Experimental Study of FPT Algorithms for the Directed Feedback Vertex Set Problem

Rudolf Fleischer, Xi Wu, and Liwei Yuan*

Fudan University; Software School, CSE Department and IIPL; Shanghai, China
{rudolf,wuxi,yuanliwei}@fudan.edu.cn

Abstract. We evaluate the performance of FPT algorithms for the directed feedback vertex set problem (DFVS). We propose several new data reduction rules for DFVS. which can significantly reduce the search space. We also propose various heuristics to accelerate the FPT search. Finally, we demonstrate that DFVS is not more helpful for deadlock recovery (with mutex locks) than simple cycle detection.

1 Introduction

In the *minimum feedback vertex set problem (*FVS*)* we are given a graph $G = (V, E)$ and we want to find a minimum number of nodes in V whose removal would make the graph acyclic. The problem is NP-complete on undirected (UFVS) and directed (DFVS) graphs. Since DFVS has some applications in compiler optimization [14,16] and database deadlock recovery (see [4]), it is important to solve this problem quickly. Therefore, the parameterized versions of FVS have recently been studied. In k-DFVS the input is a pair (G, k), where G is the graph and $k > 0$ is a parameter, and the task is to either find a set of at most k nodes blocking all cycles or to report that such a set does not exist. A parameterized problem is *fixed parameter tractable* (see [6]) if it can be solved in time $O(f(k)poly(n))$, where n is the number of nodes of G and f is an arbitrary computable function. Intuitively, a problem is in the class FPT if it can be solved efficiently for fixed parameter k. We denote the runtime by $O^\star(f(k))$, omitting the less interesting polynomial dependency on n. Obviously, we do not expect $f(k)$ to be polynomial for NP-hard problems.

If we can reduce (G, k) in time $g(k)$ to an equivalent parameterized problem (H, k'), where g is some computable function and k' only depends on k, such that the size of H is bounded by some function $h(k)$, then (H, k') is a *kernel* of (G, k). It is known that being in FPT is equivalent to having a kernel [6]. While a polynomial-size kernel immediately implies the existence of an FPT algorithm

* According to international standard, the authors are listed in alphabetic order; first-author order would be X. Wu, L. Yuan, and R. Fleischer. This research was supported by the Shanghai Leading Academic Discipline Project (project number B114), the Shanghai Committee of Science and Technology of China (nos. 08DZ2271800 and 09DZ2272800), and the Robert Bosch Foundation (Science Bridge China 32.5.8003.0040.0).

A. Fiat and P. Sanders (Eds.): ESA 2009, LNCS 5757, pp. 611–622, 2009.

(reduce the problem to its kernel, then solve the problem on the kernel by brute-force search), it is not known how to derive a polynomial-size kernel from a given FPT algorithm.

Wang et al. [16] first observed that DFVS is more difficult than UFVS. Bodlaender showed that UFVS is in FPT [1], and Thomasse recently gave a quadratic size kernel [15]. For planar graphs, there is even a linear size kernel [2]. On the other hand, the FPT status of DFVS had been an open problem for fifteen years [5] until Chen et al. recently proposed an FPT algorithm [4] based on branching. It is not known whether the problem has a polynomial size kernel.

Our work. We implemented the DFVS algorithm by Chen et al. [4] and tested it on random graphs with various values for edge density ed and optimum DFVS size k. Fig. 1(a) shows the runtime of the algorithm for graphs with $ed = 2$ and $k = 2, 4, 6, 8$. As expected, the performance degrades dramatically when the parameter k increases; for $k = 8$ we often observed a time-out (set to three hours in our experiments), in particular on low-density graphs. Data reduction rules are very attractive to speed up FPT algorithms because no branching is involved and they may sometimes lead to a good kernelization. Inspired by the recent kernelization techniques for UFVS by Thomasse [15], we propose four new data reduction (preprocessing) rules for DFVS: *Dummy Nodes*, *Chaining Nodes*, *Flower*, and *Shortcut*. We achieved another speed-up by starting Chen's FPT search not in an arbitrary initial configuration. We propose three heuristics to choose a good initial configuration: a simple greedy heuristic based on picking large-degree nodes, and two more sophisticated heuristics based on computing good approximations to the optimal DFVS solution [7].

