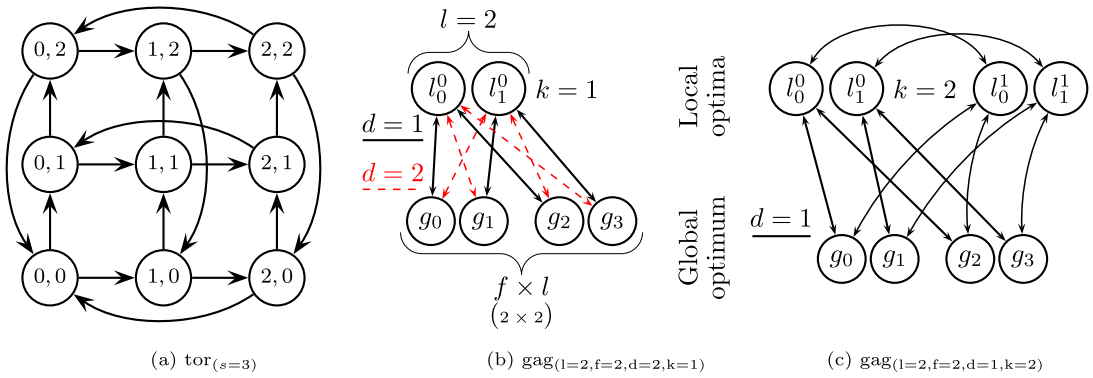
All algorithms were implemented in C/C++, including the prior SA-based solutions (Galinier et al. [15] and Tang et al. [16]) for which we used the original implementation provided by Tang et al. [16]. All experiments were executed in a Intel Xeon (E5-2660v4 @ 2.00 GHz) dual socket server with 64 GB of RAM, running Ubuntu 18.04 LTS with kernel versions 4.15.0-144-generic. The graphs are read from a file, where the first line contains the size of the graph (number of vertices and edges) and the following lines contain the adjacency lists. If not stated otherwise, we test at least 28 samples for each combination of algorithm and graph. The scatter plots present all the points collected from each sample and the box/whiskers plots present statistics from aggregating all samples (from top to bottom): maximum, 90% percentile, median, 10% percentile and minimum.

6.1. Graphs used in this experimental study

This section describes the various types of graphs used in our evaluation study.

Torus graphs. The torus graphs are defined according to a given parameter s. The set of vertices consists of pairs of modulo integers, i.e., $V=\{(i,j)\mid i,j\in\mathbb{N}_0\wedge i,j<s\}=\mathbb{Z}_s\times\mathbb{Z}_s$ Each vertex (i,j) has exactly two edges, an edge to $(i+1,j)$ and another to $(i,j+1)$. Note that we assume that the arithmetic is modulo s, therefore if $i+1=s$ then the destination vertex is actually $(0,j)$. Due to this, there exists a cycle for each column j. Likewise there are a set of edges to $(i,0)$, when $j+1=s$. These edges create cycles for each line i. Hence to obtain a DAG sub-graph V' it is necessary to exclude at least one vertex per column and at least one vertex per line. In short, a total of s vertices (that could be visually seen as a diagonal) need to be excluded in order to construct the max-DAG. Fig. 7(a) depicts an example for $s=3$. Note that the arrangement of the vertices matches the previous formula. One can see that a set containing 1 vertex per column and 1 per line is a minimum feedback vertex set of that graph.

Greedy adverse graphs. The second class of graphs that we consider are Greedy adverse graphs (gag) that are synthetically designed to serve as hard/adversarial problems for greedy approaches. More in detail, these graphs are designed as presented in Fig. 7. These graphs are designed to contain k local optima and



(a) $\text{tor}_{(s=3)}$ (b) $\text{gag}_{(\text{l=2,f=2,d=2,k=1})}$ (c) $\text{gag}_{(\text{l=2,f=2,d=1,k=2})}$

Fig. 7. Fig. 7(a) depicts a Torus (tor) graph with $s = 3$. The minimum feedback vertex set must contain one vertex per column and one vertex per line (e.g., the diagonal). Figs. 7(b) and 7(c) illustrates how the gags are constructed. Note that edges in the gag are bi-directional. The max-DAG in gags is composed by all the vertices in the global optimum.

one global optimum, depicted respectively above and below in Fig. 7(b) and 7(c). Each of k local optima is a set of l vertices, labeled in the figure as l_i^j where $i \in [0, l-1]$ and $j \in [0, k-1]$, with no edges among them. The global optimum is composed by $f \times l$ vertices, labeled in the figure as g_i where $i \in [0, f \times l - 1]$ also with no edges among them.

The parameter d is used to control the degree of adversity of the graph to greedy algorithms, which corresponds to the number of vertices that need to be discarded from the local optimum in order to include one vertex of the global optimum (and attain a DAG). To ensure this, each vertex l_i^j is bidirectionally connected to the vertices with identifiers $g_{(i+d')+f' \times l \mod f \times l}$, where $f' \in [0, f-1]$, $d' \in [0, d-1]$. Vertices from different local optima are also bidirectionally connected, i.e., a vertex l_i^j is connected to all vertices l_i^m, $m \in [0, k-1]$, $m \neq j$. Thus, a DAG constructed by picking vertices from the local optima cannot contain more than l vertices. Fig. 7 provides two examples of this class of graphs, where we have set $l = 2, f = 2, d = 2, k = 1$ (Fig. 7(b)) and $l = 2, f = 2, d = 1, k = 2$ (Fig. 7(c)). Note that these graphs, as for the case of the torus graphs, have known global optima. This knowledge can be exploited to evaluate how far away the solutions output by the algorithms being evaluated are from the optimum solution.

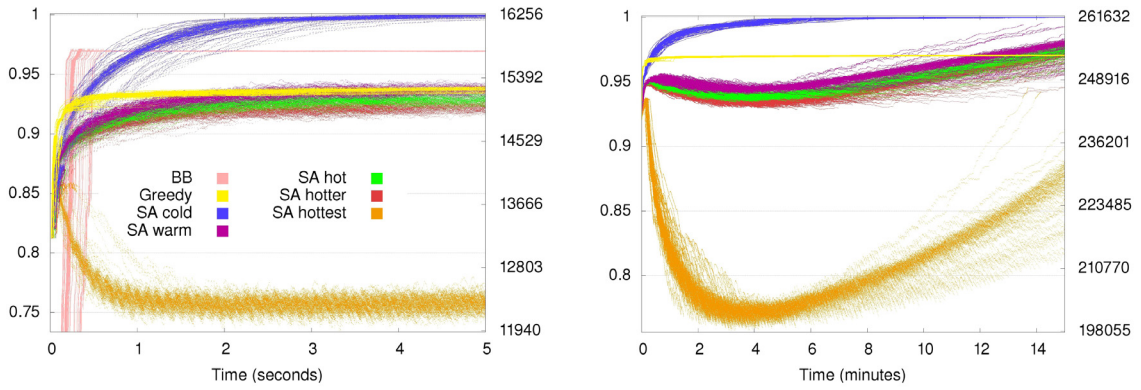
Conflict graphs of TM applications. In our evaluation (Sections 6.5 and 6.6) we also make use of graphs obtained when using real-world applications, namely applications that rely on TM to regulate concurrent access to shared data. In the following we explain more in detail the TM applications used and how the graphs were generated. The transaction abstraction in TM (and in databases [39]) is used to encapsulate a set of memory accesses (i.e., reads/writes) generated by concurrent threads and that need to be executed with atomic semantics [40] (i.e., intuitively, equivalently to having taken place instantaneously). If two transactions concurrently access the same memory regions, and one of the accesses is a write operation, then the two transactions are said to conflict. The conflict relations among transactions can be naturally encoded using a, so called, conflict graph, whose vertices are represented by transactions and whose edges encode conflict relations between pairs of transactions. One of the fundamental results in the theory of transactional computing is that a concurrent execution history of a set of transactions is guaranteed to be serializable, i.e., explicable via a corresponding a sequential execution, iff the corresponding conflict graph is acyclic [41]. Thus, solving the FVS problem for a transaction conflict graph allows for identifying the minimum number of transactions that has to be aborted in order to ensure that the corresponding concurrent execution is serializable.

