A Network-Flow-Based Scheduler: Design, Performance History and Experimental Analysis

Harold N. Gabow
University of Colorado at Boulder
and
Tadayoshi Kohno
Reliable Software Technologies

Association for Computing Machinery, Inc., 1515 Broadway, New York, NY 10036, USA
Tel: (212) 555-1212; Fax: (212) 555-2000

We describe a program that schedules physician attending teams at Denver Health hospital. The program uses network flow techniques to prune an exponentially sized search space. We describe the program design, its performance history at the hospital, and experiments on a simplified version of the program.

General Terms: Experimental Algorithms

Additional Key Words and Phrases: scheduling, graphs, matroids

1. INTRODUCTION

In 1986 one of us (H.G.) wrote a program to schedule the doctors for attending rounds at Denver Health hospital. This program has been used every year since then (excepting 1993–1995) to quickly produce yearly schedules of high quality. The program uses network flow techniques in a backtracking scheme to explore a large space of candidate solutions efficiently. This paper describes the design of the program and the graph theoretic ideas involved, our computational experience in

Name: Harold N. Gabow
Address: Department of Computer Science, University of Colorado at Boulder, Boulder, CO 80309. e-mail: `hal@cs.colorado.edu`
Name: Tadayoshi Kohno
Address: Reliable Software Technologies, 21351 Ridgetop Circle, Dulles, VA 20166. e-mail: `kohno@rstcorp.com`

generating real-world schedules, and the experiments we conducted to gain greater insight into the workings of the program.

Section 2 discusses the doctor-scheduling problem, our graph model and algorithm, and performance of the algorithm in generating real-world schedules. Section 3 presents a simplified version of the problem and Section 4 describes experiments on this version. Section 5 summarizes our conclusions. This section closes with some notation.

Let $\mathbf{Z}_+$ be the set of nonnegative integers. We typically denote a singleton set without using braces, e.g., $I \cup e$ is the same as $I \cup \{e\}$.

We usually use n and m to denote the number of vertices and edges respectively of the graph under consideration. In an undirected graph G let $d(v)$ denote the degree of a vertex v; if the graph is not clear from context we write $d(v, G)$. We often identify a graph and its edge set, e.g., we write $d(v, A)$ for a set of edges A. We denote a bipartite graph as $G = (X \cup Y, E)$, meaning that the vertex set is partitioned into sets X and Y and each edge joins vertices of these two sets.

2. PROBLEM FORMULATIONS AND ALGORITHMS

2.1 The real-world problem

The problem is to generate a yearly schedule for Department of Medicine attending teams at the Denver Health hospital. The details that we describe below are typical of general medicine attending teams in hospitals.

The Department of Medicine consists of roughly 30 doctors.[1] It is divided into about a dozen divisions (e.g., Cardiology, Pulmonary), the largest being about 5 doctors. About half a dozen doctors do not belong to any division. (All these numbers vary from year to year, e.g., the number of doctors has ranged from 31 to 41.)

Each month a team of 4 doctors from the department attends. (In 1996 the teams had 5 doctors.) Attending entails making rounds and managing the care of all the patients on the medicine service. Attending is a large commitment of time and energy, yet all other subspecialty duties must still be fulfilled.

Attending teams are drawn from all the doctors in the department. A given division can spare only one doctor to attend during any given month. A given doctor is required to attend a certain number of months over the year, the number varying from doctor to doctor. These required man-months sum to exactly 48, the number of man-months for a 4 man team attending each month of the year. Each doctor supplies a list of months during which he is available or prefers to attend, for the entire year. It is desirable that doctors who attend more than one month get as many months off between consecutive attendings as possible, in order to minimize the disruption in their schedule.

In addition idiosyncratic constraints often occur. There have been doctors who prefer to attend in consecutive months (not many). A doctor may need to attend in some given month, but his other months can be selected from his preference list. Two doctors may be required to attend in the same months ("co-attendings") or required to attend in different months (husband and wife).

[1] We shall speak of doctors using masculine terminology but this is for convenience only.

The most important measure of quality of a schedule is the time off between consecutive attending months. To make this notion precise we define the *clearance* of a schedule to be the minimum number of months off between two consecutive attendings of any doctor. For instance a schedule has clearance at least 1 exactly when no doctor attends in two consecutive months. We also define the *division constraint* as the requirement that no division supplies more than 1 attending doctor in any month. The *preference list* of a doctor is the list of months he supplies as possible attendings.

2.2 Nonconflicting f-factor problem

Consider an undirected graph $G = (V, E)$. For any function $f : V \to \mathbf{Z}_+$ an *f-factor* is a spanning subgraph F such that each vertex $v \in V$ has $d(v, F) = f(v)$ [6]. For given G and f let $\mathcal{F}$ be the family of all f-factors. Let $\mathcal{C}$ be a family of inclusionwise minimal subsets of E, where we call each member of $\mathcal{C}$ a *conflict*. A *nonconflicting f-factor* is a subgraph $F \in \mathcal{F}$ containing no conflicts. The *nonconflicting f-factor problem* is to find such a subgraph, given G, f and $\mathcal{C}$.

We cast our scheduling problem as a nonconflicting f-factor problem as follows. Let D be the set of all doctors and M the set of all months $\{1, \ldots, 12\}$. Define a bipartite graph

$$G = (D \cup M, E)$$

where $(d, m) \in E$ for $d \in D$, $m \in M$ exactly when doctor d is available to attend in month m. The function f specifies the number of months each doctor must attend as well as the size of the attending team in each month (the latter being 4 in our problem). Fix an integer c as the desired clearance of a schedule. The family $\mathcal{C}$ is defined in a straightforward way to take all remaining scheduling criteria into account. Thus the division constraint corresponds to pairs $\{e, e'\} \in \mathcal{C}$ where edges e and e' join the same division to the same month. The clearance constraint corresponds to pairs $\{e, e'\} \in \mathcal{C}$ where edges e and e' join the same doctor to months that differ by at most c. The remaining constraints of $\mathcal{C}$ model the idiosyncratic constraints, if any.

The desired schedule is any nonconflicting f-factor that has the largest possible clearance, c^*. We find c^* and the corresponding schedule by solving the nonconflicting f-factor problem for consecutive values of $c = 0, 1, \ldots$, stopping when we find the first infeasible value $c^* + 1$.

2.3 Basic algorithm

The *basic algorithm* solves the nonconflicting f-factor problem as follows. For any two sets of edges $I, O \subseteq E$ define

$$\mathcal{F}[I, O] = \{F \in \mathcal{F} : I \subseteq F \subseteq E - O\}.$$

(The edges of I and O must be "in" the f-factor and "out of" the f-factor respectively.) Start with any $F \in \mathcal{F}$. If there is no conflict then we are done so assume that F has a conflict. Index the edges of F as $e_i, i = 1, \ldots, |F|$ so that F_k is a conflict, where in general we write $F_i = \{e_1, \ldots, e_i\}$. Observe that any nonconflicting f-factor does not contain F_k and so belongs to exactly one of the disjoint families

$$\mathcal{F}_i = \mathcal{F}[F_{i-1},\ e_i],\ i = 1, \ldots, k. \tag{1}$$

Note that F_0 is empty. Also if convenient we can choose F_k to contain a conflict rather than to actually be a conflict. In fact this is done in all our implementations, as discussed in Section 2.5.

We process every such family $\mathcal{F}_i$ in the following way. Define $g : V \to \mathbf{Z}_+$ by $g(v) = f(v) - d(v, F_{i-1})$. Let F' be a g-factor in the graph $G - F_i$. (If no such g-factor exists then we are done processing $\mathcal{F}_i$.) Recursively process $\mathcal{F}_i$ using F' as the starting f-factor. Note that the recursion shrinks the graph for the purpose of finding factors (i.e., the graph changes from G to $G - F_i$). However in general we must test for conflicts $\mathcal{C}$ using the f-factor in the original graph (for graph $G - F_i$ the factor F' corresponds to $F' \cup F_{i-1}$ in G). It is clear that the recursive procedure we have sketched discovers a nonconflicting f-factor as long as one exists.

We find F' efficiently as follows. The set $F - F_i$ is just one edge shy of being the desired g-factor, i.e., if $e_i = (x, y)$ then $d(v, F - F_i) = g(v)$ for every vertex $v \neq x, y$ and $d(v, F - F_i) = g(v) - 1$ for $v = x, y$. Search for an augmenting path for $F - F_i$. If such a path exists we get the desired F'; in the opposite case no F' exists.

When G is bipartite as in our scheduling problem f-factors correspond to network flows. We can find an augmenting path using the Ford-Fulkerson labelling method [2]. For general graphs we use Edmonds' algorithm [5]. In either case F' is found in $O(m)$ time. (We can "skip over" $\mathcal{F}_i$'s that have no factor F' using a technique of [3]. However this modification does not seem worth programming.)

2.4 Discussion

Our problem involves a huge unstructured search space of possible schedules. Looking only at f-factors prunes away a large portion of this space. The partitioning scheme (1) avoids generating f-factors that have a known conflict (F_k). By using augmenting paths it is easy to reject large subspaces as being infeasible, or quickly start a search over some subspace.

This intuition suggests that a primary goal is to reduce the total number of subproblems. This in turn suggests the heuristic of choosing a conflict F_k for (1) that makes k small. On the other hand it is not difficult to imagine problems where this greedy heuristic gets "distracted" by small conflict sets when the essential conflict to work on is a large set. Our work on reducing the number of subproblems is discussed in Section 4. For that discussion note that the number of subproblems is precisely the number of recursive calls made by the algorithm.

When G is bipartite we can handle a natural extension of the problem. If the bipartition has sets D, M then we may wish to have $d(v, F) = f(v)$ for $v \in M$ and $d(v, F) \leq f(v)$ for $v \in D$ (i.e., excess manpower is available). The above algorithm handles this case, assuming that for the above edge $e_i = (x, y)$ with $x \in D, y \in M$ we search for an augmenting path starting from vertex y.

2.5 Implementation

Our first program (which is the one still used at Denver Health) implements the basic algorithm. Any implementation of the basic algorithm requires a number of important problem-specific decisions. We now give a high-level summary of these aspects of our implementation.

We store the graph $G = (V, E)$ as a collection of $|D|$ adjacency lists for the doctors. This facilitates checking clearance constraints. We number the vertices so the doctors are $1, \ldots, |D|$, and doctors in a given division are consecutive. This facilitates checking division constraints. Each month has a list of the doctors currently matched to it. This is used in searching for an augmenting path (as will be seen in Fig. 1). All lists are sequentially allocated since we do not modify the graph.

Only the clearance and division constraints are explicitly checked by the program. Idiosyncratic constraints vary from year to year. We have always been able to handle these constraints by simple modifications to the input graph, most notably by node splitting. For example a doctor d who must attend two months, one of which must be month m, can be represented by vertices d_1, d_2 with $f(d_1) = f(d_2) = 1$, d_1 adjacent to only vertex m and d_2 adjacent to the months in d's preference list that differ from m by more than c.

For the rest of the discussion define *adjacency list order* to be the natural total ordering of E induced by the adjacency list representation of G. To define this precisely recall that the doctors are numbered from 1 to $|D|$. Adjacency list order is lexicographic order on E, using the key (d, i) for the ith edge in the adjacency list of doctor number d.

We generate the conflict partitions $\mathcal{F}_i$ using (1) as follows. First the edges of the current f-factor F are moved to the beginning of their adjacency lists. This gives a new adjacency list order. Now the edges of $e \in F$ are scanned in adjacency list order to find the first such e that forms a conflict with previous edges. (e is checked with the previous edges of F to see if there is a clearance or division conflict; checks for idiosyncratic conflicts could be added here.) Suppose such a conflict is found. Index the edges of F up to e as $e_1, e_2, \ldots, e_k = e$. Use the partition (1) defined by this sequence of k edges.

Observe that the graph corresponding to any $\mathcal{F}_i$ of (1) consists of all the edges that follow e_i in adjacency list order. (This uses the fact that the edges of F are at the beginning of their adjacency lists.) This continues to hold after any number of recursive calls: Any edge preceding e_i in adjacency list order belongs to $I \cup O$ for the current partition $\mathcal{F}_i$. Hence no work is needed to construct the graph for recursive calls. (More details on this issue are in Fig. 1 below.) This is the prime advantage of our partitioning strategy.

A potential disadvantage of our partitioning strategy is that the partition of (1) may contain more sets than necessary. To illustrate suppose in the above discussion edge e completes a clearance or division conflict. This conflict must be $\{e, e'\}$ for some edge $e' \in F$ preceding e. It is possible to define a partition into two families, $\mathcal{F}[\emptyset,\ e]$ and $\mathcal{F}[e,\ e']$. Our strategy generates $k \geq 2$ families, where possibly k is large. As already mentioned generating a large number of families is inefficient from a greedy point of view. However it is unclear what the global consequences of the strategy are, since a priori any of our k families can contain the desired solution. We return to this issue in Section 4.