We evaluated the performance of our data reduction rules and heuristics on randomly generated graphs with varying number of nodes, edge density, and optimal solution size. In particular, we analyzed the impact of each parameter and heuristic on the total runtime. We measured runtime, kernel size (since we do not have a kernelization yet, we actually measure the maximum problem size sent to standard min cut when solving the skew separator problem, an important step in Chen's algorithm [4]), memory usage, and recursion depth. Our data suggest that *Chaining Nodes*, *Flower*, and *Shortcut* reductions can significantly reduce the FPT search space enabling us to solve DFVS on graphs that are several orders of magnitude larger than what Chen's algorithm can handle (see Fig. 1). Overall, we think best approach for solving DFVS is to first apply our reduction rules, then use the *Big-Degree* heuristic to compute an initial configuration for Chen's FPT algorithm, which is then used to solve the problem.

When we started this study, our main goal was actually to evaluate the algorithms on real data from the purported main application of DFVS [4,7], namely deadlock recovery in multi-thread computing environments. However, due to restrictions of the parallel programming model, cycle detection, rather than DFVS search, is already sufficient for efficient deadlock recovery. For example, it was reported in a recent paper on deadlock immunity [10] that cycle detection only incurs a loss of efficiency of 6% in state-of-the-art multi-core systems, so there is no need to employ complicated FPT algorithms.

Structure of the paper. In Section 2, we give some basic definitions. In Section 3, we briefly review Chen's FPT algorithm for DFVS and propose our new reduction rules and heuristics to speed-up the algorithm. In Section 4 we discuss implementation details of the algorithms, the random graph generator, and we present and discuss our experimental data. In Section 5 we discuss the suitability of DFVS for the deadlock recovery problem for concurrent threads. We conclude the paper in Section 6 with some thoughts about future work.

2 Preliminaries

Let $G = (V, E)$ be a directed graph with nodes V and edges E. A *simple path* is a path with no repeated nodes. Two paths are *internally disjoint* if no node appears on both paths except maybe the end nodes. A *directed cycle* is a path ending at its start node. Similarly, a *simple cycle* is cycle in which no node appears twice. A directed graph is *acyclic* (DAG) if and only if it contains no directed cycle.

For a node u, a *u-flower of order k* is a set of k directed cycles that are pairwise disjoint except for their common node u. We call these cycles *petals*. To compute u-flowers in G, we split u into two nodes s and t, with s incident to all outgoing edges and t to all incoming edges. Then, the maximum flower size at u equals the maximum number of node disjoint paths from s to t in G, which we can easily compute using standard min-cut techniques. and *Menger's Theorem* for directed graphs [3] which states that the maximum number of internally disjoint paths from s to t equals the size of a minimum (s, t) node cut if s is not adjacent to t. In the following, we use $petal(u)$ to denote the maximum flower size at u. Note that $petal(u) = 1$ implies that there exists another node v such that *all* cycles containing u also contain v (i.e., v dominates u); v is the single cut node between s and t in Menger's Theorem.

3 Chen's Algorithm and Speed-Ups

Chen's algorithm is based on *iterative compression* [9]. Given an input (G, k), we arbitrarily choose an induced $(k + 1)$-node subgraph H of G and an arbitrary k-node subset I of the nodes in H. Note that I is a k-node feedback vertex set of H. Then we repeatedly add another node of G to H and I, until $H = G$. Note that whenever we just added a new node, I is a $(k + 1)$-node feedback vertex set of H. We then use the subroutine *FVS-Reduction* to *compress* I into a k-node feedback vertex set of H. Eventually, $H = G$ and I is a k-node feedback vertex set of G. If in some iteration we cannot find a k-node feedback vertex set for H, we conclude that G does not have a k-node feedback vertex set. We can improve the runtime in two ways: use data reduction rules, and use heuristics to find a good initial configuration for the algorithm.

Data reduction rules. We preprocess the given graph by applying different rules (in arbitrary order and as long as possible) to reduce the size of the graph and the parameter k. Essentially, this reduces the search space for the FPT algorithm

later, whose runtime is exponential in k. Initially, we set the feedback vertex set $I = \emptyset$.