The TM graphs adopted in this study were constructed using traces of read-/write-sets produced during the execution of two applications: TPC-C [42] and Genome [43].

TPC-C [42] is a benchmark originally proposed for On-Line Transaction Processing (OLTP) in database systems. This benchmark portrays the activity of a wholesale supplier and includes a mix of five concurrent different transactions operating over nine tables. In this study we set the distribution of transactions as follows: 45% New Order, 43% Payment and 4% of each of the remaining transactions (Order Status, Delivery and Stock Level), we also set the number of warehouses to 1 (tpcc graph) and 30 (tpcc30 graph). For our study, we use a porting of TPC-C to operate in a TM environment, which maintains the tables prescribed by the TPC-C benchmark via in-memory data-structures. This TM-based TPC-C implementation has been already used to evaluate a number of TM implementations [44,45].

Genome is one of the benchmarks of the STAMP benchmark suite [43] for TM systems. Genome analyzes a large number of DNA segments and matches them, using the Rabin–Karp string matching algorithm, to reconstruct the original source genome. Firstly, a set of unique segments is created. Afterwards, each thread tries to remove a segment from a global pool of unmatched segments and add it to its partition of currently matched segments. By means of using transactions, threads can concurrently, yet safely, add to the set of unique segments and remove from the global pool of unmatched segments. We set the parameters of this benchmark as follows -g16384 (number of nucleotides) -s64 (sampled nucleotides) -n1048576 (number of segments).

We used TinySTM [46] as our TM implementation and instrumented it to log into a file the read and write set of transactions. Afterwards, we process these logs to generate the graphs. The number of vertices in the graphs correspond to the number of concurrent threads used in the benchmark. In order to collect the conflict graphs, we let each thread execute a transaction optimistically (i.e., without conflict detection at the TinySTM level) until they all reach commit time. At this point we log to file the read-set and write-set of all the concurrent transactions — which we use offline to generate an instance of a conflict graph — and rely on the TinySTM commit time logic to resolve conflicts and abort/restart, if needed, conflicting transactions. Note that TinySTM commit logic does not use a graph-based approach to ensure consistency, but a simple/lightweight heuristic that, unlike exact solutions of the FVS problem, can generate spurious aborts (i.e., unnecessarily abort transactions). Yet, graph-based concurrency control mechanisms that adopt a transaction execution model analogous to the one used in our work to extract the conflict graphs (and rely on FVS/max-DAG algorithms to resolve conflicts) do exist in the recent literature in the area Ding et al. [11].



(a) Torus with $s = 128$ (b) Torus with $s = 512$. BB runs out of memory after ≈16s.

Fig. 8. Finding the max-DAG for a torus graph. Higher is better. Simulated annealing works better using colder temperatures in this class of graphs. As shown, SA hottest drops the quality of the solution due to the temperature being set excessively high. The Greedy approach does not search for solutions that may reduce the quality of the solution, hence, being prone to be locked in some non-optimal solution. The temperatures used can be found in Table 3. The left y-axis shows the normalized quality of the obtained solutions w.r.t. the best possible solution, while the right y-axis shows the size of the current solution.

Table 2
Graphs generated using TM-based applications.

	tpcc	tpcc30	genome (1)	genome (2)
Total number of vertices	4096	4096	4096	4096
Total number of edges	4745007	715937	4200	50218
Number of complete SCCs	19	129	3139	586
Number of non-complete SCCs	1	3	124	1
Tot. no. vertices in non-complete SCCs	1876	2401	564	3033
Tot. no. edges in non-complete SCCs	37707	453547	1253	36092
Avg. density[a] of non-complete SCCs	0.01071983	0.02823978	0.00461070	0.00392473
Std. Dev. density of non-complete SCCs	N/A	0.37376849	0.1368112469	N/A

[a]This value is a weighted average of all the non-complete SCCs in the graph (i.e., SCCs with more vertices have higher weight).

Table 2 reports the characteristics of four graphs, two generated using TPC-C and two using Genome, which we are going to use in the rest of this evaluation study. The four graphs were generated using 4096 concurrent threads, which yields graphs that contain 4096 vertices. As we can see by the data in the table, these graphs are quite heterogeneous for what concerns the number of SCCs that they contain (ranging from 20 to more than 3000), as well as for the characteristics of their SCCs. In particular, these graph contains a significant number of complete SCCs, which are composed either by clusters of mutually conflicting transactions or by individual transactions that did not develop any conflict. The table reports information also describing the characteristics of the non-complete SCCs included in these graphs. By the data in the table, we note that the topologies of non-complete SCCs vary significantly, when evaluated in terms of total number of vertices and edges that they contain, as well as based on their average density, where we define as density of an SCC the ratio between the number of edges in that SCC divided by the number of edges of a complete SCC with the same number of vertices (i.e., $|V|^2 - |V|$)

Note that this class of graphs does not have a priori known solutions. Hence, in Section 6.5 we will report the time that a given algorithm takes to reach (or surpass) the best solution identified by any run of any algorithm.

Other graphs used in previous works. Finally, in Section 6.6, in which we compare with previous SA-based solutions, namely, the works of Galinier et al. [15] and Tang et al. [16], we employ the data sets originally used in those works, which can be found in Pardalos et al. [47], as well as the above described graphs generated by TM applications.

Table 3
Temperatures used in gag and tor graphs.

Designation in plots	Initial			Final		
	ΔE	p	T	ΔE	p	T
Hottest	2	0.1	0.861	1	0.001	0.112
Hotter	1	0.009747	0.176	1	0.0001	0.0814
Hot	1	0.0078	0.167	1	0.00008	0.0793
Warm	1	0.00585	0.158	1	0.00006	0.0768
Cold	1	0.000001	0.0528	1	0.00000001	0.0391

The temperature T decays linearly during the execution and it is calculated in function of ΔE and p as follows: $T = \Delta E/(\log_2(1/p) - 1)$. More details on simulated annealing can be found in Section 4

6.2. Tuning the temperature

We start by evaluating the impact of the temperature in SA algorithm when used in large scale torus and greedy adverse graphs. To this end, we consider five different settings for the temperature, whose parameters are defined in Table 3.

Fig. 8 reports the results obtained with the Torus graphs when setting the scale parameter s to 128 (Fig. 8(a)) and 512 (Fig. 8(b)). In addition to the SA algorithm presented in this work, we include in these figures also the BB approach, — an exact method (see Section 4) that requires exponential time — and a greedy algorithm

Table 4
Greedy adverse graph ($l = 32$, $f = 4$, $d = 5$ and $k = 11$).

	Time interval											
	[0 s,1 s[				[1 s, 5 s[				[5 s, 20 s]			
	<l	=l	>l	=g	<l	=l	>l	=g	<l	=l	>l	=g
SA hottest	0.050	0.709	0.100	0.141	0	0	0	1	0	0	0	1
SA hotter	0.032	0.794	0.143	0.031	0	0.810	0	0.190	0	0.825	0	0.175
SA hot	0.035	0.787	0.142	0.036	0	0.810	0	0.190	0	0.821	0	0.179
SA warm	0.033	0.786	0.149	0.032	0	0.807	0	0.193	0	0.816	0	0.184
SA cold	0.035	0.767	0.162	0.036	0	0.783	0	0.217	0	0.812	0	0.188
Greedy	0.394	0.455	0.151	0	0.285	0.533	0.182	0	0.283	0.532	0.185	0
BB	0.006	0.994	0	0	0	1	0	0	0	0.885	0.115	0

The table contains the distribution of solutions over time (80 samples), organized as follows: <l, solutions worse than local optima; =l, solutions at some local optimum; >l, solutions better than local but worse than global optimum; and, =g, global optimum solutions.

that never explores configurations that would lower the quality of the current solution.