We use a "hardest-first" heuristic to number the doctors. Intuitively doctors or groups of doctors that are highly constrained should have low numbers. As a simple illustration consider a doctor d who has only one feasible schedule for the current clearance value c. If d has a high number his edges will generate unresolvable conflicts until, by luck or by exhaustive search, the attending months that he needs have not been completely assigned to lower-numbered doctors. (For instance if doctor d must match edge (d, m) and the preceding doctors $1, \ldots, d-1$ have 4 matched edges incident to m, then no partition family $\mathcal{F}_i$ with these 4 edges in I contains a nonconflicting f-factor.) Thus d should have a low number. The same holds for small sets of doctors that collectively generate similar conflicts. Following this intuition we number large divisions first, as well as divisions with highly constrained doctors. Within a division the most constrained doctors are numbered first. This ordering has been done by hand in a quick pass over the given input file.

The search for an augmenting path is written as a recursive depth-first search, for programming simplicity. See Fig. 1. If the current graph consists of the edges following $e = (d, m)$ in adjacency list order then any doctor $d' < d$ has marked_doc$[d']$ = **false**. This forces the search to use only edges in the current graph.

```
boolean procedure DFS(d) {
  marked_doc[d] = true;
  for each unmatched edge (d, m) ∋ not marked_mth[m] do {
    /* advance on unmatched edge */
    if month m is deficient then {
      /* an augmenting path has been found */
      /* code to augment 2 edges goes here */
      return true }
    else {
      marked_mth[m] = true;
      for matched edges (m, d') ∋ not marked_doc[d'] do {
        if DFS(d') then {
          /* code to augment 2 edges goes here */
          return true } } } };
  return false }
```

Fig. 1. Low-level pseudocode for the depth-first search to find an augmenting path. The marked_doc array marks doctors that should not be visited because they have been reached in the search or they are not in the current graph. The marked_mth array marks months that should not be visited because they have been reached in the search.

Although our design was motivated to make creating subproblems easy, there is still some extra bookkeeping needed. First we must ensure that an unsuccessful recursive call still returns with a valid flow (though not necessarily the same flow as it started with). Second, as already mentioned we must check for conflicts outside of the recursive structure: Suppose the current graph consists of the edges following $e = (d, m)$ in adjacency list order. Suppose further that we wish to check an edge e' for conflicts, where e' is incident to the same division D as e. To find a division conflict we must examine all the edges of the fixed set F_{i-1} of (1) that are incident to D (these of course are not in the recursive graph). Some straightforward extra bookkeeping keeps track of those edges.

2.6 Performance history

The program is about 400 lines of commented C code (orginally PASCAL) including I/O and schedule-checking routines. Over the years it has been run on Intel 286, 386, 486 and Pentium machines in DOS and Linux. For the problems from real-world data the program always finds a maximum clearance schedule in seconds or tens of seconds. Since this seemed too good to be true we have, on several occasions, manipulated the real-world input to make it more difficult. Without increasing the size of the input it is usually possible to add constraints to get a problem that can require on the order of hours of computing time. One such data set ran for a week without arriving at a solution. So it is clear that even problems of the moderate size encountered at Denver Health can be difficult.

This experience reinforces our intuition that the key ingredient is successful pruning of the search space. The basic algorithm's restriction of the search space to f-factors is sufficient to solve the problems encountered in this real-world application. The program is fast enough that it is easy to run many times to make incremental improvements. This was done in some years when the Chief of Medicine so desired. The schedules have always been well-received by all the doctors.

2.7 Background from matroids

We also developed a more powerful version of the algorithm which we now discuss. We begin with a brief summary of background from matroids. See [5] for a detailed treatment.

A matroid consists of a ground set V and a family $\mathcal{I}$ of "independent" subsets of V. The independent subsets satisfy axioms that are motivated by two commonly occurring matroids: the family of forests of a graph and the family of independent sets of vectors contained in some arbitrary finite subset of a vector space. In a *partition matroid* the ground set has a partition $\mathcal{F}$, along with a function $f : \mathcal{F} \to \mathbf{Z}_+$. A set A is independent, i.e., $A \in \mathcal{I}$, if

$$|A \cap S| \le f(S) \text{ for each } S \in \mathcal{F}. \tag{2}$$

We also use a related matroid known from folklore that we call a *laminar matroid*. Recall that a *laminar family* on V is a family of subsets of V, any two of which are either disjoint or one contains the other. In a laminar matroid the ground set is given along with a laminar family $\mathcal{F}$ on it. There is again a function $f : \mathcal{F} \to \mathbf{Z}_+$ and again a set A is independent if (2) holds. It is a simple exercise to verify this system is a matroid. In fact we can also allow a lower bound function $\ell : \mathcal{F} \to \mathbf{Z}_+$ and replace (2) by the requirement that every maximal independent set ("base") of the matroid B satisfies

$$\ell(S) \le |B \cap S| \le f(S) \text{ for each } S \in \mathcal{F}.$$

Let I be an independent set of a matroid. For an element e with $I \cup e \notin \mathcal{I}$, the *fundamental circuit* of e in I, $C(e, I)$, is $\{f \in I : I \cup e - f \in \mathcal{I}\}$. For a laminar matroid $C(e, I) = I \cap S$ for the minimal set $S \in \mathcal{F}$ that contains e and has $|I \cap S| = f(S)$. (If there are lower bounds, assuming that I is a base $C(e, I) = I \cap S - \cup\{T \in \mathcal{F} : e \notin T,\ |I \cap T| = \ell(T)\}$ for S determined as above.)

Consider matroids $\mathbf{M}_i$, $i = 1, 2$ on the same ground set. A *(matroid) intersection* is a set that is independent in both matroids. The *matroid intersection problem* is to find a maximum cardinality intersection of two given matroids $\mathbf{M}_i$. The bipartite matching problem is a special case, corresponding to using a partition matroid for each $\mathbf{M}_i$. Specifically for a bipartite graph $G = (X \cup Y, E)$, the ground set of both matroids is E. A set $A \subseteq E$ is independent in $\mathbf{M}_1$ if for each $x \in X$, $|\{(x,y) \in A : y \in Y\}| \leq 1$. $\mathbf{M}_1$ is a partition matroid. Similarly the set Y gives a partition matroid $\mathbf{M}_2$.

The matroid intersection algorithm, due to Edmonds [5], works by finding augmenting paths. There are two main differences between these augmenting paths and the more well-known augmenting paths from network flow: First suppose that e, e' are two consecutive elements in the path. In network flow e and e' are edges with positive capacity forming a length two path. In matroid intersection e and e' are ground set elements, exactly one of which belongs to the current intersection I. Furthermore, letting $C_i(e, I)$ denote the fundamental circuit in matroid $\mathbf{M}_i$, $e \notin I$ implies $e' \in C_2(e, I)$ and $e \in I$ implies $e \in C_1(e', I)$.[2] The second difference is that for matroid intersection an augmenting path cannot have a "shortcut" (defined in the natural way).

2.8 Matroid algorithm

The *nonconflicting matroid intersection problem* is defined in the natural way: Given matroids $\mathbf{M}_i$, $i = 1, 2$ on the same ground set V, plus a family of conflicts $\mathcal{C}$ on V, we seek a maximum cardinality matroid intersection that contains no conflict. The basic algorithm presented above extends to this problem in a straightforward way. (To process an $\mathcal{F}_i$, start with the intersection $F - e_i$ and find an augmenting path that does not contain an element of F_i.) We use the term *matroid algorithm* to refer to any such implementation of the basic algorithm. Clearly the crucial ingredient of a matroid algorithm is the choice of matroids. We now illustrate how a matroid algorithm can capture more constraints than the basic algorithm.

Recall the division constraint says that only one member of a given division can be scheduled in any given month. This is a common type of constraint. For instance the Anesthesiology Department at the University of Colorado Medical School generates a monthly schedule for its doctors. Each day each anesthesiology service must be covered by a doctor. By definition only one service on a given day can be covered by any given doctor. This gives a constraint isomorphic to the division constraint, if we view all the services on a given day as forming a division.

One can model division constraints in the graph itself, rather than in the constraint set $\mathcal{C}$. Instead of using f-factors one uses network flows. The flow network is designed to model the division constraints by using 1 extra node for each division in each month. We did not do this because the flow network is much larger than the original graph; also the network is not bipartite so the code becomes more complicated.

Our approach is to use the matroid algorithm. Recall the bipartite graph $G = (D \cup M, E)$ from the basic doctor scheduling problem. We define two matroids, both having ground set E. Let $A \subseteq E$. A is independent in the partition matroid $\mathbf{M}_1$ if for each doctor d, $|\{(d,m) \in A : m \in M\}| \leq f(d)$. Define a laminar matroid $\mathbf{M}_2$ as follows. For each month $m = 1, \ldots, 12$ the laminar family has a set S_m containing every edge incident to m. Furthermore S_m is partitioned into sets $S_{m,j}$, where j ranges over all the divisions j and $S_{m,j}$ contains every edge joining month

[2]For definiteness we have assumed the augmenting path starts with an element e with $I \cup e$ independent in $\mathbf{M}_1$.

m to a doctor in division j. The matroid's bounding function f has $f(S_m) = 4$, $f(S_{m,j}) = 1$. The matroid intersections correspond (1-to-1) to the assignments that satisfy the division constraint and do not overcommit any doctor. We use the matroid algorithm to find a maximum cardinality nonconflicting intersection.

The main difference between this matroid algorithm and the basic algorithm is in how an augmenting path is found. Only one extra test is required, as we now show. (See Fig. 2.) Recall that in Edmonds' matroid intersection algorithm, consecutive elements e, e' of an augmenting path for the current matroid intersection I satisfy $e' \in C_2(e, I)$ or $e \in C_1(e', I)$. The condition $e \in C_1(e', I)$ means that e' is incident to the same doctor as e (as in an ordinary augmenting path). To interpret the condition $e' \in C_2(e, I)$ let $e = (d, m)$. If I contains an edge $g = (d', m)$ where d' is in the same division as d, then we must have $e' = g$. In the opposite case e' is essentially chosen as usual: If I contains fewer than 4 edges incident to m then an augmenting path has been found. If I contains 4 edges incident to m then e' can be any of these edges.

```
if some matched edge (m, d') has d' in d's division then {
  if not marked_doc[d'] and DFS(d') then {
    /* code to augment 2 edges goes here */
    return true }
  else return false };
```

Fig. 2. To change the basic algorithm into the matroid algorithm insert this code as the first statement of the **for** loop in Fig. 1 (i.e., the code goes between the "advance" comment and the **if** statement). The marked_mth array ensures that the augmenting path is shortcut-free but still allows the path to visit a month up to 4 times.

The advantage of this matroid algorithm is that it does more pruning automatically than the basic algorithm, since it only considers subgraphs that obey the division constraint. The disadvantage is that its augmenting paths can be longer: Conceivably the new test can force an augmenting path to visit the same month node 4 times.

We conclude with two more remarks comparing the matroid algorithm with the alternative network flow approach mentioned above. The matroid algorithm has the advantage that it sometimes finds a shorter augmenting path. (It never finds a longer augmenting path.) This is caused by the requirement that an augmenting path for matroid intersection must be shortcut-free.

Secondly we observe that one can use the network flow approach to derive an algorithm equivalent to the matroid algorithm: Instead of explicitly constructing the flow graph one uses the original graph and "hard-codes" the new nodes and edges into the routine that searches for an augmenting path. However cleaner theory makes the matroid algorithm easier to code and debug. The matroid algorithm also adapts easily to other matroids, as illustrated below.

2.9 Performance history

We ran this matroid algorithm and the basic algorithm on difficult instances constructed by modifying real-world input to involve large divisions and have no feasible schedule. In typical instances the basic algorithm ran over 20 minutes and the matroid algorithm was under 4 minutes on a 486 33MHz machine (17M recursive calls versus 3.3M recursive calls). This is in keeping with our intuition on the superiority of the matroid algorithm in pruning the search space. On smaller problems the matroid algorithm was sometimes slightly slower, most likely because of longer augmenting paths.

Another illustration of the importance of pruning was provided by real-world data in 1997, when the scheduling rules were changed. (The doctors did not find the new system superior, so the original system was reinstituted in 1998.) In the 1997 system each month's team consisted of 3 "normal" doctors and 1 "backup" doctor, the latter to be called in emergencies. Some doctors volunteered for backup duty for 1 month; some divisions volunteered to collectively supply a backup for some number of months. All of the ordinary constraints continued to hold, regardless of whether a doctor was scheduled as normal or backup.