Rule 1 (*Self-Loop*). If u has a self-loop, we add u to I, decrease k by 1, and delete u and adjacent edges from the graph.

Rule 2 (*Edge Canonicalization*). If there are two nodes u and v with multiple edges from u to v we remove all but one edge.

Rule 3 (*Dummy Nodes*). If there is a node u without incoming edges or without outgoing edges, then we remove u and its incident edges from the graph.

Rule 4 (*Chaining-Nodes*). If there is a node u such that (v, u) $((u, v))$ is the only incoming (outgoing) edge, then we merge u with v.

It is easy to see that these four rules are *safe*, i.e., the reduced instance has a solution if and only if the original problem has a solution. Rule 1 and Rule 2 are already implicit in Chen's algorithm. Rule 3 and Rule 4 are the directed versions of similar rules for the undirected case [15]. The next rules are new.

Rule 5 (*Shortcut*). If there is a node u with $petal(u) = 1$, then we delete u and all incident edges from the graph, but we add all shortcuts bypassing u as new edges, i.e., for any path $v \to u \to w$ we add the edge (v, w).

Rule 6 (*Flower*). If there exists a node u with $|petal(u)| > k$, we add u to I, decrease k by 1, and delete u and adjacent edges from the graph.

Rule 5 is safe because any node u with $petal(u) = 1$ is dominated by another node (see above). Rule 6 is safe because not choosing u for the feedback vertex set would imply we must take one node in each of the at least $k + 1$ disjoint cycles through u. Note that Rule 4 is actually a special case of Rule 5. However, Rule 4 can be tested in constant time, while Rule 5 requires a min-cut computation. Therefore, we list them as two rules. Rule 6 is the directed version of the the undirected flower reduction proposed by Thomasse [15]. In contrast to the undirected case, this rule can be tested efficiently using standard min-cut techniques.

Note that we have reduction rules for nodes with small petal size and large petal size, but no rules for petal size between 2 and k. Such rules might be necessary to find a true kernelization for `DFVS`.

Initial Heuristics. It may be helpful to choose the initial subgraph H and its k-node feedback vertex set I more carefully. This was already suggested by Chen [4]. Assume a heuristic gives us a set F of nodes whose removal makes G acyclic. If $|F| \leq k$, we can choose $H = G$ and $I = F$ and we are done. Otherwise, we pick a random k-node subset F_0 of F and start Chen's algorithm with $H = G - (F - F_0)$ and $I = F_0$. Clearly, F_0 is a feedback vertex set of H, and if F is a good approximation to the optimal solution, H may be close to G. We propose three heuristics: Chen et al. had suggested to use [4], which unfortunately exhibited the worst performance in our experiments.

Heuristic 1 (*Big Degree*). We greedily add nodes of maximum undirected degree (i.e., *indegree* + *outdegree*) to F until the graph becomes acyclic.

Heuristic 2 (*Fractional Approximation*). We first compute a $(1+\epsilon)$-approximation for fractional `DFVS` [7], then greedily add nodes with heaviest fractional weight to F until the graph becomes acyclic.

Heuristic 3 (*Full Approximation*). We compute a `DFVS` approximation F with factor $O(\min\{\log\tau^* \log\log\tau^*, \log n \log\log n\})$ [7], where τ^* is the cost of a minimum fractional feedback vertex set.

4 Experiments

We used a PC with Intel(R) Xeon(TM) CPU (3.20GHz), 4 processors, 1 core per processor. The machine had 2 GB main memory (DDR2 SDRAM), 2 MB L2 cache memory, and an 80 GB serial ATA hard drive. We tested the algorithms on random graphs (see Section 4.1). For each set of parameters we used ten random graphs and recorded minimum, maximum, and average performance. For all runs of the algorithms we set a time-out threshold of three hours. We implemented the algorithms in `C++` using `LEDA-6.2` [12] on the `Fedora Core 8` operating system. We compiled the programs with `g++-4.1.2 -O3`. We have approximately `4,000 LOC` in total.