The results show that the setting of the temperature can impact significantly the quality of the obtained solution and that, for the hottest considered configuration, the convergence speed can be quite severely hindered. In fact, in torus graphs, colder temperatures provide the best results, strongly outperforming every other alternative.

The BB method runs out of memory in the largest scale Torus (Fig. 8(b)) and is stuck in a solution that is approximately 4% from away from global optimum in the smaller scale Torus (Fig. 8(a)). In both graphs, the Greedy solution, as expected, is trapped in a local minimum that is approximately 6%/4% away from the global optimum for the small/large scale torus graph, respectively.

Besides the Torus graph, we also experimented using different temperatures with the greedy adverse graph, whose results are shown in Table 4. This gag as the following settings: $l = 32, f = 4, d = 5, k = 11$. We collected the current DAG size over execution time and reported the distribution of solutions worse than local optima (<l), at some local optimum (=l), better than the local but worse than the global optimum (>l), and at the global optimum (=g). The results are then split in 3 time intervals, from the start of the run until 1 s, 1 s to 5 s and after 5 s until the end of the run (at ∼20 s). For a given solution, and a given time interval, the larger the probability mass that is concentrated on top quality configurations (i.e., closer to =g) the better the performance of the algorithm. We marked with a gradient of colors the column =g to better visualize the quality of the returned DAGs. Color green indicates that the global optimum is identified with high probability, while yellow and red indicate lower probabilities (i.e., the algorithms are reporting inferior solutions).

Let us first focus on the period [5 s, 20 s[. By using "hottest" temperature settings, the SA identifying the global maximum (=g) in all the runs (hence, probability 1 is reported). Indeed, SA hottest had already identified the global maximum in all the runs in the [1 s, 5 s[interval. Conversely, SA with colder temperature settings is more likely to converge to some local optimum. In the case of SA cold, 81.2% of the obtained solutions had the same quality as some local optimum (=l). Greedy also quickly converges into some local optimum (=l). However, differently from SA algorithms, Greedy was not capable of obtaining the global optimum.

In the period [0 s, 1 s[, one can see that all SA approaches report approximately the same distribution. The reason, as we will show in the next section, is the initialization process, which, at least in this type of graphs, tends the increase the likelihood that the local search process becomes trapped into local minima. Colder temperatures have more difficulties in "escaping" from local minima (Greedy never finds the global optimum), while the "hottest" temperature, which will accept exploring solutions of relatively lower quality, succeeds in avoiding this issue. The distribution in the [1 s, 5 s[interval is not much different from then one in [5 s, 20 s[, which indicates that the different algorithms have already converged.

6.3. Choice of the initial solution

In Section 5.3 we proposed to adopt as initial solution a configuration obtained via a 2-approximation of the vertex cover. Fig. 9 illustrates the trade-offs associated with such initialization method in SA and Greedy, which are compared against solutions that start constructing the max-DAG from a single vertex (selected at random). The plots are obtained two Torus graphs of different scales ($s = 128$ and s=512). The plots clearly show that the algorithms initialized with this heuristic manage to identify higher quality solutions in a shorter period of time. It is also interesting to observe that in Greedy, which is not allowed to drop the quality of its current solution, the adoption of this initialization strategy, although beneficial in the short term, may degrade the final solution.

In Fig. 9(a) "Greedy (no init)" manages to surpass "Greedy" with initialization after ≈15 s, as "Greedy (no init)". In the larger graph (Fig. 9(b)) such effect is only observed after much more than 15 min, which is the time allowed for each of the samples to run, i.e., it is visible that the "Greedy (no init)" still has not converged to some maximal solution. However, if the time is limited, as in this case, the initialization may provide significant benefits.

To conclude the study on the solution initialization, Table 5 presents a study on the same instance of a greedy adverse graph presented in the previous section. We will now focus on the SA hottest, SA cold and Greedy approaches, with and without initialization.

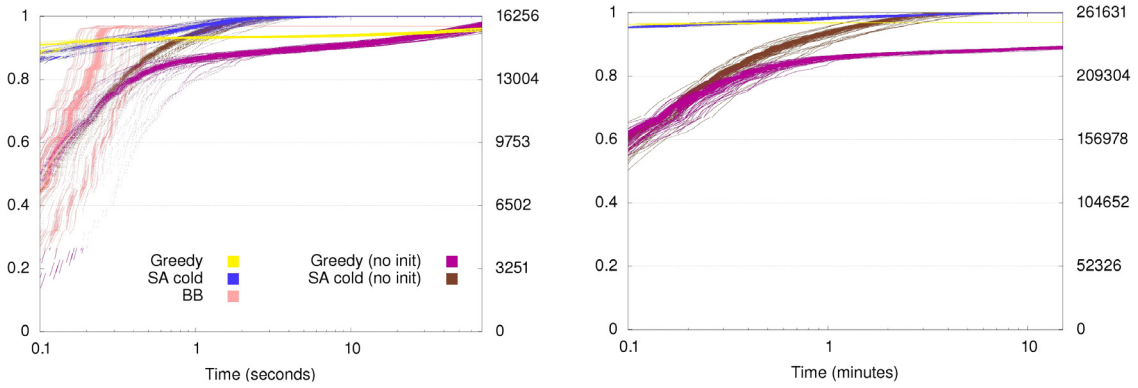
Analogously to the Torus graphs, the initialization in the Greedy approach has the adverse effect of hampering the quality of the solutions to which Greedy converse. In fact, after 5 s, Greedy identifies solutions with quality better than a local optimum (>l) with probability 0.185, while "Greedy noinit" 63.4% of the times. A similar pattern can be observed also in the other time intervals, i.e., [0 s, 1 s[and [1 s, 5 s[, which indicates that the Greedy approach has a higher probability to converges quickly to some solution and to remain stuck with that solution until the end of the run. Recall that, as explained in Section 6.1, this type of graphs is constructed in such a way that, in order to increase the quality of the solution identified in the long term, it is necessary to accept in the short term solutions of lower quality — which makes the performance of greedy algorithms particularly sensitive to the choice of the initial configuration.

Although the initialization step is not affecting SA hottest, it does affect SA cold. SA cold without initialization obtains the global optimum 37.7% of the times, while SA cold only obtains the global optimum 18.8% of the times. This can be explained by considering that colder temperatures are less prone to accept

Table 5
Same graph as in Table 4 (gag $l = 32, f = 4, d = 5, k = 11$).

	Time interval											
	[0 s,1 s[				[1 s, 5 s[				[5 s, 20 s]			
	<l	=l	>l	=g	<l	=l	>l	=g	<l	=l	>l	=g
Hottest noinit	0.056	0.666	0.134	0.144	0	0	0	1	0	0	0	1
Hottest	0.050	0.709	0.100	0	0	0	0	1	0	0	0	1
Cold noinit	0.040	0.563	0.312	0.086	0	0.576	0	0.424	0	0.623	0	0.377
Cold	0.035	0.767	0.162	0.036	0	0.783	0	0.217	0	0.812	0	0.188
Greedy noinit	0.318	0.110	0.571	0	0.251	0.119	0.631	0	0.247	0.119	0.634	0
Greedy	0.394	0.455	0.151	0	0.285	0.533	0.182	0	0.283	0.532	0.185	0

Added the variants without initialization (solutions start from any vertex and then construct the DAG).



(a) Torus $s = 128$ (b) Torus $s = 512$. BB runs out of memory after ≈16s.