To get a quick solution we treated backup doctors as contributing an idiosyncratic constraint to $\mathcal{C}$. We added a small amount of code to check that constraint (i.e., each month required 3 normal doctors and 1 backup). The resulting program ran on three different machines (with slightly different input) each for over 2 days. No answer (i.e, a schedule or proof of infeasibility) was found and the runs were aborted.

Our second attempt was based on modifying the solution space. We treated each month as a partition matroid, with one block for the 3 normal doctors and another block for the backup. Each doctor volunteering for backup duty was also treated as a partition matroid, with one block for normal duty and another for backup. Hence every matroid intersection satisfied the backup constraint, and the algorithm did not generate any extra subproblems. The maximum clearance schedule was found in seconds. This again confirmed the importance of pruning, as well as the power of the matroid approach.

One can imagine other natural uses of laminar matroids. If a division j must supply an attending doctor in a specific month, we use a lower bound $\ell(S_{m,j}) = f(S_{m,j}) = 1$. In the anesthesiology example we can model a doctor who must be assigned to the operating room between 2 and 4 days each week, with at least 10 such assignments over the entire month.

3. CLEARANCE MATCHING PROBLEM

To further understand our scheduling problem we studied a simplified version that allows only clearance constraints. These constraints seem to model the essence of the real world application. They are natural and common to other applications. Studying them in isolation allows us to compare the relative power of our combinatorial ideas in a purer setting. Also as we shall see, it allows us to easily generate difficult problem instances. This section defines the clearance matching problem, demonstrates its NP-completeness and sketches our clearance matching algorithms.

Consider a bipartite graph $G = (I \cup S, E)$, where I is a set of *items* and $S = \{1, \ldots, s\}$ is a set of *time slots*. Writing $V = I \cup S$ we also have a function $f : V \to \mathbf{Z}_+$ and a nonnegative integer *clearance* c. A *clearance* c *matching* (c*-matching* for short) is a subset $M \subseteq E$ such that $d(v, M) \leq f(v)$ for every $v \in V$, and for every $i \in I$, (i, s), $(i, s') \in M$ implies $|s - s'| > c$. If $d(v, M) = f(v)$ for every $v \in I$ ($v \in V$) the matching is *complete* (*perfect*). The *complete* (*perfect*) *clearance* *matching problem* is to find such a matching for given G, f and c. It corresponds to the central constraint of the doctor scheduling problem.

3.1 NP-completeness

Itai et. al. [3] showed that for a bipartite graph with a collection of edge pairs, it is NP-hard to find a maximum cardinality matching that contains at most 1 edge from every pair. We sharpen this to the following result.

THEOREM 1. *The complete clearance 1 matching problem is NP-hard.*

PROOF. We reduce satisfiability (SAT) to complete clearance 1 matching. The clearance matching graph G_{CM} corresponding to a given instance of SAT is illustrated in Fig. 3. The item nodes of G_{CM} consist of a *clause node* for each clause and two nodes for each variable, a *decision node* d and *selector node* s. The time slot nodes are indexed from 1 to $2n + 2m$, where n is the number of variables and m is the total number of occurrences of variables in all the clauses. We shall also use n_C as the number of clauses.

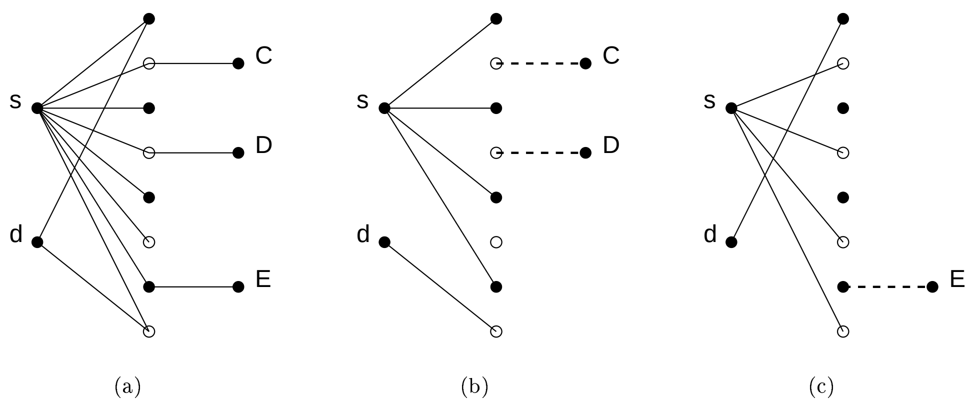


Fig. 3. (a) Edges in G_{CM} corresponding to Boolean variable x, where x occurs in clauses C and D and $\overline{x}$ occurs in E. 8 consecutive time slots correspond to x; even (odd) time slots are drawn hollow (solid). (b) Matched edges (solid) and potentially matched edges (dashed) when $x = T$. (c) The same when $x = F$. In the entire graph, one dashed edge for each clause gets matched.

We introduce some notational conventions that are used throughout this section. Writing $d[x]$ indicates that the decision node d corresponds to the Boolean variable x. Similarly for the other nodes. A time slot is *even* or *odd* depending on the parity of its number (e.g., time slot number 1 is odd).

The edges corresponding to a typical variable are illustrated in Fig. 3(a). In general a Boolean variable x that together with its complement occurs a total of t times in various clauses corresponds to $2t + 2$ consecutive time slot nodes. These time slots are adjacent to the selector node $s[x]$ and to no other selector. The first and last of these time slots are adjacent to the decision node $d[x]$ and to no other decision node. Each occurrence of a literal x ($\overline{x}$) in a clause C corresponds an edge joining an even (odd) time slot of x to the clause node for C; that time slot is not adjacent to any other clause node. The bounding function $f(v)$ is equal to 1 for v a clause or decision node and $t+1$ for v a selector node.

The given Boolean formula is satisfiable if and only if there is a complete clearance 1 matching. To prove this the key observation is that the matched edge incident to a decision node $d[x]$ uniquely determines the matched edges incident to the corresponding selector $s[x]$ because of the clearance 1 constraint. Fig. 3(b)–(c) illustrate the matching corresponding to a satisfiable formula. $\square$

Table 1 records the size of the reductions presented in this section that are used in Section 4. The reduction of the theorem corresponds to the first entry.

source problem	CM version	size of G_{CM}			
		items	time slots	edges	f-factor edges
SAT	complete	$2n+n_C$	$2n+2m$	$4n+3m$	$2n+n_C+m$
SAT	perfect	$2n+n_C+1$	$2n+2m$	$4n+5m$	$2n+2m$
IS	complete	$n+m+2$	$2n+4m$	$4n+6m$	$2n+3m$
CLIQUE	complete	$2n+2m+4$	$2n+10m$	$4n+20m$	$(3-k)n+10m$

Table 1. Size of the clearance matching problem in our reductions.

The perfect clearance 1 matching problem is shown to be NP-hard by a similar reduction. We introduce one new item node called the *padding node* p. It is adjacent to all time slots that are not neighbors of a decision node. Hence its degree is $2m$. Its bound is $f(p)=m-n_C$. The reduction is proved correct by the same argument.

We now present two more reductions to clearance 1 matching that are used in the next section, The reductions are similar to the one presented for SAT and so detailed proofs are omitted.

Fig. 4 illustrates the reduction of the independent set problem (IS) to complete clearance 1 matching. Let G_{IS} denote the graph for the given instance of IS, and G_{CM} the graph we construct for clearance matching. Let k be the size of the desired independent set of G_{IS}. All graph notations (e.g., n, $d(\cdot)$, etc.) refer to G_{IS}, and "vertex" ("node") refers to G_{IS} (G_{CM}).

G_{CM} has an *includer node* i with $f(i)=k$ and an *excluder node* e with $f(e)=n-k$. Each vertex x of G_{IS} has $2d(x)+2$ consecutive time slots. The first (last) time slot is adjacent to the includer (excluder). There is a *selector node* s for x that is adjacent to every time slot of x and has $f(s)=d(x)+1$. Each edge (x,y) of G_{IS} has an *adjacency node* a that is adjacent to 2 even time slots, one belonging to x and the other y. Every time slot is adjacent to at most one adjacency node. Every time slot and adjacency node v has $f(v)=1$.

The idea of the reduction is that vertices with a time slot matched with i (e) are in (not in) the independent set. A matched edge from i to the first time slot of a vertex x forces the selector $s[x]$ to be matched like Fig. 4(b). Thus no adjacency node $a[x,y]$ can be matched to x. Instead $a[x,y]$ must be matched to y, which forces y to be matched like Fig. 4(c). Thus y is not in the independent set.

Fig. 5 illustrates the reduction of the clique problem (CLIQUE) to complete clearance 1 matching. The notation is similar to the IS reduction, using graphs G_{CL}

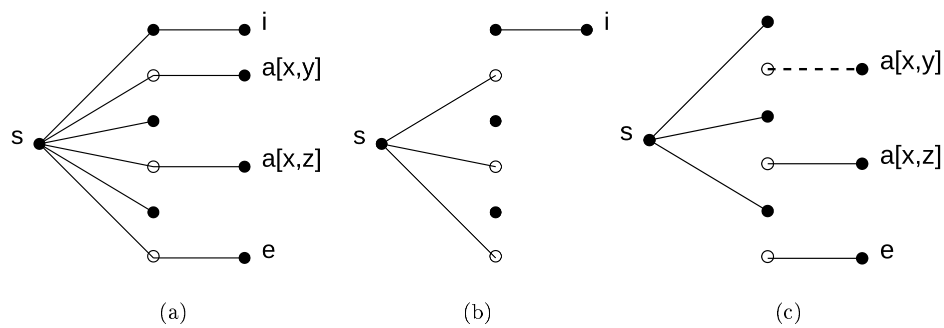


Fig. 4. (a) Edges in G_{CM} corresponding to a vertex x that is adjacent to 2 vertices, y and z. 6 consecutive time slots correspond to x; even (odd) time slots are drawn hollow (solid). (b) Matched edges when x belongs to the independent set. (c) Matched edges (solid) and potentially matched edges (dashed) when x and y do not belong to the independent set but z does.

and G_{CM}, clique size k, graph notations referring to G_{CL}, and "vertex" ("node") referring to G_{CL} (G_{CM}).

We use both vertex and edge gadgets. The vertex gadget is illustrated in Fig. 5(a)–(c). G_{CM} has a *vertex includer node* $i[V]$ with $f(i[V]) = k$ and a *vertex excluder node* $e[V]$ with $f(e[V]) = n - k$. Each vertex x has $2d(x)+2$ consecutive time slots. The first (last) time slot is adjacent to the includer (excluder). There is a *selector node* s for x that is adjacent to every time slot of x and has $f(s) = d(x)+1$. There is a *trim node* t for x that has $f(t) = d(x) - (k-1)$; t is adjacent to the time slots of x that are not neighbors of $i[V]$ or $e[V]$.

The edge gadget is illustrated in Fig. 5(d)–(f). G_{CM} has an *edge includer node* $i[E]$ with $f(i[E]) = k(k-1)/2$ and an *edge excluder node* $e[E]$ with $f(e[V]) = m - k(k-1)/2$. Each edge e has 6 consecutive time slots. The first (last) time slot is adjacent to the excluder (includer). There is a *selector node* s for e that is adjacent to every time slot of e and has $f(s) = 3$. There is also an *adjacency node* a for e that is adjacent to the third and fifth time slots of e and has $f(a) = 2$. In addition a is adjacent to an odd time slot of each of its endpoints (and these time slots are adjacent to no other adjacency nodes). Each time slot t has $f(t) = 1$.

The idea of the reduction is that vertices with a time slot matched with $i[V]$ ($e[V]$) are in (not in) the clique, and edges with a time slot matched with $i[E]$ ($e[E]$) are clique (non-clique) edges. Let C denote the set of vertices matched with $i[V]$. The matched edge from $i[V]$ to the first time slot of a vertex $x \in C$ forces the selector node $s[x]$ to be matched like Fig. 5(b). The trim node $t[x]$ allows at most $k-1$ odd time slots of x to be matched to adjacency nodes. Fig. 5(c) shows the configuration of any vertex not in C; none of its odd time slots are matched to adjacency nodes. Hence at most $k(k-1)/2$ time slots of vertices (of V) can be matched to adjacency nodes.