For Chen's algorithm, we found it non-trivial to map some of the basic graph operations to program code. For example, there is no way for two different graphs in `LEDA` to share common nodes, which is required when we want to compute induced graphs. We also found that some key operations common in FPT algorithms are not well supported by `LEDA`. For example, FPT algorithms often apply kernelization rules and then use brute-force search on the kernel to solve the problem. The brute-force search is usually done by iterating over all subsets of size k or each topological order of the graph. Therefore, we extended `LEDA` with two interfaces:

- `bool foreach_subset(list<node> &sub, graph &G, subset_prop_func pf, void *pars)`

- `bool foreach_topord(list<node> &sub, graph &G, topord_prop_func pf, void *pars)`

In the subset enumerator, `sub` specifies a subset of nodes in G. It comes together with a property function that tests whether the subset has a certain property. Each time we generate a new subset, we call the property function with the new subset and the user-supplied parameters `pars`. This function will stop and return `true` once the property function returns `true`. If all subsets have been tried, we simply return `false` to indicate that for any subset of `sub` the property cannot be satisfied.

In subsection 4.1, we first briefly discuss how we generated the random graph instances. Then we present our experimental data in four subsections. Complete experimental data and the source code of our programs are available online [8]. In

Section 4.2 we present data on the runtime performance and kernel size of Chen's original algorithm. In Section 4.3 we show the effectiveness of our new data reduction rules in improving the FPT search and reducing large input instances. In Section 4.4 we evaluate and compare three heuristics to obtain an initial configuration for Chen's algorithm. Finally, in Section 4.5 we evaluate the runtime performance of Chen's algorithm with respect to different parameters k for a graph with fixed optimum solution.

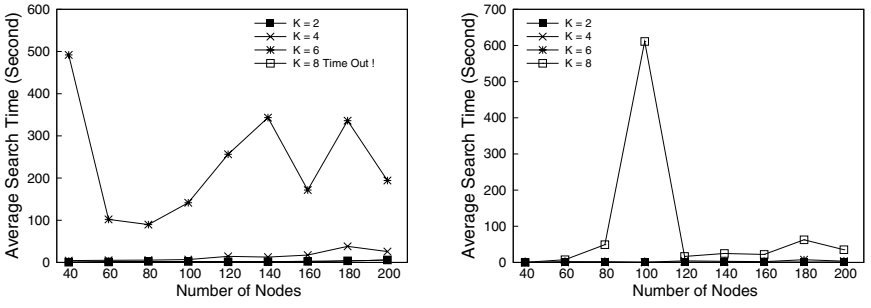
4.1 The Random Graph Generator

Our goal was to generate sufficiently difficult "random" instances while still precisely controlling three parameters: n, the number of nodes in the graph; k, the size of the optimum `DFVS` solution; and ed, the edge density ($m = ed \cdot n$). It seems difficult to generate truly random graphs with these parameters fixed, so we developed our own methods to generate graphs that seem to be quite random. We generate the graphs in two stages. First, we generate a random spanning tree [17] from which we then obtain a random connected `DAG` [13]. Finally, we add cycles to the graph, which poses two challenges: how to precisely control the minimum feedback vertex set size, and how to ensure that the newly generated graph is difficult to solve. We first randomly choose k nodes as the optimum solution, and then generate k node-independent cycles passing through each of them. We do not generate self-loops since they can easily be removed. Note that this method cannot generate k node-independent cycles for $k > \frac{n}{2}$. To create more overlapping cycles, we randomly fix a topological order for the nodes not in the solution. Each time we generate a cycle, we first randomly choose a subset A of the solution set and a subset B of the remaining nodes. We keep the nodes in B in their topological order when building cycles through these nodes. This ensures that the optimum solution size cannot exceed k. To ensure that our graphs are difficult to solve, we randomly add more cycles until we reach the required *edge density* ed, where the number of edges in a cycle is within $\frac{1}{4}$ of the total *edge bound*. Also, we try not to generate cycles that are too big by limiting the cycle size to at most $\frac{n}{4}$. Although we know one optimal solution of the generated graphs, they usually did not have a unique solution in our experiments.

4.2 Chen's Algorithm

Since FPT search scales poorly with n and k, we only considered graphs with $n = 40, 60, 80, \ldots, 200$, $k = 2, 4, 6, 8$, and edge density $ed = 2, 3, 3.5, 4$. We generated ten graphs for each triple (n, k, ed). Fig. 1(a) shows the average search time as a function of n and k for $ed = 2.0$. For all experiments, the memory usage was constant with approximately 10 MB.