Fig. 9. Finding the max-DAG for a torus graph (note that time in x-axis is in logscale). The proposed initialization method allows for effectively enhancing the initial performance of the algorithm with both SA cold and Greedy. With Greedy, the initialization methods leads to a degradation of the quality of the final configuration identified algorithm for the smaller scale graph (Fig. 9(a)). Conversely, the time taken by SA cold to reach the global optimum is roughly unchanged with or without initialization.

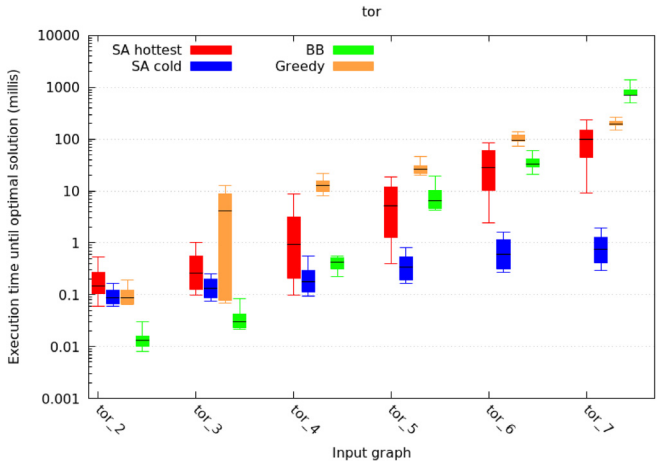


Fig. 10. Time taken by different algorithms to identify the optimal solution (note the logscale for the y-axis). At 49 vertices (tor_7), BB is clearly slower than SA.

lower quality configurations (i.e., thus being more prone to become stuck in a local minimum) and, as already discussed for the case of Greedy, the proposed initialization heuristic tends, with this graph, to increase the likelihood to remain trapped in local minima.

6.4. Evaluation of SA, BB and greedy algorithms on small scale graphs

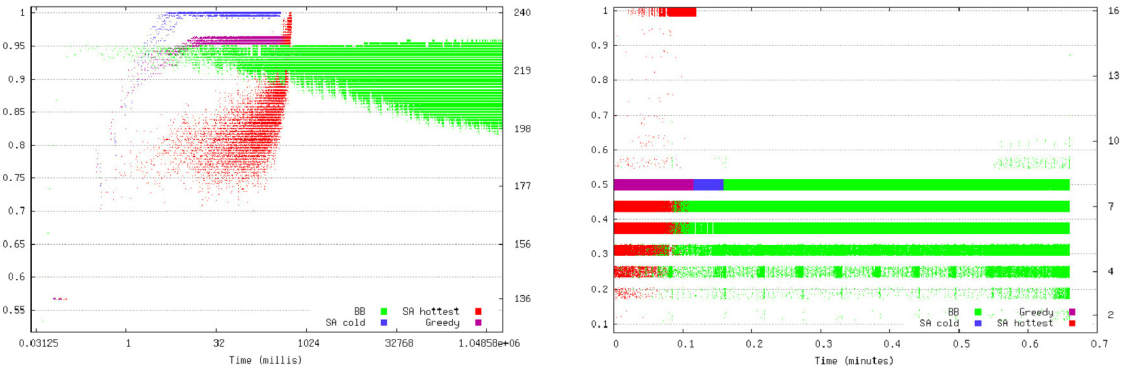
As we will see in the next section, in real-world applications the target graphs are often composed by a number of SCCs that have heterogeneous sizes. In the TM applications considered in Section 6.5, for instance, the corresponding conflict graphs tend to contain a small number of large SCCs (containing several thousands of vertices) and a much larger number of relatively small SCC (i.e., with less than 16 vertices).

Motivated by this observation, in the following we evaluate the performance of the SA, BB and Greedy algorithms with graphs that have a smaller scale than the ones considered so far.

We start by presenting the solutions obtained in small graphs by SA, BB and Greedy algorithms. Fig. 10 presents the performance of the different algorithms for the torus graph from $s = 2$ up to $s = 7$. As shown in this figure, BB is capable of finding the optimal solution in time comparable with the other algorithms for $s < 7$, with the exception of SA cold, which provides the optimal solution in less time for $s > 3$. For example, SA cold takes $\sim 60\times$ less time than BB in $s = 6$, but BB takes a similar amount of time to discover the optimal solution when compared with SA hottest. It is worth noting that in this scenario (where the problem is tractable), BB has the advantage of knowing when to stop (i.e., whether it has already identified the global optimum), while the other algorithms only stop after having exhausted their time budget.

Fig. 11(a) depicts the obtained solutions for a torus with $s = 16$. In this scenario, BB takes a considerable amount of time to complete its execution (≈17 min), and, as one can see, early truncating the BB execution may result in poor solutions (below the quality of SA or Greedy). SA and Greedy manage to get to the optimal solution under ≈1 s.

Fig. 11(b) considers a gag with 32 vertices in which the d parameter is set to a relatively high value, i.e., 3. Recall that this parameter controls how likely it is for Greedy or SA approaches with cold temperature to be trapped in local optima. As expected, in these settings, SA with the hottest temperature settings is the best performing solution; SA cold and Greedy are the 2nd best



(a) Torus with $s = 16$ (256 vertices), note that x-axis is in logscale

(b) Greedy adversarial graph with $l = 8$, $f = 2$, $d = 3$ and $k = 8$ (32 vertices)

Fig. 11. BB takes a considerable amount of time to get to the optimal solution even in modest size graphs. The temperatures used for SA can be found in Table 3.

performing solutions, achieving similar performances. BB is by far the worst solution.

Overall, the results discussed in this study clearly show that while, on the one hand, BB does not scale, it can actually be quite competitive with small scale graphs.

6.5. Conflict graphs generated by TM applications

In this section we evaluate SA cold and BB in graphs collected from TM workloads, described in Section 6.1.

Given that, as already mentioned when presenting the table in Table 2, these graphs contain a significant number of complete SCCs (i.e., SCCs in which each vertex is connected to every other in the same SCC), we included in all tested algorithms an optimization that detects in constant time whether an SCC is complete before starting to execute the actual optimization algorithm. To this end, we simply verify whether the number of edges ($|E|$) in the SCC equals $|V|^2 - |V|$ (where $|V|$ denotes the number of vertices in the graph). This is possible since in this scenario it can be excluded the existence of edges that start and end in the same vertex (a transaction can only conflict with another transaction). Once a complete SCC is identified, the optimal solution of the max-DAG problem for that SCC is trivially anyone of its vertices — sparing the cost of running any optimization algorithm.

Besides the SA cold and BB already evaluated in the previous sections, we also created a *hybrid* approach that combines both SA cold and BB. All algorithms are given a *budget*, i.e., a fixed number of operations to execute. In the case of SA cold, the budget is used to interpolate the linear decay of temperature and in the case of BB it is used to stop the computation after that number of operations have been done (a timeout is also possible). The key idea behind Hybrid is to first execute BB with a small *budget* (in these experiments 128 operations per SCC) and then continue the execution using SA with an initial solution computed by BB. In fact, SA cold can only stop after having exhausted its budget, as it does not store any state of previous solutions and has no way of knowing how far away the current solution is from the optimal one. On the other hand, BB can determine exactly when to stop, as it either explores the solution space or bounds bad configurations. For this reason, if an SCC is simple enough for BB, then BB will stop much earlier than SA cold. However, if some SCC is too complex, then BB will probably take much longer (exponential time) than SA cold to reach a "good enough" solution. With the hybrid approach we aim to strike a good trade-off between these two algorithms.

Fig. 12 presents a study on graphs generated from TM workloads. The absolute number of vertices in the current solution is reported on the right y-axis, whereas the left y-axis reports the quality of the current solution normalized to the best solution (i.e., the largest DAG found across all solutions). The x-axis presents the time that the solution takes to reach the solution quality in the y-axis.