A matched edge from $i[E]$ to the last time slot of an edge e forces the selector node $s[e]$ to be matched like Fig. 5(b). Its adjacency node is matched to odd time slots belonging to its two end vertices. Thus at least $2f(i[E]) = k(k-1)$ odd time slots of vertices of C are matched to adjacency nodes. From the previous paragraph

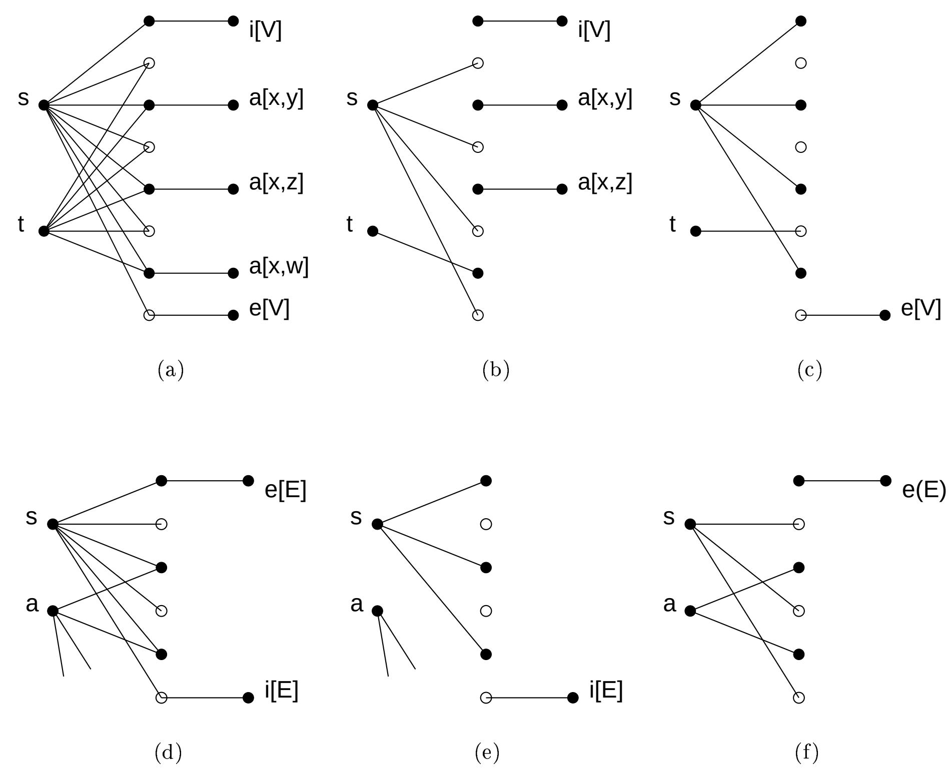


Fig. 5. (a) Edges in G_{CM} corresponding to a vertex x that is adjacent to 3 vertices y, z, w when $k = 3$. 8 consecutive time slots correspond to x. As usual in all parts even (odd) time slots are drawn hollow (solid). (b) Matched edges when x, y and z but not w are in the clique. (c) Matched edges when x is not in the clique. (d) Edges in G_{CM} corresponding to an edge e. 6 consecutive time slots correspond to e. (e) ((f)) gives the matched edges when e is (is not) a clique edge.

we conclude equality holds in this estimate. This implies C is a clique.

3.2 Algorithms

We implemented both the basic algorithm and a matroid algorithm for the general clearance c matching problem. The implementation details of Section 2.5 apply to these new programs with two exceptions: remarks about division constraints (there are none) and hardest-first numbering (this is not part of the program; more precise ordering schemes are studied in Section 4). The rest of this section describes the matroids used in the matroid algorithm.

Both matroids have ground set E. Matroid $\mathbf{M}_2$ models the degree constraints of the time slots. That is $\mathbf{M}_2$ is a partition matroid, where each time slot i corresponds to a block with bound $f(i)$.

$\mathbf{M}_1$ is a laminar matroid that models each item and a subset of its clearance constraints, as follows. The laminar family has a set S_i for each item i, containing every edge incident to i. Each S_i is partitioned into sets $S_{i,j}$ in some way that guarantees each $S_{i,j}$ consists of pairwise conflicting edges, i.e., (i,s), $(i,s') \in S_{i,j}$ implies $|s-s'| \le c$. The bounding function has $f(S_i) = f(i)$ and $f(S_{i,j}) = 1$. Our program constructs the $S_{i,j}$ in a greedy way: It repeatedly chooses as the next set $S_{i,j}$ the largest set of pairwise conflicting edges containing the edge (i,s) that has the smallest value s not yet in a set. Other strategies are possible.

Any clearance c matching corresponds to a matroid intersection. The converse does not hold since two conflicting edges can belong to different subsets $S_{i,j}$. However it is obvious that this model prunes the solution space better than the f-factor model of the basic algorithm.

On the other hand an augmenting path for the matroid algorithm can be decidedly longer than one for the basic algorithm. To see why let I be the current intersection of the matroid algorithm. Consider an item i with S_i partitioned into sets $S_{i,j}$, $j = 1, \ldots, r$. Suppose that each of these r sets contains an edge of I. (In the matroid algorithm this is the case for every item i except one.) Recall that in an augmenting path 2 consecutive edges e, e' with $e \in I$, $e' \notin I$ satisfy the condition $e \in C_1(e', I)$. If e is incident to item i this condition means that e and e' belong to the same set $S_{i,j}$. Hence the augmenting path can pass through item i up to r times. In contrast an augmenting path in the basic algorithm passes through i at most once.

Our test cases will have items with large values of r. Hence the potential difference in augmenting path length can be large. It may not be entirely clear which of our algorithms, basic or matroid, will perform better.

4. EXPERIMENTS

4.1 Motivation

We conducted a number of experiments on implementations of the basic and matroid algorithms for clearance matching. Our experimental work was motivated by these four goals:

1. To discover how well the algorithms performed on problems that were larger and more difficult than the real-life problems at Denver Health.
2. To compare the efficiency of the basic and matroid algorithms.
3. To study the effects of different ordering rules on performance of the algorithms.
4. To discover new ideas to improve performance of the algorithms.

Regarding 1, in contrast to the real-life problems our experiments do not have division constraints. However Section 3 has shown it is the clearance constraints that make the problem hard. In fact one can easily modify our matroid algorithm for clearance matching so it handles division constraints automatically, i.e., all solutions satisfy the division constraint. (To do this model clearance constraints using laminar matroid $\mathbf{M}_1$ of Section 3.2 and model division constraints using laminar matroid $\mathbf{M}_2$ of Section 2.8.)

Regarding 2, we have indicated that our intuition is the matroid algorithm is faster because of better pruning. But Section 3.2 shows that augmenting paths in the matroid algorithm can be longer than in the basic algorithm.

Regarding 3, Section 2.5 indicates how bad ordering decisions can increase the number of subproblems. However we are unaware of tools for analyzing how good an ordering is or designing a good order.

Similarly for 4 there seems to be no way of refining the design of our algorithms for greater efficiency from an analytic point of view. Hence the necessity of experimental evaluation.

4.2 Experimental setup

All coding was done by one person in C, using a consistent coding style. All tests were run on a Pentium 200MHz MMX computer with 32 megabytes of RAM under the Linux kernel 2.0.30. Our main measurements were user time and number of recursive calls. The latter, as mentioned in Section 2.4, is precisely the total number of subproblems generated. The former can be imprecise because of various system overheads.

Some larger runs were actually executed on a more powerful computer (Pentium III 800MHz with 128 megabytes of RAM). These tests only reported the number of recursive calls, not user time. Since this is platform-independent the switch is inconsequential.

For most experiments we imposed a time limit (60, 90, 180 or 600 seconds of wall clock time). Any program execution that did not complete within its time limit was terminated. Terminated runs pose a problem in reporting statistics such as mean user times or mean number of subproblems. This problem is compounded when we wish to compare several programs working on the same problem and some executions succeeded while others were killed. In reporting means in these situations we usually use a philosophy similar to the trimmed mean: If only a few runs were killed we calculated the means by discarding the problem instances where one or more programs were terminated. Since the number of discarded values is small using such a trimmed mean should not skew the results. In situations where the trimming strategy seemed like it might effect the presented results, we discussed it in more detail.

Finally note that preliminary details of the experiments that are omitted from this paper are in [4].

4.3 Test families

We studied two types of clearance matching problems, reduced instances of NP-hard problems and random instances.

The reduced instances were constructed by applying the transformations of Section 3 to problem generators from DIMACS Implementation Challenges [1]. First we reduced satisfiability problems to both complete and perfect clearance 1 matching. Our 461 SAT instances had from 5 to 10 variables, and clauses numbering in the double-digits or low hundreds (an average of 21 clauses). To illustrate representative problem sizes, an instance of 3SAT having 10 variables and 134 clauses transformed to a graph with 155 items, 752 slots and 1906 edges; another instance with 15 variables and 50 clauses transformed to 81 items, 328 slots and 806 edges.[3]

Next we studied complete clearance 1 matching problems that were transformed clique problems. All clique instances were constructed using the `sanchis` generator from the DIMACS site. The clique graphs had from 6 to 10 vertices, 5 to 39 edges and clique sizes between 2 and 5. We used both the IS reduction and the CLIQUE reduction of Section 3. From now on we refer to the latter as the CL reduction. For the former we interpreted a clique problem as an independent set problem on the complementary graph and reduced that. To illustrate representative problem sizes, for the IS reduction a clique instance with 6 vertices and 7 edges transformed to a c-matching problem with 16 items, 44 time slots and 72 edges; an instance with 10 vertices and 22 edges with a maximum clique size of 5 transformed to 35 items, 112 time slots, and 178 edges.[4] For the CL reduction, the first of these instances transformed to 27 items, 82 time slots and 154 edges.[5]

Our clique-reduced problems are smaller than the satisfiability-reduced problems because they were found to be more difficult. This could be caused by the generators or the reductions – it was unclear which. Furthermore the CL reduction usually produced harder problems than IS (see Section 4.5). For that reason fewer CL problems were studied.

We also studied a simple family of random instances of perfect clearance 1 matching. The family is parameterized by the number of items $|I|$, number of time slots $|S|$, degree constraint values f_i and f_s, and a lower bound on the number of edges m_0. The bounding function $f(v)$ of a corresponding graph is equal to f_i (f_s) for $v \in I$ ($v \in S$). We require $f_i > 1$ (so there are operative clearance constraints) and $f_i|I| = f_s|S|$ (otherwise no perfect c-matching exists). A random instance is chosen by first generating f_i random edges incident to each item i; then generating a minimal number of random edges to ensure that each time slot s is incident to at least f_s edges; and finally generating more random edges to ensure the lower bound m_0 is met. All edges are chosen uniformly at random, not allowing parallel edges to occur.

---

[3]These sizes differ slightly from those predicted by the 2nd line of Table 1 because the generators can give clauses containing a literal and its complement. Our program allocated only 2 slots total to those 2 occurrences, not 4 as described in Section 3.

[4]The figures presented in this paper deal exclusively with the $n = 6$ graphs. However we ran larger graphs like these, and they gave consistent results.

[5]These sizes differ slightly from Table 1 because the program discarded trim nodes t with $f(t) = 0$.

We used two values of f_s, 1 (as in the families of reduced instances) and 4 (as in the real-life scheduling problem). For $f_s = 1$ ($f_s = 4$) we varied $|I|$ between 10 and 50 (10 and 100) in steps of 10. The rest of the process was the same for both values of f_s: We varied $|S|$ independently from $|I|$, between 10 and 100 in steps of 10. For each such $|I|$, $|S|$ pair with an integral $f_s > 1$, we used 6 values of m_0 evenly spaced between $f_s|S|$ and $|I||S|$ (the complete bipartite graph). For each such set of parameters we generated 3 c-matching instances. (Note that two random problems on same the complete bipartite graph will be different because of different adjacency list orderings.) Representative problem sizes were 30 items, 60 time slots and 930 edges for $f_s = 1$; 50 items, 50 time slots and 1350 edges for $f_s = 4$.

4.4 Random instances

Each of our algorithms behaved similarly on the two families $f_s = 1$ and $f_s = 4$. We discuss the family $f_s = 1$ first. 755 of the random graphs had a perfect c-matching. Both of our algorithms were efficient on these graphs: The mean time was less than 0.09 (0.03) seconds for the basic (matroid) algorithm. A small number of killed runs were ignored in these means. A small but slightly larger number of graphs took disproportionately large amounts of time. Usually the matroid solver was faster on these graphs.

162 of the random graphs did not have a perfect c-matching. Of these all but 19 were immediately rejected by the basic algorithm because there was no f-factor. All 162 were immediately rejected by the matroid algorithm because there was no perfect matroid intersection.

For the family $f_s = 4$ we generated 500 instances with a perfect c-matching and 217 with no such matching. Of these 217 instances, 84 were immediately rejected by the basic algorithm because of no f-factor. Of the remaining 133, the basic algorithm did not terminate after 60 seconds on 36 of them. In contrast none of the 217 instances had a valid matroid intersection, and so they were all immediately rejected by the matroid algorithm.

To summarize, the results on random instances illustrated the benefits of the better pruning by the matroid algorithm, particularly on infeasible problems.

4.5 Orderings

As mentioned in Section 2 our algorithms' efficiency is sensitive to the ordering (numbering) of the items in the clearance matching problem. This ordering determines adjacency list order, and the algorithm constructs its partition (1) based on the lowest numbered item with a clearance conflict. To better understand the effects of orderings, we experimented on different orderings for our SAT- and CLIQUE-reduced instances.