We see from the data that the runtime of the algorithm scales poorly with k, which is typical for FPT algorithms. For $k = 8$ and $ed = 2.0$, the algorithm never finished within 3 hours. For $ed = 3.0$ and $k = 8$, the algorithm showed time-outs when n was larger than 140.



(a) No reductions. (b) With our new reductions.

Fig. 1. Runtime of Chen's algorithm, $ed = 2.0$. Note that the top graph in (a) corresponds to the case $k = 6$ (because $k = 8$ never finished within three hours), while the top graph in (b) is for the case $k = 8$.

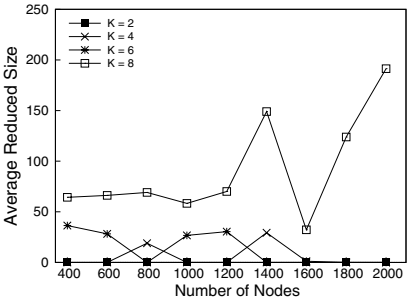
It seems that the runtime is not monotonely increasing in n, probably because it is dominated by the exponential dependency on k. The high runtime for low values of n (for example, $(40, 8, 2.0)$) is due to the fact that these graphs have many independent cycles, and Chen's algorithm seems to perform poorly on such graphs. A possible explanation may be found in the iterative compression technique. If we have many nodes on a few cycles and must add an additional node which is on a not yet blocked independent cycle, we must keep the new node and kick out one of the previously chosen nodes. Identifying such a node might be time-consuming.

Also, the runtime does not increase monotonely in the edge density. The algorithm can still quickly solve all instances with small parameter k when the number of edges increases significantly. Also, for bigger k, the runtime does not increase monotonely with the number of edges. For example, for $k = 8$, the problem becomes most difficult for the FPT algorithm when $ed = 2$. Most graphs time-out within three hours for this configuration.

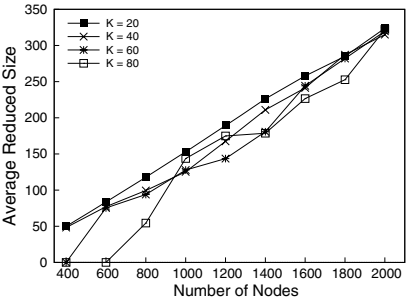
As we do not have kernelization rules for `DFVS`, we evaluated the kernel size by recording the maximum size of an instance of the standard min cut problem when solving the skew separator problem (in Chen's algorithm). However, we found this size is roughly the same as the size of the original graph (the deviation is usually two or three nodes).

4.3 Data Reduction Rules

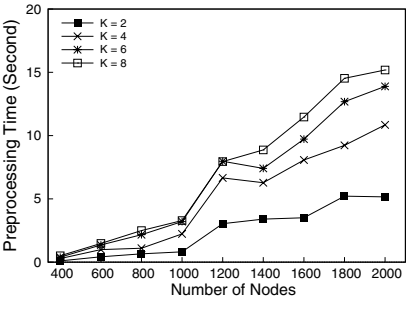
Fig. 1(b) shows the average runtime of Chen's algorithm after applying the new reduction rules for the same graphs as in Section 4.2. Now we can solve all the graphs quickly (the average longest time was 10 minutes, most often even less than 1 minute). However, the runtime is still not monotone with regard to n or ed; it is actually closely related to the reduced size after preprocessing. For example, there is a peak in the runtime when $ed = 2.0$ and $n = 100$. For this configuration, some instances got solved directly by preprocessing, while others needed hundreds



(a) Reduced size, small k.



(b) Reduced size, large k.



(c) Preprocessing time.