The top plots show graphs from TPC-C with different workload configurations. On the top-left plot, the hybrid approach manages to reach the best solution faster than the other two solutions. The low BB performance is due to its poor efficiency when processing complex SCCs, SA cold is significantly better than BB and appears to be slowly approaching hybrid. The top-right plot tells a similar story, but in this case SA cold obtains, between $\sim$ 20 and $\sim$ 50 milliseconds, better solutions than Hybrid. Similar effects can also be observed in the bottom-left plot. Hybrid is the overall best performing solution, being outperformed only in the initial phases of some runs by SA cold. However, in this case, the SCCs are much simpler to solve, which makes BB more competitive when compared to the previously analyzed scenarios.

Finally, the bottom-right plot shows the importance of tuning the budget parameter of the Hybrid algorithm. Because Hybrid uses an initial budget of 128 operation to obtain an early solution with BB, which, in this case, is a too short budget, Hybrid fails to obtain a good solution to initialize SA cold with. In this case, Hybrid behaves very similarly to SA cold, which is actually good up to the $\sim$ 9 minutes point. Soon after this point, the "pure" BB approach finally finds a better solution.

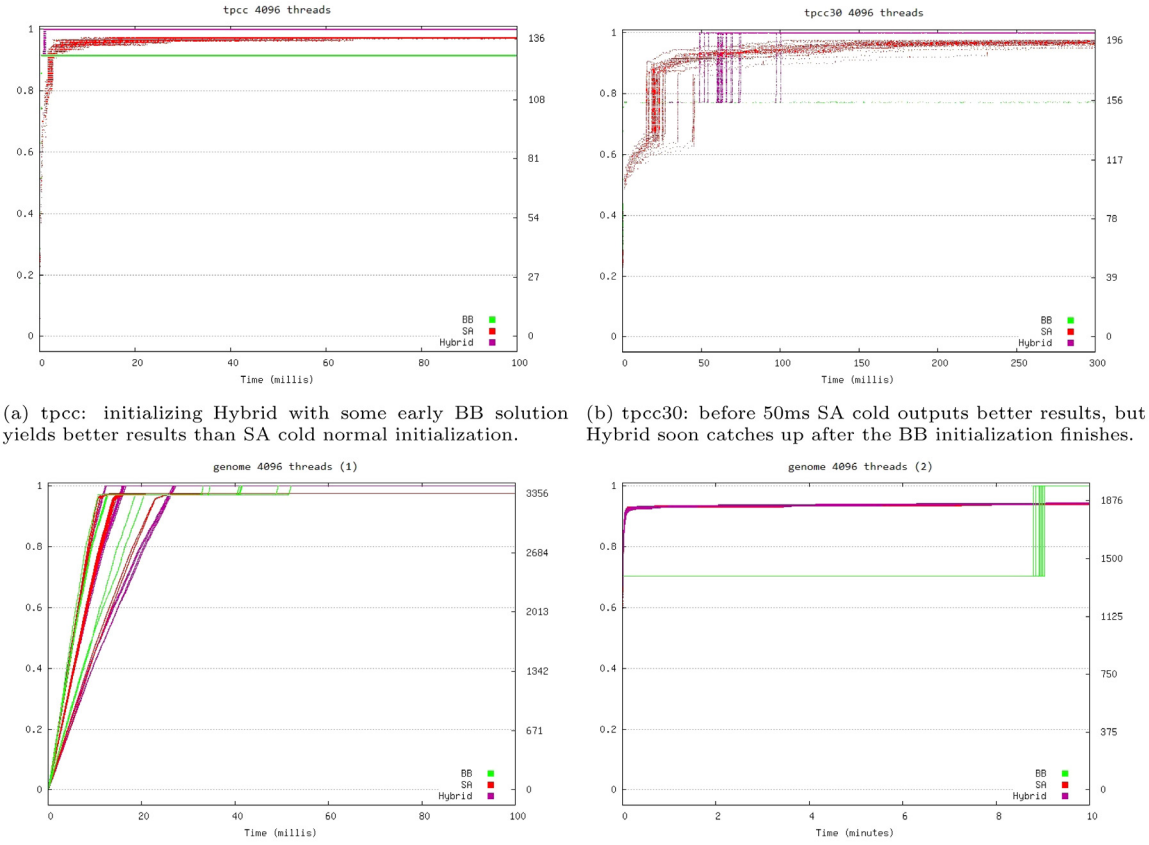
6.6. Comparison with other SA-based algorithms in the literature

To the best of our knowledge there exist two works that proposed simulated annealing algorithms to solve the minimum FVS problem, namely, Galinier et al. [15] and Tang et al. [16].

We obtained the original code from Tang et al. [16], which contained also the implementation of the approach by Galinier et al. [15]. Both these algorithms are implemented in C/C++, just like the solution presented in this paper. Besides the graphs of Tang et al. [16] (data set was collected by Pardalos et al. [47][2]), we also evaluate all the considered solutions using the TM conflict graphs presented in Section 6.1.

We start by reporting in Table 6 the average and standard deviation over 28 runs of the number of vertices included in the solution by each solution on each considered graph (one graph per row). The input graph is shown in the leftmost column of Table 6. The results reported in Table 6 were obtained by allocating a fixed amount of time, namely 1 s, to each solution. In this case we are considering the FVS problem, recall that smaller

2 Publicly available at http://mauricio.resende.info/data/



(a) tpcc: initializing Hybrid with some early BB solution yields better results than SA cold normal initialization.

(b) tpcc30: before 50ms SA cold outputs better results, but Hybrid soon catches up after the BB initialization finishes.

(c) genome (1): The algorithms achieve similar performance, though Hybrid is the first to obtain the best solution (before 30ms), while BB obtains the best solution only after 30ms.

(d) genome (2): Hybrid and SA cold perform similarly. BB eventually (after 9min) discovers the best solution while SA cold and Hybrid are still improving their solution.

Fig. 12. Study on graphs created from Transactional Memory (TM) transaction dependencies (Read/Writes and Write/Write conflicts) in commonly used applications. The TM implementation used is TinySTM by [46]. All graphs have 4096 vertices and a variable number of complete/non-complete SCCs and edges.

numerical values imply better quality solutions. In this table, the last 4 rows correspond to the TM graphs (see Section 6.1), whereas the other rows correspond to graphs originally used to evaluate the solution by Tang et al. [16]. For these graphs, the table reports only the number of vertices and edges that they contain.

The SA approach proposed in this work is configured with the cold and warm temperatures shown in Section 6.2. As for the approaches by Galinier et al. [15] and Tang et al. [16], we employ the default parametrization defined in the code. In more details, Galinier et al. [15] has two parameters to control the temperature, namely, the initial temperature, T_0, and a parameter, α, that regulates the geometric decay over time. T_0 is set to 0.6 and α to 0.99. Tang et al. [16] has an extra parameter for the probability distribution that controls how often a move (i.e., a possible modification to the current DAG) is picked. This parameter, θ, is set to 5. There are two other parameters to control the number of iterations. The parameter for the number of trials is set to $5 \times |V|$. These algorithms repeat a trials step until their solutions stop improving. However, given that we are using a time limit as stop condition, to ensure a fair comparison, the parameter that regulates how many times the trials step need to be repeated is set to infinity (i.e. until the program receives a timeout).

Table 6 shows that "SA cold" and "Hybrid" yield FVSs that are, in absolute sizes, smaller consistently smaller than all the considered baselines (except for the smallest considered graph, where the differences are anyway negligible). Considering the average of all graphs, SA cold obtains FVSs 23.4% smaller when compared to Tang et al. [16], with the largest gains achieved in the graph $|V|, |E| = 1000, 3000$, where the Hybrid and SA cold produce approximately $2\times$% larger FVSs. Regarding the remaining approaches, SA cold obtains FVSs 28.2% smaller than Galinier et al. [15], 11.2% when compared to Greedy, 15.5% when compared to SA warm and has very similar FVSs to Hybrid (< 0.01% difference on average across all graphs).