As will be seen, our orderings are defined in terms of the reductions used. Hence from the point of view of clearance matching the orderings may seem artificial. However the orderings do correlate with certain structural properties of the clearance matching instances. Also as will be seen, our experiments reveal phenomena that can be attributed to entirely to the orderings.

We first discuss orderings for transformed SAT problems. In all such orderings the clause nodes c and the padding node p (if present) were assigned the highest numbers. The logic for this decision is clear: the matched edges incident to these nodes are essentially determined by the matched edges incident to the other nodes, the decision nodes d and selectors s.

The order for the d and s nodes is specified by an order name of the following form:

$$\{ds, sd, DS, SD\}[-,+][\_\mathrm{nop}]$$

To describe the corresponding orders let x_i, $i = 1, \ldots, n$ be the SAT variables.

The first part of an order name is one of the 4 strings listed in curly braces. ds signifies the alternating order $d[x_1], s[x_1], \ldots, d[x_n], s[x_n]$. DS signifies the blocked order $d[x_1], \ldots, d[x_n], s[x_1], \ldots, s[x_n]$. sd and SD are defined analogously.

If the middle part of the name is empty the variables x_i, $i = 1, \ldots, x_n$ are ordered as in the given SAT instance. If the middle part is $-$ the variables are ordered by nonincreasing degree, i.e., x_i occurs at least as many times as x_{i+1}. Similarly $+$ means nondecreasing degree.

If the third part of the name is _nop there is no padding node in the reduction (i.e., complete c-matching). An empty third part means there is a padding node (perfect c-matching).

By eyeballing plots like Fig. 6 we hypothesized that the DS and DS_nop orderings were worse than the others. We verified this in a number of experiments, with results like Table 2.

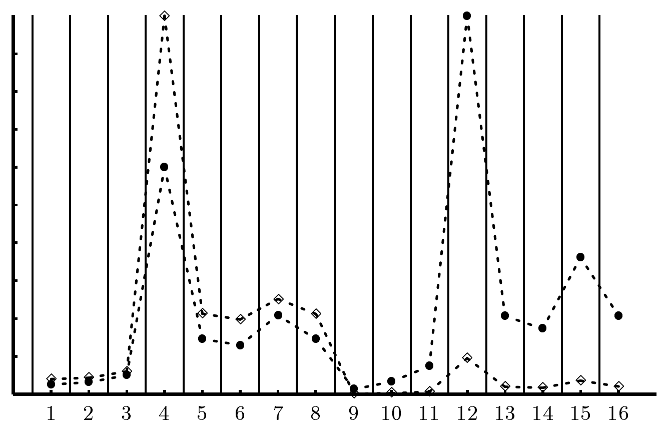


Fig. 6. Plot of the user time ($\diamond$) and number of recursive calls ($\bullet$) the matroid algorithm used when solving clearance matching instances reduced from a satisfiable Boolean formula with 5 variables and 34 clauses. The maximum displayed user time is 54.24 seconds, for order (4), DS. The maximum number of recursive calls needed was 65439, for order (12), DS_nop. The orderings specified along the horizontal axis are (1) ds, (2) $ds+$, (3) $ds-$, (4) DS, (5) sd, (6) $sd+$, (7) $sd-$, (8) SD, (9) ds_nop, (10) $ds+$_nop, (11) $ds-$_nop, (12) DS_nop, (13) sd_nop, (14) $sd+$_nop, (15) $sd-$_nop, (16) SD_nop.

We can give a plausible explanation of this observation. Let x_i, $i = 1, \ldots, n$ be the ordered list of SAT variables. A decision variable d has $f(d) = 1$ and so is never involved in a conflict. Hence in a DS option the matched edges for $d[x_1], \ldots, d[x_n]$

denominator	sample size	sample mean	95% confidence interval
ds	123	24.60	$[14.88, 34.33]$
$ds-$	124	44.65	$[22.40, 66.89]$
$ds+$	126	58.81	$[1.83, 115.79]$
sd	126	33.09	$[5.44, 60.74]$
$sd-$	114	69.21	$[30.82, 107.59]$
$sd+$	124	53.65	$[-4.47, 111.77]$
SD	119	35.33	$[6.09, 64.58]$
$SD-$	110	72.96	$[32.02, 113.89]$
$SD+$	124	54.95	$[-3.63, 113.52]$

Table 2. Statistics of the ratios of the time required by the basic algorithm on satisfiable Boolean formulas. The time for DS is in the numerator. Thus the first line indicates that the DS order was over 24 times slower than ds.

will almost always be fixed (i.e., they will usually be in the set F_{i-1} of the current partition (1).) Each such set of n fixed edges corresponds to a truth assignment. Each such truth assignment that is not satisfying will generate many subproblems before it is rejected. This makes orderings with the DS option inefficient. None of the other options have such an obvious vulnerability.

Our experiments also revealed groups of orderings with essentially the same performance on unsatisfiable instances, e.g., ds, sd, and SD. This is illustrated in Table 3 below.

denominator	algorithm	field	sample size	sample mean	95% CI
sd	basic	time	58	1.01	$[0.99, 1.02]$
SD	basic	time	58	1.00	$[0.99, 1.02]$
sd	basic	rec calls	58	1.02	$[1.00, 1.03]$
SD	basic	rec calls	58	1.02	$[1.00, 1.03]$
sd	matroid	time	85	1.02	$[0.99, 1.05]$
SD	matroid	time	85	1.01	$[0.99, 1.04]$
sd	matroid	rec calls	85	0.98	$[0.96, 1.01]$
SD	matroid	rec calls	85	0.99	$[0.97, 1.01]$

Table 3. Statistics for the ratios of the time and number of recursive calls required by the basic and matroid algorithms on unsatisfiable Boolean formulas with respect to ds, sd, and SD. The value for ds is in the numerator.

We can explain some of these observations rigorously. We will prove that for SAT-reduced instances of c-matching, the sd and SD options are essentially the same. More precisely choose an algorithm, basic or matroid. Fix a SAT-reduced instance of c-matching, along with its adjacency list representation. Also fix the complete c-matching that both algorithms use as their initial matching. Choose one possibility for the second part of the order name, i.e., empty, + or −. Similarly choose a third part, empty or _nop. The *sd-execution* refers to the execution of the algorithm using the sd option, plus all the choices fixed above. Similarly for the *SD-execution*.

Recall that for the basic or matroid algorithm, a recursive call is for a subproblem in (1). To be more precise we write that problem showing all the edges that have been fixed in recursive calls, viz.

$$\mathcal{F}[F_{i-1},\ O \cup e_i]. \tag{3}$$

Here F_{i-1} is the set of edges that have been fixed in the matching, over all previous recursive calls. O is the set of edges that were fixed to be not in the matching, over all previous recursive calls. e_i is an edge that was in the previous matching and has just been fixed to be not in the matching. Let ι be the item that is incident to e_i. The recursive call for subproblem (3) attempts to make the matching complete or perfect by finding an augmenting path that starts at ι.

Say that a such recursive call is *instantaneous* if it returns without finding an augmenting path, after scanning just 1 edge.

PROPOSITION 2. *The sd-execution makes a number of instantaneous recursive calls not made by the SD-execution. Aside from these there is a step-by-step correspondence between the two executions such that at all times they have the same matching and corresponding sets of fixed edges.*

PROOF. For definiteness we phrase the argument in terms of the orders *sd*_nop and *SD*_nop (i.e., complete c-matching). This simplifies some notation but the same argument applies to the other versions. It will be helpful to refer to Fig. 3.

The *sd* order is

$$s[x_1], d[x_1], \ldots, s[x_n], d[x_n], C_1, \ldots, C_r,$$

where C_i, $i = 1, \ldots, r$ denotes the clause nodes. The *SD* order is

$$s[x_1], \ldots, s[x_n], d[x_1], \ldots, d[x_n], C_1, \ldots, C_r.$$

Consider the recursive call for subproblem (3) in the *sd*-execution. We show the proposition holds for this call by induction on the number of recursive calls in the *sd*-execution.

A priori there are 3 possibilities for the above item ι: it is either a selector $s[x_j]$, a decision node $d[x_j]$ or a clause node C_j. We first show the third case $\iota = C_j$ never occurs: Once the *sd*-execution fixes the matched edge incident to the decision node $d[x_n]$, we have a complete c-matching. (There cannot be a conflict for a clause node C_k since $f(C_k) = 1$.) Hence at that point the algorithm returns. This implies that a clause node C_j never gets an incident edge fixed in a recursive call, i.e., it can never be ι.

Now we consider the two remaining cases for ι.

Case 1. $\iota = s[x_j]$.

In the *sd*-execution the set of matched edges that have been fixed, F_{i-1}, consists of the matched edges incident to $s[x_k]$ and $d[x_k]$, $k < j$, plus strictly fewer than $f(s[x_j])$ matched edges incident to $s[x_j]$. O contains various edges incident to $s[x_k]$, $k \le j$.

By induction assume the 2 executions currently have the same matching M. Also the *SD*-execution has fixed the same edges incident to $s[x_k]$, $k \le j$, as *sd*, but no other edges (i.e., no edges incident to decision nodes are fixed).

Consider an arbitrary Boolean variable x with selector node s, decision node d, first and last time slot α and ω respectively. If (s, α) is fixed in M, any complete matching must contain (d, ω). Similarly if (s, ω) is fixed in M, any complete matching must contain (d, α).

This implies that for $k < j$, the matched edge incident to $d[x_k]$ (which is fixed in the *sd* ordering) must be included in any complete matching in the current partition for the SD ordering. (It is fixed implicitly rather than explicitly.)

This implies that the *sd*- and SD-executions search the same subgraph to find an augmenting path. If there is no augmenting path both recursive calls return. In the opposite case both executions find the same augmenting path and use it to form the same new complete matching. If this matching has a conflict they find the same next conflict to partition on (as in (1)). This gives rise to the same family of subproblems, except that the SD-execution never creates a subproblem with ι equal to a decision node d. So the two executions have the desired correspondence.

Case 2. $\iota = d[x_j]$.

It is easy to see that this recursive call in the *sd*-execution is instantaneous: Only 1 unmatched edge (ι, κ) is incident to ι (κ is the first or last time slot of x_j). The matched edge incident to κ is $(\kappa, s[x_j])$. Hence that edge is fixed (i.e., in F_{i-1}). Thus the recursive call returns after scanning (ι, κ), so it is instantaneous. □

All our data was consistent with the proposition. Table 4 gives illustrative exact figures.

ordering	algorithm	recursive calls		
sd	basic	350889	296113	391587
SD	basic	350689	295947	391374
sd	matroid	23588	21992	25480
SD	matroid	23407	21832	25292

Table 4. Number of recursive calls for *sd* and SD orderings on 3 unsatisfiable instances of 3SAT. The instances had $n = 10$, $n_C = 27$.

On the other hand it is not entirely clear what the implications of the proposition are for execution time. The instantaneous recursive calls are few enough and fast enough that lower level coding differences are significant (e.g., decision nodes can be deleted in *sd* but actually scanned in SD). Table 5 indicates that the *sd* option may be slightly faster. However the edge in user time is minor and some exceptions were observed. (Table 2 shows such exceptions. However the confidence intervals are relatively large.)

algorithm	satisfiable?	sample size	sample mean	95% CI
basic	Y	138	0.97	$[0.92, 1.01]$
basic	N	58	1.00	$[0.99, 1.00]$
matroid	Y	178	0.93	$[0.87, 0.98]$
matroid	N	85	1.00	$[0.98, 1.03]$

Table 5. Statistics for the ratio of user time for the *sd* (numerator) and SD (denominator) orderings. The basic and matroid algorithms were run on satisfiable and unsatisfiable SAT-reduced instances.

Turning to the ds option, it has a high-level similarity to sd and SD. Specifically in a subproblem (3) with $\iota = s[x_j]$, all three options fix the matched edges incident to $s[x_k]$ and $d[x_k]$, $k < j$. The difference is that $d[x_j]$ has a fixed matched edge in the ds option. This can lead to differences in behavior not easily described. However the high-level similarity certainly contributes to the similar observed performance.

Next we examine orderings for CLIQUE problems transformed by the IS reduction. Let x_i, $i = 1, \ldots, n$ be the vertices of the clique graph, as ordered by the clique generator. Let e_i, $i = 1, \ldots, m$ denote the edges of the complement of the clique graph in natural lexicographic order (an edge is represented as (x_i, x_j) where we assume $i < j$).

We looked at 3 blocked orderings, *IAS*, *SAI* and *ASI*. I denotes a block for includer nodes and excluders, ordered i, e. A (S) denotes a block consisting of the adjacency (selector) nodes. Thus for example *IAS* orders the items as i, e, $a[e_1], \ldots, a[e_m]$, $s[x_1, \ldots, s[x_n]$.