Rules	$k = 4$	$k = 40$
Chain	607	377
Dummy	785	679
Cut	996	1000
Chain+Dummy	522	377
Chain+Cut	366	377
Dummy+Cut	151	377
Dummy+Dummy+Cut	0	377

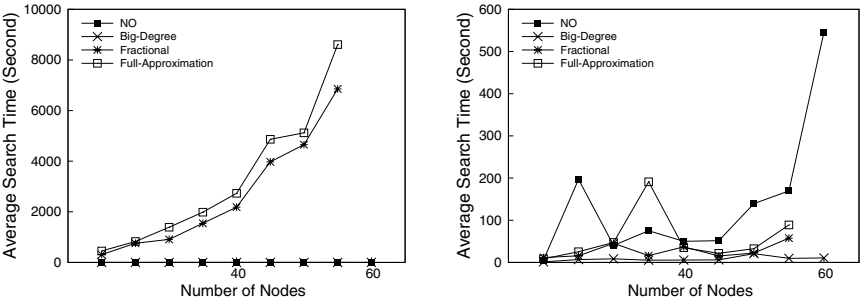
(d) Reduced size by combining different rules, $n = 1000$, $ed = 3.0$.

Fig. 2. Chen's algorithm with data reductions, $ed = 2.0$

of seconds to be solved. In particular, there is one instance with a big kernel (33 nodes) which was only solved after more than $2,000$ seconds.

We designed two more experiments to further test the power of data reductions. In the first experiment, we kept the small values for k and varied n between 400 and $2,000$, with step length 200. Fig. 2(a) and 2(c) show the average size reductions and preprocessing times, respectively. Surprisingly, we found that the reduction rules could directly handle most of the input instances when $ed > 2$. There are two possible explanations: The parameter k may be too small, or our "random" graphs actually have many overlapping cycles. In both cases, the *Flower* reduction could reduce the input size considerably. The preprocessing time grows linearly in n and ed. On the average, it was just 3 minutes for all graphs together.

In the second experiment, we varied k in $\{20, 40, 60, 80\}$ for the same n as in the first experiment. Fig. 2(b) shows the average reduced size for these configurations. We observe that for fixed edge density the reduced size scales linearly with the number of nodes, which indicates that our reduction rules work consistently for different graph sizes. Also, the reduced size increases with increasing edge density. This is quite different from the first experiment where the preprocessing could directly solve almost all graphs when ed is large. The reason is that the Flower reduction rule does not work well in these cases if k gets larger. Thus, it would be



(a) Heuristics. (b) Chen's algorithm after heuristic.

Fig. 3. Runtimes of heuristics for Chen's algorithm, $k = 8$ and $ed = 3.0$

important to devise reduction rules that work well for relatively large parameter k as well as large edge density.

Individual reduction rules. We tried to find out which rules are more important than others. We studied two groups of graphs, the first group had parameters $n = 1000$, $k = 4$, $ed = 3.0$, and the second group had $n = 1000$, $k = 40$, $ed = 3.0$. We evaluated three types of rules: *Chain*, *Dummy*, and *Cut*. Type *Chain* rules consisted of the *Chaining Nodes* and *Shortcut* rules; Type *Dummy* rules contained the *Dummy Nodes* rule and the reduction that removes all nodes with *petal* size 0; Type *Cut* rules were just the *Flower* rule. Table 2(d) summarizes the average reduced sizes for these two configurations. We see that the *Cut* rules do not reduce the graph significantly, but they are useful in determining the nodes in the minimum feedback vertex cover. For example, the *Cut* rules may reduce a graph from 1000 to 996 nodes when $k = 4$, but these 4 nodes are in the feedback vertex set, so we have already solved the problem even though there are still 996 unprocessed nodes. The *Flower* rules might help other rules to further reduce the graph. Our data show that the graphs are reduced significantly when we combine *Chain* or *Dummy* rules with *Flower* rules. This is because a *Flower* rule actually selects a node for the feedback vertex set and then deletes it from the graph, which may trigger other rules. The *Flower* rule becomes useless when k grows, because graphs usually do not have large flowers.

4.4 Heuristics for the Initial Configuration

We evaluated the three heuristics mentioned in Section 4 to find a good initial configuration for Chen's algorithm. Since the FPT algorithm does not scale well with n and k and the approximation algorithms are very slow even on small graphs, we only considered graphs with $n = 20, 25, 30, \ldots, 60$ nodes, with $k \in \{2, 4, 6, 8\}$ and $ed = 3.0$.