Fig. 13 provides us with a different perspective on the performance of the compared solutions, by reporting the distribution of the percentage of distance from optimum (where the optimum is defined as the best solution identified by any algorithm) achieved by each solution when considering three different time budgets, namely 10 ms, 100 ms and 1 s. Unlike the data in Table 6, which reports data on the absolute sizes of the FVS identified by each algorithm, the plot in Fig. 13 allows us to estimate how far away each approach is relatively to an idealized exact solution. This perspective allows for clearly visualizing the benefits provided by the approach presented in this work across different time scales and considering different statistical indicators. More in detail, for the time limits 10 ms, resp., 100 ms and 1 s, SA cold is on average closer to the minimum FVS by a factor of $1.4\times$, resp., $9.3\times$ and $13.3\times$ than the solution by Tang et al. [16]. Similar gains are observed also comparing the solution quality at different percentile levels. For instance, considering the 95-th percentile, the relative improvement of SA cold vs Tang et al. [16] is $1.6\times$, $9.6\times$, and $10.8\times$ better at 10 ms, 100 ms and 1 s, respectively.

7. Conclusions and further work

In this paper we studied the feedback vertex set problem in directed graphs. We approached the problem from its dual

Table 6
The table reports the average size of the FVS identified by the algorithms of [15], [16], SA warm, SA Cold and hybrid with a 1 s time limit. The smallest average FVS size is marked in green, the maximum is marked in red. The algorithms are given 1 s to output the result.

Graph	Galinier et al.		Tang et al.		Greedy		SA warm		SA cold		Hybrid	
$\|V\|, \|E\|$	Avg	dev	Avg	dev	Avg	dev	Avg	dev	Avg	dev	Avg	dev
50,100	3.0	0	3.0	0	3.6	0.49	4.1	0.94	3.1	0.30	3.1	0.22
50,150	11.3	0.64	9.5	0.50	11.2	1.01	11.6	0.92	9.8	0.70	9.8	0.54
50,200	17.3	0.70	15.5	0.50	15.3	1.67	15.8	1.44	13.2	0.40	13.0	0
50,250	21.6	0.86	19.0	0.50	20.4	1.31	20.8	1.34	17.4	0.57	17.5	0.80
50,300	24.1	0.54	21.9	0.30	23.3	1.10	22.2	1.60	19.2	0.40	19.2	0.40
50,500	33.0	0.67	31.2	0.36	30.9	0.91	31.2	1.83	28.1	0.30	28.1	0.30
50,600	35.6	0.49	34.4	0.48	35.5	1.16	35.6	1.36	32.0	1.02	31.6	0.74
50,700	36.8	0.40	35.1	0.44	35.8	1.41	36.2	1.44	33.3	0.43	33.5	0.80
50,800	38.3	0.64	37.9	0.36	38.7	0.78	38.2	1.81	34.1	0.30	34.1	0.22
50,900	38.8	0.43	38.1	0.30	39.5	0.59	38.7	1.93	36.1	0.30	36.1	0.30
100,200	12.4	0.48	10.1	0.22	10.4	0.66	13.0	1.48	9.9	0.79	10.0	0.80
100,300	29.0	0.95	24.0	0.74	22.6	1.39	24.4	2.03	17.6	0.66	18.1	0.86
100,400	38.5	0.87	32.8	0.87	28.8	1.50	30.7	2.17	23.8	0.68	24.1	0.83
100,500	49.0	0.74	43.2	0.93	39.9	1.84	40.9	1.92	33.7	0.79	33.8	0.83
100,600	55.1	0.74	49.5	0.59	43.2	1.12	47.0	2.66	38.0	0.89	38.3	1.00
100,1000	70.5	0.80	67.0	0.74	61.4	1.74	62.0	2.26	54.4	0.73	54.7	0.71
100,1100	71.9	0.96	68.1	0.70	64.5	1.40	63.5	2.42	55.7	0.78	55.8	0.87
100,1200	74.0	0.59	70.6	0.73	65.4	1.93	66.3	2.26	57.8	0.89	58.7	1.01
100,1300	75.5	0.59	72.5	0.97	67.7	1.31	68.8	2.46	61.5	1.12	61.2	0.87
100,1400	76.3	0.71	73.6	0.49	68.9	2.02	69.5	2.66	61.7	1.06	61.7	0.95
500,1000	84.2	1.44	59.1	2.17	42.3	1.52	52.4	3.62	36.9	2.06	36.0	1.34
500,1500	174.0	2.87	128.9	2.97	87.1	3.25	99.5	5.82	72.2	2.03	73.0	2.47
500,2000	238.2	2.48	196.4	2.22	129.7	3.55	148.8	5.56	112.1	1.88	112.0	2.20
500,2500	281.4	2.60	243.4	2.85	166.8	3.86	183.7	6.57	144.6	2.06	143.4	3.07
500,3000	314.6	2.06	278.2	3.20	203.0	4.27	215.0	5.47	172.8	2.51	173.6	2.91
500,5000	379.9	1.76	357.5	2.71	286.4	4.83	293.7	3.69	248.6	2.82	248.8	2.91
500,5500	391.1	1.47	371.1	1.40	301.7	4.29	310.6	5.54	264.6	2.29	264.5	3.09
500,6000	398.3	1.52	381.2	1.90	311.5	3.49	318.5	5.51	277.7	2.33	276.0	3.49
500,6500	403.9	2.90	389.8	1.08	323.1	4.74	330.1	7.47	288.2	1.77	288.4	3.04
500,7000	410.7	1.42	398.3	1.26	336.3	4.26	342.3	5.25	297.5	2.62	297.6	2.22
1000,3000	372.7	3.10	287.8	3.25	169.9	3.66	207.9	7.47	149.8	3.87	149.0	2.93
1000,3500	438.6	4.57	347.0	3.79	205.7	5.22	247.3	5.64	183.3	3.21	184.0	3.13
1000,4000	493.5	4.46	400.5	3.53	254.2	5.62	289.9	9.44	216.9	3.58	214.4	2.65
1000,4500	543.4	4.23	459.0	4.14	303.4	5.86	328.5	8.80	254.0	3.63	254.0	3.91
1000,5000	583.3	3.51	501.1	5.42	332.2	6.34	364.3	9.24	284.0	3.22	284.4	3.26
1000,10000	779.9	2.68	732.4	3.28	569.5	5.06	585.6	8.94	496.1	4.81	496.5	3.22
1000,15000	844.3	2.26	818.0	2.12	675.8	5.74	691.0	10.3	607.8	3.65	607.4	2.82
1000,20000	880.2	1.46	860.8	2.45	738.0	4.34	756.6	9.97	678.2	3.96	675.3	3.37
1000,25000	901.3	1.58	888.0	1.34	780.6	3.41	797.7	7.59	725.8	3.25	724.0	4.36
1000,30000	916.4	1.31	906.7	1.31	809.0	4.65	834.1	6.90	763.0	3.16	762.8	3.39
Genome (1)	745.0	3.02	690.6	3.28	668.0	4.98	683.7	5.44	656.7	1.28	655.1	0.30
Genome (2)	2902.6	12.5	2593.3	10.6	2291.0	4.99	2368.2	10.9	2245.3	3.60	2247.9	4.02
TPCC	3962.4	0.48	3961.6	0.49	3960.5	0.50	3961.6	0.74	3960.1	0.30	3960.1	0.22
TPCC30	3907.9	1.37	3902.4	0.80	3900.1	0.22	3902.2	1.42	3900.0	0	3900.2	0.36