For both the basic and matroid algorithms, the orderings listed by increasing efficiency were *IAS* (worst), *ASI*, and *SAI* (best). This is illustrated in Fig. 7–8. (Corresponding figures for *ASI* are omitted but are as expected.) In both figures note the difference in time scale between (a) and (b)!

Before continuing we should explain these figures in greater detail. Fig. 7–8 and all subsequent figures dealing with the *IAS* and *SAI* instances are from the same suite of experiments. All the data is from CLIQUE problems transformed by the IS reduction. In figures plotting user time the timer has a resolution of 100 ticks per second. This causes some discretization, e.g., Figure 8(b). The plot symbol '×' means some program in the comparison did not complete within 60 seconds.

Each x-value from 0 to 25 plots the time for 14 or 15 CLIQUE instances. Identical times get overprinted, e.g., it appears that only 1 run is plotted for x-value 0 in Fig. 7(a) but there are 15.

The CLIQUE instances are numbered from 0 to 25 so that, roughly speaking, difficulty increases with increasing number. Here are (some of) the parameters used in the `sanchis` generator to construct the graphs. All graphs had $n = 6$ vertices. e denotes the requested number of edges; c denotes the requested maximum clique size. The actual clique graph generated may have smaller values of both parameters, but usually there was no decrease in these tests. These requested parameters are listed in Table 6. Finally note that in all exeriments the IS problem used a value $k = 3$, i.e., we tested for the existence of an independent set of 3 vertices. Thus some of the instances are feasible and some are not.

	0	1	2	3	4	5	6	7	8	9	10	11	12
e	14	13	13	12	12	12	11	11	11	10	10	10	9
c	5	5	4	5	4	3	5	4	3	5	4	3	4

	13	14	15	16	17	18	19	20	21	22	23	24	25
e	9	9	8	8	8	7	7	7	6	6	6	5	5
c	3	2	4	3	2	4	3	2	4	3	2	3	2

Table 6. Reqested parameters for clique generator.

We offer the same explanation for the inefficiency of the IAS ordering as for the DS ordering for SAT: The includer and excluder are never involved in conflicts, since neither node is adjacent to 2 consecutive time slots. Hence in IAS their matched incident edges will almost always be fixed, and those incident edges corresponding to invalid independent sets will generate many wasted subproblems.

We did not extensively study ordering strategies for problems using the CL reduction. The problems tended to be much harder than their IS counterparts. For instance a clique problem was solved using the IS reduction in 57 recursive calls, but the CL reduction was terminated after $22,653,202$ recursive calls in 180 seconds on our faster computer. On the other hand the CL reduction sometimes actually beat the IS reduction (using analogous orderings). We believe the CL problems are more difficult to understand because they contain more gadgets. Nonetheless some results on CL problems are included in subsequent sections.

4.6 Basic versus matroid algorithm

Section 4.4 gives evidence for the greater efficiency of the matroid algorithm on random instances. This section continues the comparison using data for the SAT and CLIQUE families.

First consider SAT. We will just summarize the most striking aspects of our measurements. Unsatisfiable Boolean formulas require searching the entire space. Hence we expect the matroid algorithm to perform at its best on these. The time advantage over the basic algorithm was observed to be a factor of 3 or more. When no padding node was used it was a factor of 10. For satisfiable Boolean formulas with no padding node and selector nodes first the matroid algorithm was a factor of 4 or more faster.

Next consider the IS reduction for the CLIQUE family. Fig. 7–8 show an advantage in execution time for the matroid algorithm of about 1 order of magnitude. Fig. 9 shows how the number of recursive calls compare (on the same test suite). This figure and subsequent ones plot the value of $\ln r$ for a ratio r. This eliminates the bias in choice of numerator, since r and $1/r$ become equidistant from the horizontal axis. The plots show the advantage in number of recursive calls is roughly the same as the time advantage.

Recall from Section 3.2 that a possible disadvantage of the matroid algorithm is longer augmenting paths. This would translate into longer search time for each augmenting path. Fig. 9 seems to indicate this is not the case, since the time advantage is close to the advantage in number of augmenting paths. Fig. 10 compares the two algorithms in terms of number of calls to procedure `DFS` of Fig. 1. This number correlates closely with total search time. The ratios appear to be close to the ratios of Fig. 9. This supports the conclusion that the matroid algorithm does not have a significant disadvantage in terms of search time.

We also looked at the CL reduction. These experiments ran on the faster PIII computer. An execution was terminated if it did not complete in 180 seconds. Fig. 11 shows the results by plotting the ratio of number of recursive calls. First note that we have not changed the numbering of the `sanchis` clique problems. So since CL reduces clique instances directly, the size of clearance matching instances now decreases with increasing number. The basic algorithm was terminated on all the harder problems of the suite; the matroid algorithm was terminated on all instances of problem 14 but no others. The advantage of the matroid algorithm is decided and increases with problem size.

4.7 Move-to-front ordering

It seems quite difficult to design a simple vertex numbering scheme that always leads to a small number of subproblems. As another approach we investigated a self-adjusting strategy.

We first review how problems are partitioned in our implementation of the basic algorithm. In the given clearance matching graph $G = (I \cup S, E)$ the items get numbered from 1 to $|I|$. This determines the adjacency list order of the edges. Consider an invocation of the basic algorithm on the family $\mathcal{F}_i = \mathcal{F}[F_{i-1},\ e_i]$ of (1). (It is convenient to use this notation which is less explicit than (3), but we are still referring to an invocation at the end of a possibly long chain of recursive calls.)

The invocation finds a new f-factor $F_{i-1} \cup F'$ that does not contain edge e_i. Let the edges of F' in adjacency list order be e'_j, $j = i, \ldots, r$. Choose k as the smallest index such that $F_{i-1} \cup \{e'_i, \ldots, e'_k\}$ contains a conflict. In the basic algorithm the next partition consists of the $k - i + 1$ families

$$\mathcal{F}[F_{i-1} \cup \{e'_i, \ldots e'_{j-1}\},\ e'_j],\ j = i, \ldots, k.$$

Let edge e'_k be incident to item number κ. Hence the current conflict is a violation of the clearance constraint of κ. Intuitively κ alone may be the source of the conflict. That is, given that the edges F_{i-1} are matched it may be impossible to match the edges incident to κ without violating its clearance constraint. To investigate this possibility immediately, we swap κ and another item node to make κ come right after the fixed edges F_{i-1}. More precisely let edge e_i (sic) be incident to item number ι. Assume $\iota + 1 < \kappa$. (If not we will just use the basic algorithm's partition.) The item currently numbered κ gets the new number $\iota + 1$, and the current $\iota + 1$ gets new number κ. (We cannot swap κ and ι since some fixed edges F_{i-1} may be incident to ι.) This gives a new adjacency list order. In this order let the edges of F' be e''_j, $j = i, \ldots, r$. Let e'_k be edge e''_ℓ. The next partition consists of the $\ell - i + 1$ families

$$\mathcal{F}[F_{i-1} \cup \{e''_i, \ldots e''_{j-1}\},\ e''_j],\ j = i, \ldots, \ell.$$

This is fewer subproblems than the basic algorithm, since $\iota + 1 < \kappa$ implies $\ell < k$. If our initial guess is correct we will discover infeasibility using fewer subproblems than the basic algorithm. We call this modification the *move-to-front heuristic*. It can be used with both the basic and matroid algorithms.

To illustrate the advantage of move-to-front in an extreme case, recall from Section 2.5 the example of an item i that has only one feasible c-matching. One can imagine how i can start out with a high number, be involved in a number of conflicts that decrease its number, until eventually i becomes item number 2. At that point it is much easier to get i matched without conflicts. (A simple modification of the heuristic allows i to become item 1. But the modification did not offer significant advantages in our experiments so we ignore it.)

On the other hand it is unclear if move-to-front will enjoy similar success on small groups of items that are highly constrained. Also if the guess turns out to be wrong, move-to-front wastes the effort spent verifying that the edges $e'_{i+1}, \ldots, e'_{k-1}$

have no conflicts.

We tested move-to-front versions of the basic and matroid algorithms on all our families. For random instances with $f_s = 1$, both move-to-front versions generally found solutions faster (and with fewer recursive calls). As an example of the possible extreme benefits we mention a random instance that had a perfect c-matching with $|I| = 10$, $|S| = 80$, and $|E| = 224$. The matroid algorithm did not find a solution in 18 minutes and over 19M recursive calls. With move-to-front it found a solution in 0.01 seconds and 22 recursive calls.

On the random family with $f_s = 4$, we observed the move-to-front matroid algorithm to be superior. In addition, of the 500 instances with a perfect c-matching, the basic algorithm was terminated after 60 seconds on 101 instances without finding a solution. But when move-to-front was added it found solutions for 35 of these. Similarly the matroid algorithm was terminated on 10 instances, but with move-to-front it solved all of them (in fact very fast).

Turning to the SAT family, on unsatisfiable Boolean formulas using the move-to-front matroid algorithm, the DS ordering was no longer worse than the others. For example on 85 instances the user time for DS was only 1.03 times that for ds, and similar for the other orderings. Move-to-front quickly eliminates the "mistakes" in that ordering. Furthermore the sample means and confidence intervals in our statistics are much more tightly grouped than before. This further suggests that the move-to-front matroid algorithm is not as sensitive to the initial ordering.

Finally we summarize experiments on the CLIQUE family. Fig. 12–13 are the move-to-front analogs of Fig. 7–8. For the IAS ordering (the (a) subfigures) adding move-to-front caused the time to decrease by about two orders of magnitude for both algorithms. For SAI ((b) subfigures) the basic algorithm perhaps shows a small speedup on the larger instances. Plots for the matroid algorithm are show little difference, although the values are so small that the timer resolution is significant.

To investigate this further Fig. 14–15 show the ratios of number of recursive calls for both algorithms. Ratios for IAS ((a) subfigures) are consistent with the 2 order of magnitude improvement. The ratios for SAI ((b) subfigures) show a definite but small improvement in the number of recursive calls. We also looked at ratios of number of calls to `DFS`. As before they agreed with ratios of number of recursive calls, i.e., time for a search does not change. Since IAS is well understood we do not show its ratios; the ratios for SAI are in Fig. 16.

Finally we examine CL instances. The figures are in terms of ratio of number of recursive calls. Fig. 17(a), in spite of the terminated runs, suggests that the basic algorithm improves using move-to-front, and the advantage grows with problem size. Fig. 17(b) shows the improvement on the matroid algorithm is more modest, and a nonneglible number of runs actually got worse. It seems plausible that the improvement should be less dramatic, but we do not have a good explanation for the worsening of behavior. Fig. 17(c) shows that incorporating move-to-front, the matroid version continues to have a very large advantage over the basic version.

4.8 Summary

The first goal of our experiments was to investigate the performance of the algorithms on problems that were larger and more difficult than the real-life ones. The real-life problems, with a few hundred edges, are perhaps structurally comparable to our random problems, but our random problems had a thousand edges and more. As expected our transformed problems were decidedly harder than the random ones. For SAT they had about a thousand edges and for CLIQUE they were a bit smaller than the real-life ones.

In general the algorithms were more challenged by the transformed instances than the larger random or real-life problems. Nonetheless the algorithms usually succeeded in solving the more challenging problems in reasonable time. But we can predict that an order of magnitude increase in size or difficulty of the problems will stretch our algorithms to their limits.

For the second goal, the superiority of the matroid algorithm over the basic algorithm was clearly shown. The possible disadvantage that we posited, in terms of longer time for a search, was not observed. Our experiments are entirely consistent with the intuition that high quality pruning of the search space is the most important goal in solving problems such as ours.

Our third goal was to gain insight into ordering rules. The orderings we investigated had the defect of being defined in terms of NP-hardness reductions. It would be more attractive to study orderings based on structural properties of the problem. This however seems very difficult. The experiments we conducted on several such orderings were inconclusive [4].

The most striking conclusion of our work on orderings was confirming the "tightly constrained-first" strategy used in the Denver Health system. We confirmed this principle at the opposite end (i.e, that unconstrained items should be ordered last) by observing the poor performance of DS and IAS orderings. Interestingly this opposite end strategy was not used in the Denver Health system. Although "loosely constrained-last" is logically equivalent to "tightly constrained-first," the definition of the terms is fuzzy, so in practice the rules are not equivalent. We will incorporate "loosely constrained-last" into the Denver Health system.

Our last goal was to discover improvements to the performance of the algorithms. Before our experiments we were certainly aware of the possibility of using a self-adjusting strategy like move-to-front. However there was little reason to believe it would be as effective as we discovered. The advantages of move-to-front were consistently observed.