Fig. 3 shows the runtimes for $k = 8$ (including the times for no heuristic). The graphs would be similar for other values of k. Note that we distinguish between runtime of the heuristic and runtime of Chen's algorithm afterwards. The heuristics

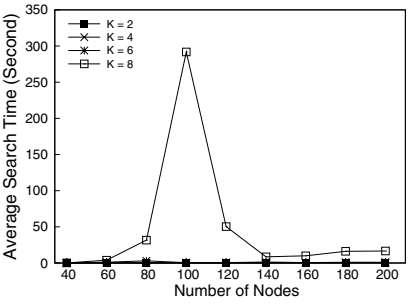


Fig. 4. FPT search time with preprocessing and *Big-Degree* heuristic

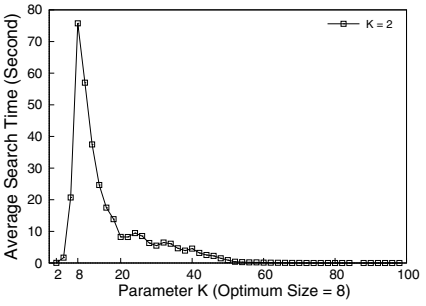


Fig. 5. The FPT search time with respect to different parameter k

seem to become very useful for larger parameters k. On all our graphs, the *Big-Degree* heuristic achieved the most significant acceleration (4x to 50x speedup). Moreover, the *Big-Degree* heuristic (which usually takes only a few seconds) runs thousands of times faster than *Fractional Approximation* and *Full-Approximation* (which usually need thousands of seconds). Overall, we conclude that the best approach for solving DFVS is to first apply our reduction rules, then use *Big-Degree* to compute the initial configuration, and then use Chen's algorithm [4] to solve the problem. For example, Fig. 4 shows that, with the *Big-Degree* heuristic, we can achieve another speed-up of $2 - -3$x compared to reduction rules without heuristic (Fig. 1(b)).

4.5 The Impact of the Parameter k

Intuitively, we may quickly find a feedback vertex set of size k if k is much larger than the optimal one. We may also quickly reject the input if k is much smaller than the optimal parameter. To test this hypothesis, we generated graphs with 100 nodes for $k = 8$ and $ed = 3.0$. Then we used Chen's algorithm to search a feedback vertex set of size $k = 2, 4, 6, 8, 10, \ldots, 98$. Fig 5 summarizes the average search times for this experiment. As expected, the runtime reaches its peak when the parameter k is near the size of the minimum feedback vertex set. The runtime is quite fast when k is much larger or much smaller than the optimum.

5 Deadlock Detection

Deadlock recovery in concurrent programs has always been considered an important application of DFVS [4,7]. Indeed, deadlock recovery is a very important topic in our modern *multi-core/many-core* computing era. Solving the concurrency problem has recently seen tremendous research interest in operating systems, programming languages and computer architecture communities.

For simplicity of analysis we mainly focus on `mutex` locks in the POSIX Thread library (`Pthread`). We will argue that, with the restrictions of the programming model, `DFVS` is not more helpful than cycle detection for deadlock recovery. This proposition is further confirmed by a report on a deadlock immunity system [10] in a recent systems conference.

In a concurrent program, the nodes of a Resource Allocation Graphs (`RAG`) are resources and threads. Resources are simply mutex locks. There are three types of edges in a `RAG`: *request*, *grant* and *own*. The *request* and *grant* edges are edges from thread to lock, while the *own* edges go from lock to thread. *request* edge means a thread is requesting a lock, and *grant* edge means the thread library allows the thread to wait on the lock (note that the thread may yield its execution to another one before being allowed to wait on the resource). The *own* edge from lock to thread means the thread currently owns this resource exclusively. A deadlock appears as a cycle in the `RAG`. In such a cycle, all edges are exclusively the *grant* and *own* edges. One lock can be owned by *only one* thread, even for *recursive* locks, though certain threads could acquire the lock multiple times. One thread, at one time, can be granted to wait on *only one* lock because threads are executed sequentially, and we cannot wait on two resources simultaneously. Thus, we cannot have *overlapping* cycles in a `RAG`, and therefore simple cycle detection suffices to resolve the deadlocks.