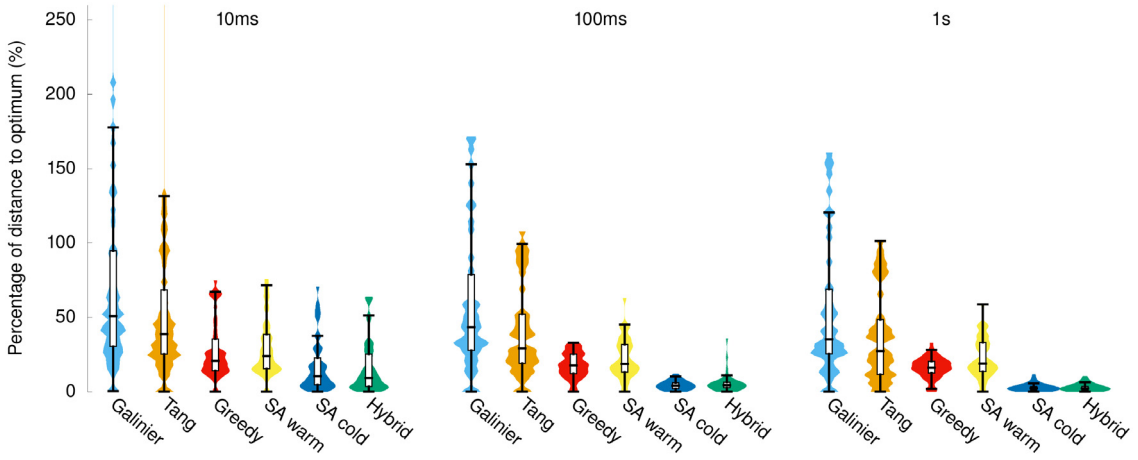


Fig. 13. Distribution of obtained feedback vertex sets (FVS). The different algorithms are given 10 ms, 100 ms and 1 s to solve the problem. The boxplot show the 5%, 25%, median, 75% and 95% percentiles. Compared against the solutions proposed by Galinier et al. [15] (labeled as Galinier) and Tang et al. [16] (labeled as Tang).

formulation, the maximum induced sub-DAG (max-DAG), which we showed to be Poly-APX-complete. Therefore, unless $P = NP$, no approximation algorithm exists.

We considered the simulated annealing approach by Galinier, Lemamou and Bouzidi [15] and proposed several improvements. We tested the resulting prototypes, which validated these proposals. We started by studying the limitations of the local search procedure by Galinier, Lemamou and Bouzidi [15], which sometimes rejects local transitions, even though the resulting graph would still be a DAG. This motivated our stochastic approach where nodes are chosen uniformly at random from available candidates, as opposed to the approach of considering all the possible transitions and ranking them, as used by Galinier, Lemamou and Bouzidi [15] and Tang, Feng and Zhong [16]. Our rationale is that the annealing process itself already has the property of forcing "bad" vertices out of the current configuration. Therefore, it does not seem sensible to invest a significant amount of time ranking the vertices to guarantee that the chosen transition is good. A good evidence of this removal of bad vertices at work is the difference in performance of SA algorithm versus the greedy algorithm. The greedy algorithm never allows for removals from its configuration. Therefore it never removes bad vertices. Hence the gap in performance between these algorithms allows us to estimate the effect of the removal of bad vertices by SA.

To further mitigate the limitation of the heuristic we considered higher temperatures and ordering resets. Overall, higher temperatures did not yield good results. Our extensive experimental results showed that this particular problem generally requires very cold temperatures in the large majority of tested graphs. In fact we tested this hypothesis by using greedy algorithms, which do not allow for vertex removal. The performance of these greedy algorithms was in general worse than the simulated annealing approach, but much better than SA with higher temperatures. We also used ordering resets to mitigate the heuristic bias.

We used efficient data structures to maintain the topological ordering, namely splay trees. This reduced the time of a stochastic step from linear in the number of vertices to logarithmic. We actually pushed this approach further by noticing that at low temperatures most transitions are rejected and therefore optimized the testing procedure even further by re-ordering the adjacency lists to match the current topological order. This yielded a fast procedure for transition rejection.

Another important component, at least in some graphs, was the initial reduction of the graph into connected components. In some cases the impact of this pre-processing step was so important that the resulting graphs could simply be solved by using a simple branch and bound approach. We thus proposed a hybrid approach that generally selected the best algorithm and obtained the best performance, at least with the tested inputs.

We evaluated our prototype extensively in order to validate these design decisions and finally also compared it against state of the art implementations [15,16]. Our algorithm obtains feedback vertex sets up to 13.3× closer to the optimal solution in a wide variety of graphs when compared to the state of the art SA algorithms. The results show that in general our algorithms obtained significant improvements particularly for larger graphs, where they are most relevant.

As future work, we are currently integrating the resulting software into a transactional memory system [10,45] and investigating the algorithm's suitability for efficient parallelization across several working threads. We are also working on improving the effectiveness of the order reset sub-routine, mentioned in Section 5.4, as improving this particular process seems likely to yield even better solutions, albeit at the cost of extra processing time.

CRediT authorship contribution statement

Luís M.S. Russo: Conceptualization, Algorithm design, Prototype development, Formal analysis, Writing, Funding acquisition. **Daniel Castro:** Prototype development, Data Curation, Writing, Experimental evaluation. **Aleksandar Ilic:** Conceptualization, Algorithm optimization, Experimental results analysis, Funding acquisition, Writing, Supervision. **Paolo Romano:** Conceptualization, Algorithm optimization, Experimental results analysis, Funding acquisition, Writing, Supervision. **Ana D. Correia:** Algorithm survey and review, Experimental results analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

We thank Tang, Feng and Zhong [16] that kindly allowed us to use their implementation.

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 and project NGPHYLO PTDC/CCI-BIO/29676/2017 and EuroHPC Joint Undertaking grant agreement No 956213 (SparCity).

References

[1] P.D. Seymour, Packing directed circuits fractionally,Vol. 15,no. 2, Springer Science and Business Media LLC, 1995, pp. 281–288, http://dx.doi.org/10.1007/bf01200760.

[2] J. Aracena, L. Cabrera-Crot, L. Salinas, Finding the fixed points of a boolean network from a positive feedback vertex set, Bioinformatics 37 (8) (2020) 1148–1155, http://dx.doi.org/10.1093/bioinformatics/btaa922.

[3] J.G.T. Zañudo, G. Yang, R. Albert, Structure-based control of complex networks with nonlinear dynamics, Proc. Natl. Acad. Sci. 114 (28) (2017) 7234–7239, http://dx.doi.org/10.1073/pnas.1617387114.

[4] F. Pan, P. Zhou, H.-J. Zhou, P. Zhang, Solving statistical mechanics on sparse graphs with feedback-set variational autoregressive networks, Phys. Rev. E 103 (2021) 012103, http://dx.doi.org/10.1103/PhysRevE.103.012103.

[5] Y.-C. Chow, W. Kostermeyer, K. Luo, Efficient techniques for deadlock resolution in distributed systems, in: 1991 the Fifteenth Annual International Computer Software & Applications Conference, IEEE Computer Society, 1991, pp. 64–65.

[6] R. Bar-Yehuda, D. Geiger, Approximation algorithms for the feedback vertex set problem with applications to constraint satisfaction and Bayesian inference, SIAM J. Comput. 27 (4) (1998) 942–959, http://dx.doi.org/10.1137/S0097539796305109.

[7] P. Festa, P.M. Pardalos, M.G.C. Resende, Feedback set problems, Springer US, 1999, pp. 209–258, http://dx.doi.org/10.1007/978-1-4757-3023-4_4.

[8] M.-A. Daigneault, J.P. David, Automated synthesis of streaming transfer level hardware designs, ACM Trans. Reconfigurable Technol. Syst. 11 (2) (2018) http://dx.doi.org/10.1145/3243930.

[9] A. Gharehgozli, C. Xu, W. Zhang, High multiplicity asymmetric traveling salesman problem with feedback vertex set and its application to storage/retrieval system, European J. Oper. Res. 289 (2) (2021) 495–507, http://dx.doi.org/10.1016/j.ejor.2020.07.038.