5. CONCLUSIONS

We have developed a program that solves a real-world scheduling problem. It uses combinatoric ideas to prune a large search space to a manageable size. This successful pruning has been demonstrated on a variety of real-world data and experimental data.

Our experiments also expose issues that deserve further investigation, such as ordering decisions in backtracking programs like ours. Our results on move-to-front are encouraging. It might be worthwhile investigating other self-adjusting strategies, including randomized strategies. Another avenue for future work is to gain insight into the relatively small number of data sets that violate our general conclusions. When is matroid or move-to-front *less* efficient? Is it possible to predict that an execution in progress will exceed a reasonable time limit? (This would allow the program to restart with a different strategy.) We are continuing work on our current group of experiments, and we also hope to look into these other issues.

Acknowledgment

We thank Dr. Patricia Gabow for help in designing the system specifications.

REFERENCES

[1] Dimacs implementation challenges. `ftp://dimacs.rutgers.edu/pub/challenge`.

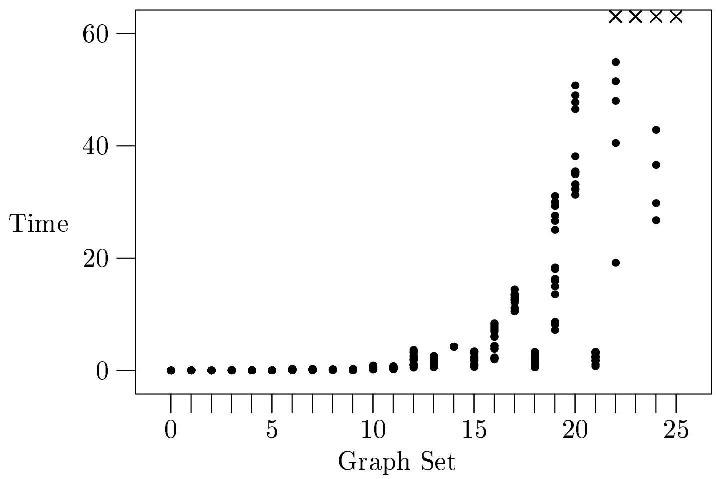
[2] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[3] A. Itai, M. Rodeh, and S.L. Tanimoto. Some matching problems for bipartite graphs. *J. ACM*, 25(4):517–525, 1978.

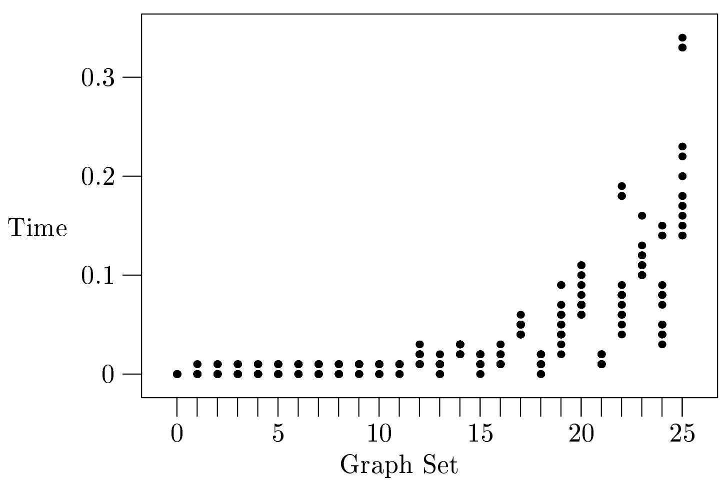
[4] T. Kohno. Experiments on an exhaustive search algorithm. University of Colorado Undergraduate Thesis, 1999.

[5] E.L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York, 1976.

[6] L. Lovász and M.D. Plummer. *Matching Theory*. North-Holland, Amsterdam, 1986.

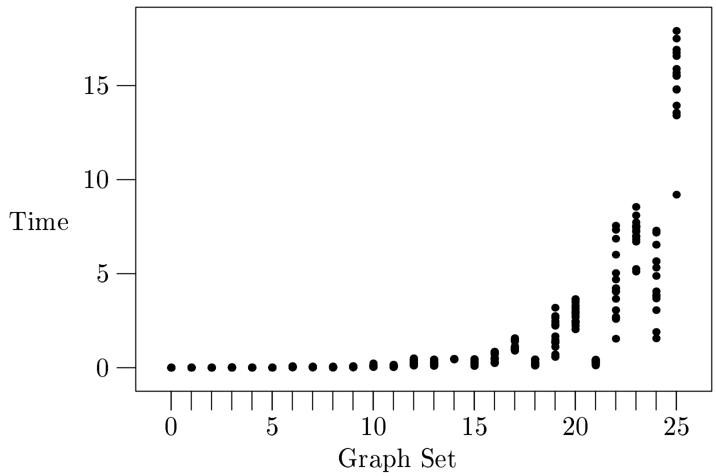


(a)

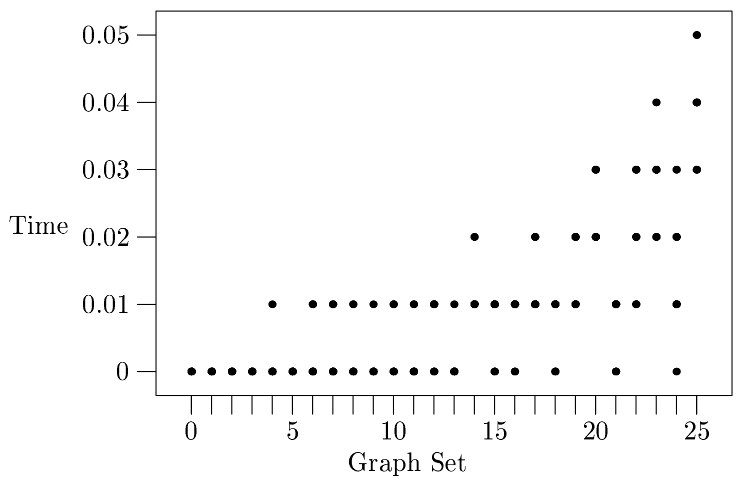


(b)

Fig. 7. Execution times for the basic algorithm on (a) *IAS* instances (b) *SAI* instances.

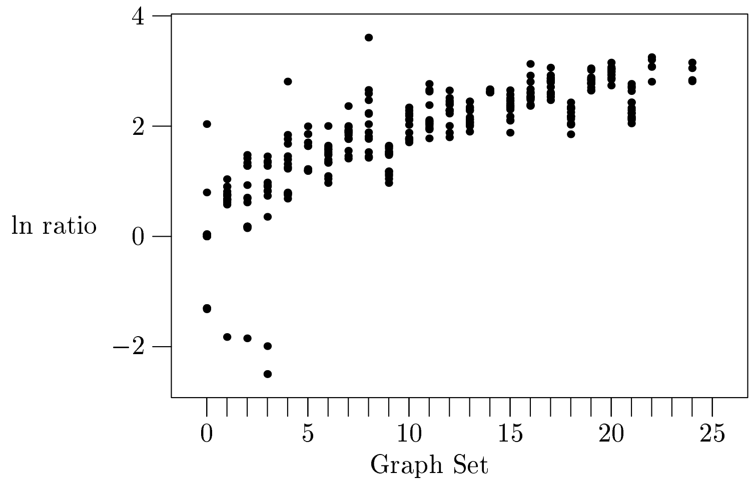


(a)

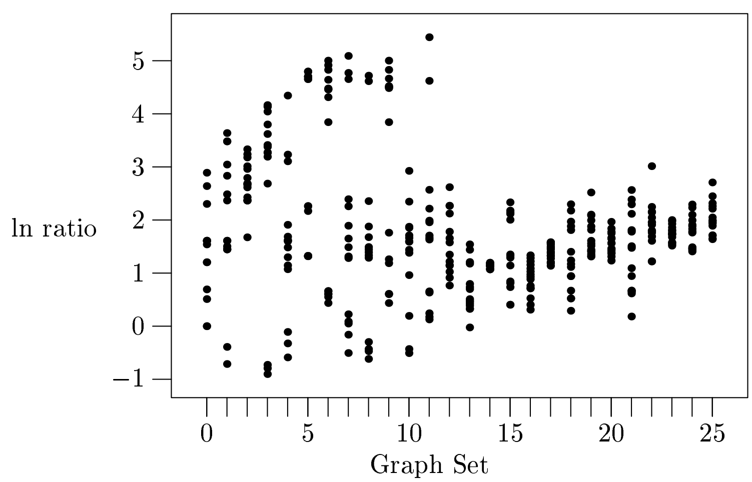


(b)

Fig. 8. Execution times for the matroid algorithm on (a) *IAS* instances (b) *SAI* instances.

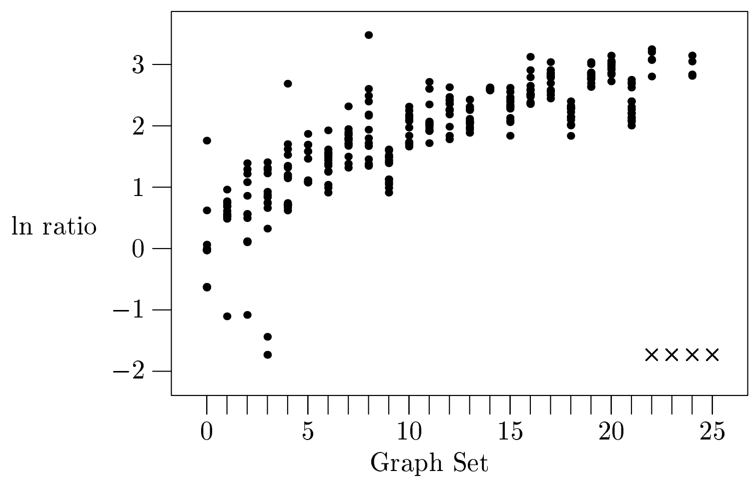


(a)

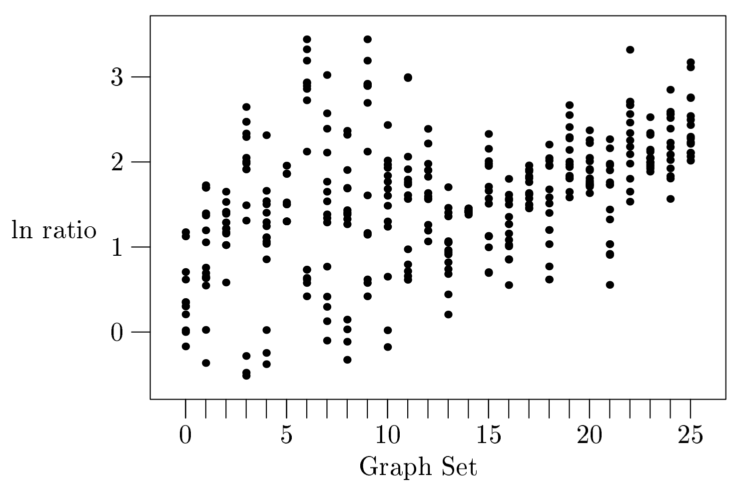


(b)

Fig. 9. Natural log of the ratio of the number of recursive calls required by the basic algorithm over the matroid algorithm, on (a) *IAS* instances (b) *SAI* instances.

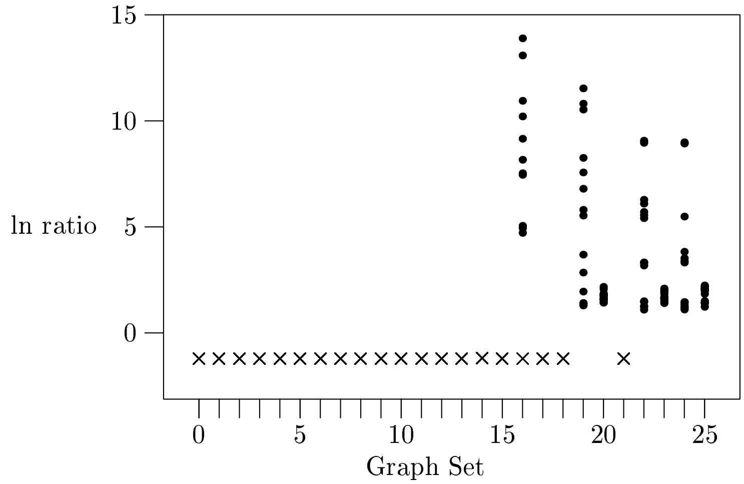


(a)



(b)

Fig. 10. Natural log of the ratio of the number of calls to `DFS` required by the basic algorithm over the matroid algorithm, on (a) *IAS* instances (b) *SAI* instances.



(a)

Fig. 11. Natural log of the ratio of the number of recursive calls required by the basic algorithm over the matroid algorithm on CL instances. All but one "×" is due to the basic algorithm alone being terminated.