6 Conclusions

We presented a comprehensive experimental study on the feedback vertex set problem in digraphs. We proposed new data reduction rules to efficiently reduce the FPT search space. Finally, we demonstrated that `DFVS` search is not more helpful than cycle detection in efficient deadlock detection in modern concurrent systems.

We would like to find better data reduction rules for `DFVS`, in particular a polynomial-size kernel. We remark that though `UFVS` has a small problem kernel, the parameter k is potentially very large for dense graphs, as in undirected graphs it is quite easy to have a cycle. We may also study other parameters than "solution size", for example, "edge density". Our reduction rules seem to perform well when the edge density is low. Better approximation algorithms for `DFVS` might also help to speed up the FPT search. We demonstrated that `DFVS` is not more helpful than cycle detection in detecting deadlocks with mutex locks, but there are also other types of locks, such as read/write locks, that may generate complex overlapping deadlocks. However, we found that other lock types, for example spin locks, behave essentially the same as mutex locks. Read/write locks may behave differently. but most thread libraries (e.g. `NPTL` (Native POSIX Thread Library) restrict the number of threads in read mode, and once a thread is waiting in write mode, later read requests are pending until completion of the write. We are now investigating the practicability of FPT `DFVS` algorithms in circuit testing to reduce the hardware overhead required for "scan registers" [11].

References

1. Bodlaender, H.L.: On linear time minor tests with depth first search. Journal of Algorithms 14, 1–23 (1993)
2. Bodlaender, H.L., Penninkx, E.: A linear kernel for planar feedback vertex set. In: Grohe, M., Niedermeier, R. (eds.) IWPEC 2008. LNCS, vol. 5018, pp. 160–171. Springer, Heidelberg (2008)
3. Chatrand, G., Lesniak, L.: Graphs & Digraphs, 2nd edn. The Wadsworth and Brooks/Cole Mathematics Series (1986)
4. Chen, J., Liu, Y., Lu, S., Sullivan, B., Razgon, I.: A fixed-parameter algorithm for the directed feedback vertex set problem. Journal of the ACM 55(5), 1–19 (2008)
5. Downey, R.G., Fellows, M.R.: Fixed-parameter tractability and completeness I: basic results. SIAM Journal on Computing 24, 873–921 (1995)
6. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1999)
7. Even, G., Naor, J., Schieber, B.: Approximating minimum feedback sets and multicuts in directed graphs. Algorithmica 20, 151–174 (1998)
8. Fleischer, R., Xi, W., Yuan, L.: DFVS Project (2009), `http://www.tcs.fudan.edu.cn/rudolf/Projects/DFVS/dfvs.html`
9. Huffner, F., Niedermeier, R., Wernicke, S.: Techniques for practical fixed-parameter algorithms. The Computer Journal 1(51), 7–25 (2008)
10. Jula, H., Tralamazza, D.M., Zamfir, C., Candea, G.: Deadlock immunity: Enabling systems to defend against deadlocks. In: Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation (OSDI 2008), pp. 295–308 (2008)
11. Kunzamann, A., Wunderlich, H.J.: An analytical approach to the partial scan problem. Journal of Electronic Tesing: Theory and Applications 1(5), 163–1741 (1990)
12. LEDA: A library of the data types and algorithms of combinatorial computing, `http://www.mpi-inf.mpg.de/LEDA/`
13. Melancon, G., Dutour, I., Bousquet-Melou, M.: Random generation of directed acyclic graphs. Technical report, CWI Amsterdam (2006), `http://www.cwi.nl/InfoVisu`
14. Seidl, H.: Personal communication (2000)
15. Thomasse, S.: A quadratic kernel for feedback vertex set. In: Proceedings of the 20th ACM-SIAM Symposium on Discrete Algorithms (SODA 2009), pp. 115–119 (2009)
16. Wang, C.-C., Lloyd, E.L., Soffa, M.L.: Feedback vertex sets and cyclically reducible graphs. Journal of the ACM 32(2), 2960–2913 (1985)
17. Wilson, D.B.: Generating random spanning trees more quickly than the cover time. In: Proceedings of the 28th ACM Symposium on the Theory of Computation (STOC 1996), pp. 296–303 (1996)