[10] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, L. Rodrigues, On speculative replication of transactional systems, J. Comput. System Sci. 80 (1) (2014) 257–276, http://dx.doi.org/10.1016/j.jcss.2013.07.006.

[11] B. Ding, L. Kot, J. Gehrke, Improving optimistic concurrency control through transaction batching and operation reordering, Proc. VLDB Endow. 12 (2) (2018) 169–182, http://dx.doi.org/10.14778/3282495.3282502.

[12] R.M. Karp, Reducibility among combinatorial problems, in: R.E. Miller, J.W. Thatcher, J.D. Bohlinger (Eds.), Springer US, 1972, pp. 85–103, http://dx.doi.org/10.1007/978-1-4684-2001-2_9.

[13] M.R. Garey, D.S. Johnson, Computers and intractability, in: A Guide to the, 1979.
[14] M. Yannakakis, Node-and edge-deletion NP-complete problems, ACM Press, 1978, http://dx.doi.org/10.1145/800133.804355.
[15] P. Galinier, E.A. Lemamou, M.W. Bouzidi, Applying local search to the feedback vertex set problem, J. Heuristics 19 (2013) 797–818.
[16] Z. Tang, Q. Feng, P. Zhong, Nonuniform neighborhood sampling based simulated annealing for the directed feedback vertex set problem, IEEE Access 5 (2017) 12353–12363, http://dx.doi.org/10.1109/ACCESS.2017.2724065.
[17] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to algorithms, third edition, third ed., The MIT Press, 2009.
[18] R. Sedgewick, K. Wayne, Algorithms (Fourth Edition Deluxe), Addison-Wesley, 2016.
[19] T. Roughgarden, Algorithms Illuminated (Part 1), in: Algorithms Illuminated, Soundlikeyourself Publishing, 2017.
[20] R. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (2) (1972) 146–160, http://dx.doi.org/10.1137/0201010.
[21] M. Sharir, A strong-connectivity algorithm and its applications in data flow analysis, Comput. Math. Appl. 7 (1) (1981) 67–72, http://dx.doi.org/10.1016/0898-1221(81)90008-0.
[22] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to algorithms, MIT Press, 2009.
[23] I. Razgon, Exact computation of maximum induced forest, in: L. Arge, R. Freivalds (Eds.), Algorithm Theory – SWAT 2006, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 160–171.
[24] F.V. Fomin, S. Gaspers, A.V. Pyatkin, Finding a minimum feedback vertex set in time $\mathcal{O}(1.7548^n)$, in: H.L. Bodlaender, M.A. Langston (Eds.), Parameterized and Exact Computation, Springer Berlin Heidelberg, 2006, pp. 184–191.
[25] V. Bafna, P. Berman, T. Fujito, A 2-approximation algorithm for the undirected feedback vertex set problem, SIAM J. Discrete Math. 12 (3) (1999) 289–297, http://dx.doi.org/10.1137/s0895480196305124.
[26] H.L. Bodlaender, On disjoint cycles, in: G. Schmidt, R. Berghammer (Eds.), Graph-Theoretic Concepts in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 1992, pp. 230–238.
[27] R. Downey, M. Fellows, Fixed parameter tractability and completeness, in: Congressus Numerantium, Vol. 87, 1992, pp. 191–225.
[28] R. Chitnis, M. Cygan, M. Hajiaghayi, D. Marx, Directed subset feedback vertex set is fixed-parameter tractable, ACM Trans. Algorithms 11 (4) (2015) http://dx.doi.org/10.1145/2700209.
[29] G. Even, An 8-approximation algorithm for the subset feedback vertex set problem, SIAM J. Comput. 30 (4) (2000) 1231–1252, http://dx.doi.org/10.1137/S0097539798340047.
[30] D. Lokshtanov, P. Misra, J. Mukherjee, F. Panolan, G. Philip, S. Saurabh, 2-approximating feedback vertex set in tournaments, in: Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA, 2020, pp. 1010–1018, http://dx.doi.org/10.1137/1.9781611975994.61.
[31] C. Papadopoulos, S. Tzimas, Polynomial-time algorithms for the subset feedback vertex set problem on interval graphs and permutation graphs, in: R. Klasing, M. Zeitoun (Eds.), Fundamentals of Computation Theory, Springer Berlin Heidelberg, Berlin, Heidelberg, 2017, pp. 381–394.
[32] F.-H. Wang, C.-J. Hsu, J.-C. Tsai, Minimal feedback vertex sets in directed split-stars, Networks 45 (4) (2005) 218–223, http://dx.doi.org/10.1002/net.20067.
[33] I. Razgon, Computing Minimum Directed Feedback Vertex Set In $O^*(1.9977^n)$, World Scientific, 2007, http://dx.doi.org/10.1142/9789812770998_0010.
[34] J. Chen, Y. Liu, S. Lu, B. Ośullivan, I. Razgon, A fixed-parameter algorithm for the directed feedback vertex set problem, Vol. 55, no.5, Association for Computing Machinery (ACM), 2008, pp. 1–19, http://dx.doi.org/10.1145/1411509.1411511.
[35] P. Crescenzi, A short guide to approximation preserving reductions, in: Proceedings of Computational Complexity. Twelfth Annual IEEE Conference, IEEE Comput. Soc, 1997, pp. 262–273, http://dx.doi.org/10.1109/ccc.1997.612321.
[36] C. Bazgan, B. Escoffier, V.T. Paschos, Completeness in standard and differential approximation classes: Poly-(D)APX- and (D)PTAS-completeness, Theoret. Comput. Sci. 339 (2–3) (2005) 272–292, http://dx.doi.org/10.1016/j.tcs.2005.03.007.
[37] M.M. Halldórsson, J. Radhakrishnan, Greed is good: Approximating independent sets in sparse and bounded-degree graphs, Algorithmica 18 (1) (1997) 145–163, http://dx.doi.org/10.1007/bf02523693.
[38] C.H. Papadimitriou, K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Courier Corporation, 1998.
[39] R. Ramakrishnan, J. Gehrke, Database Management Systems, third ed., McGraw-Hill, Inc., New York, NY, USA, 2002.
[40] R. Guerraoui, M. Kapalka, On the correctness of transactional memory, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, - PPoPP '08, ACM Press, New York, New York, USA, 2008, p. 175, http://dx.doi.org/10.1145/1345206.1345233.
[41] C.H. Papadimitriou, The serializability of concurrent database updates, J. ACM 26 (4) (1979) 631–653, http://dx.doi.org/10.1145/322154.322158.
[42] Transaction Processing Performance Council, TPC-C benchmark revision 5.11.0, in: Transaction Processing Performance Council, 2018.
[43] C.C. Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: Stanford transactional applications for multi-processing, in: 2008 IEEE International Symposium on Workload Characterization, IEEE, Seattle, WA, USA, 2008, pp. 35–46.
[44] P. Felber, S. Issa, A. Matveev, P. Romano, Hardware read-write lock elision, in: Proceedings of the Eleventh European Conference on Computer Systems, EuroSys '16, Association for Computing Machinery, New York, NY, USA, 2016, http://dx.doi.org/10.1145/2901318.2901346.
[45] D. Castro, P. Romano, J. Barreto, Hardware transactional memory meets memory persistency, J. Parallel Distrib. Comput. 130 (2019) 63–79, http://dx.doi.org/10.1016/j.jpdc.2019.03.009.
[46] P. Felber, C. Fetzer, P. Marlier, T. Riegel, Time-based software transactional memory, IEEE Trans. Parallel Distrib. Syst. 21 (2010) 1793–1807.
[47] P.M. Pardalos, T. Qian, M.G.C. Resende, A greedy randomized adaptive search procedure for the feedback vertex set problem, J. Combin. Optim. 2 (1998) 399–412.