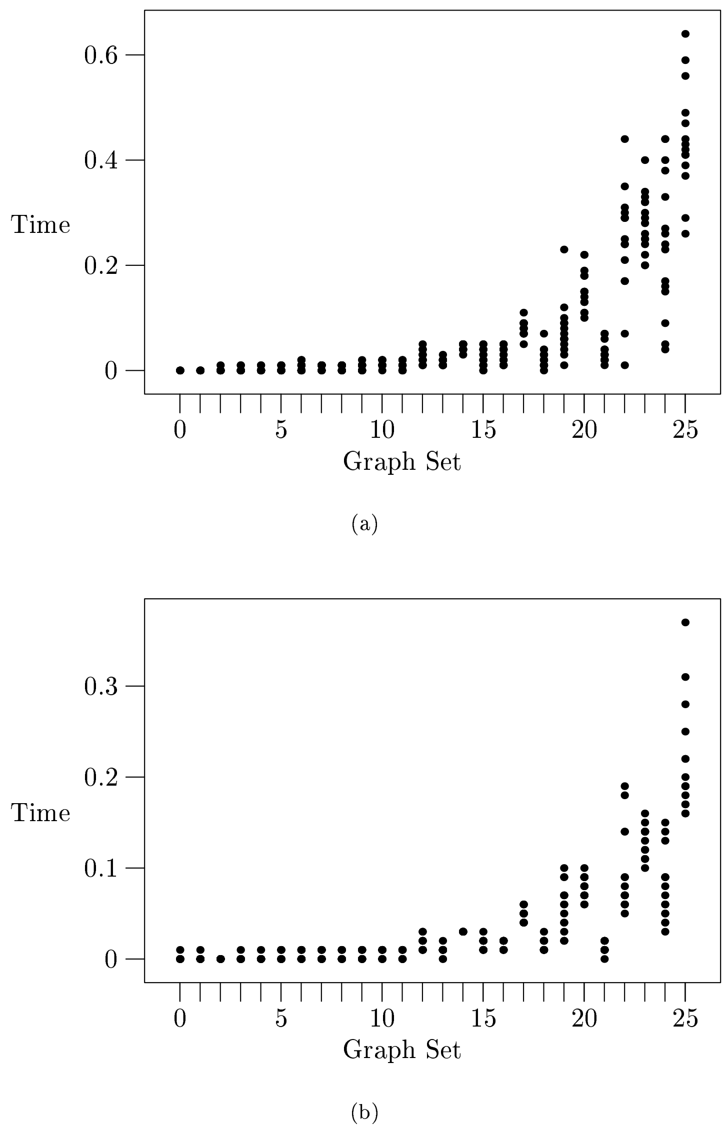
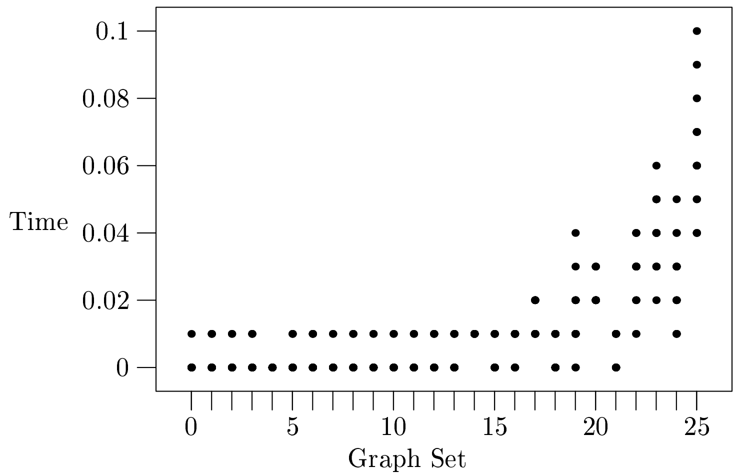
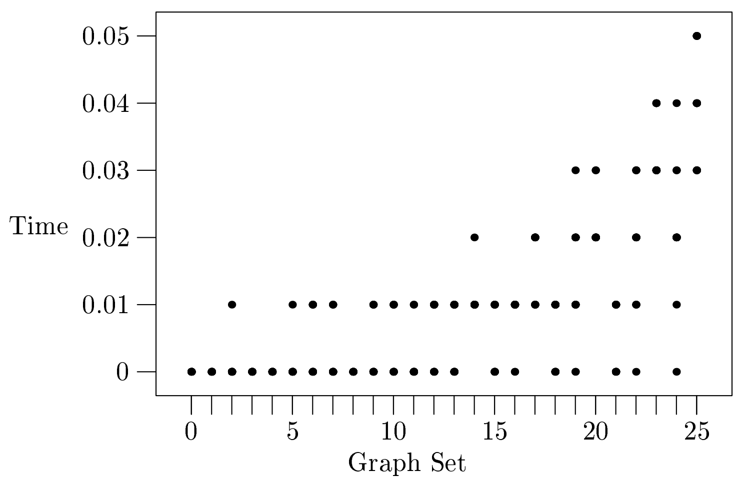


Fig. 12. Execution times for the basic MTF algorithm on (a) *IAS* instances (b) *SAI* instances.

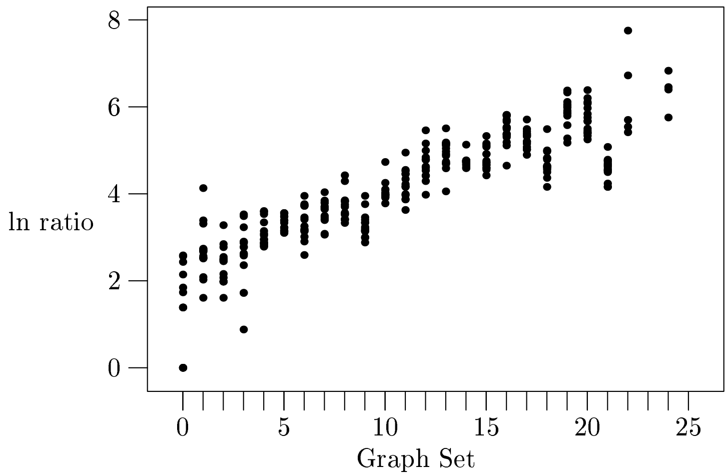


(a)

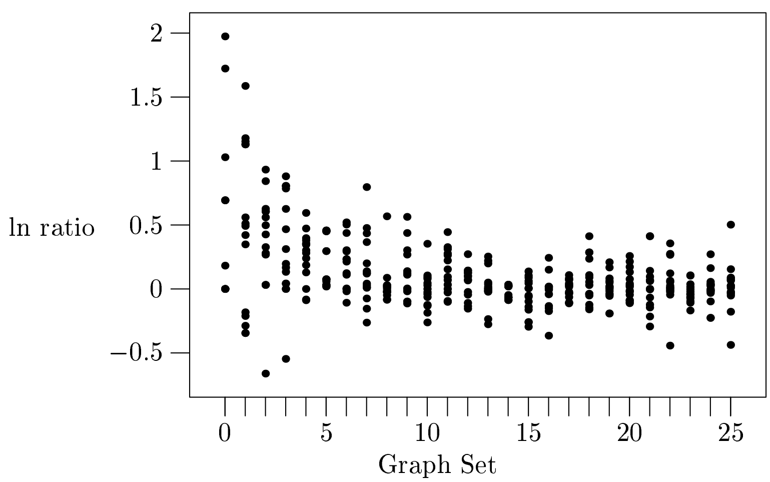


(b)

Fig. 13. Execution times for the matroid MTF algorithm on (a) *IAS* instances (b) *SAI* instances.

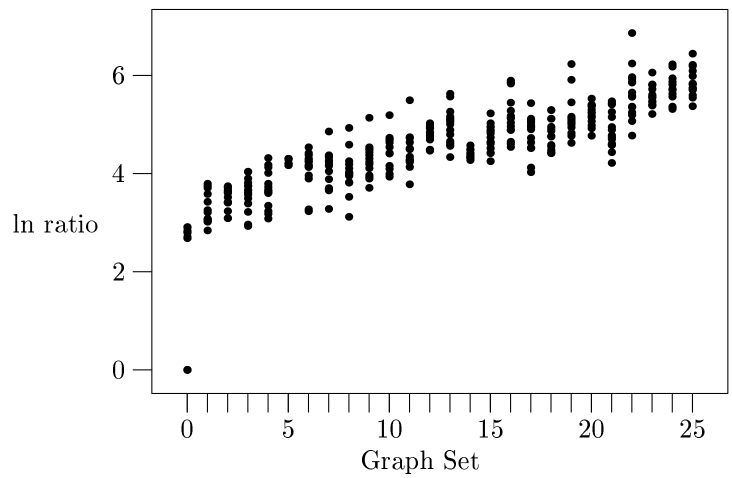


(a)

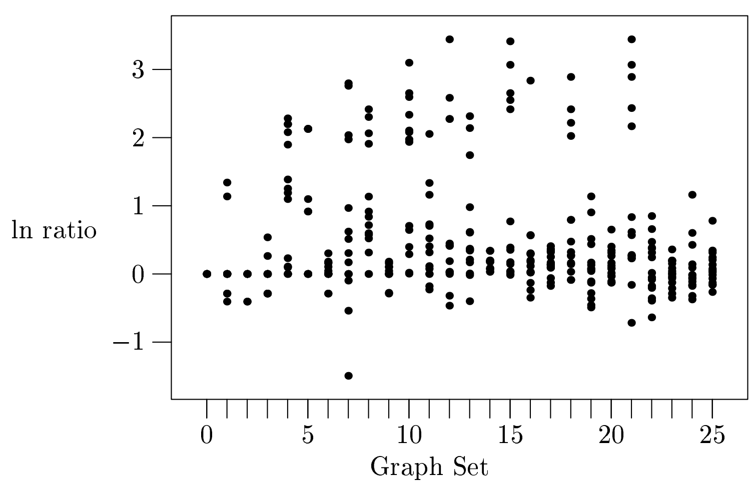


(b)

Fig. 14. Natural log of the ratio of the number of recursive calls required by the basic algorithm over the basic MTF algorithm on (a) *IAS* instances (b) *SAI* instances.

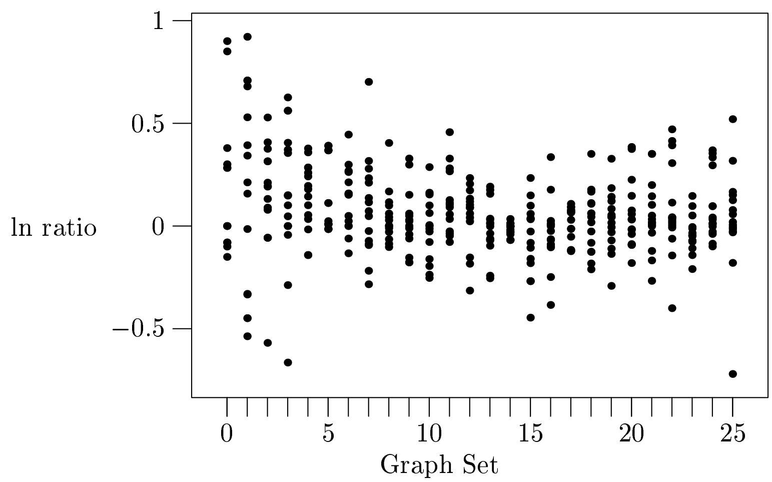


(a)

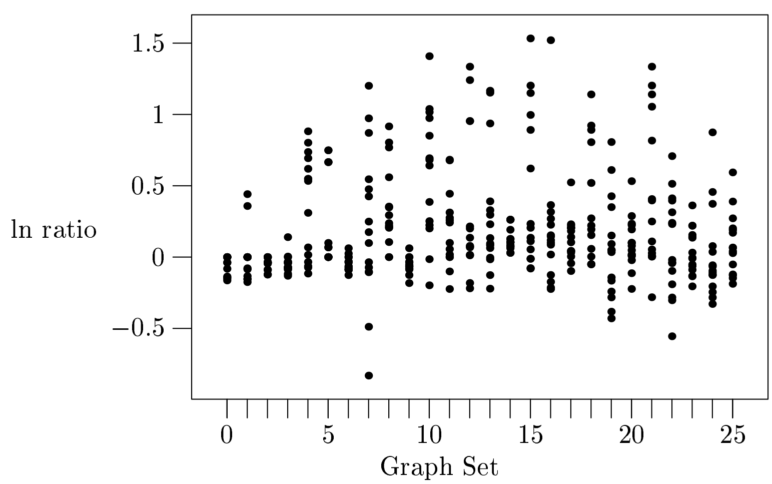


(b)

Fig. 15. Natural log of the ratio of the number of recursive calls required by the matroid algorithm over the matroid MTF algorithm on (a) *IAS* instances (b) *SAI* instances.

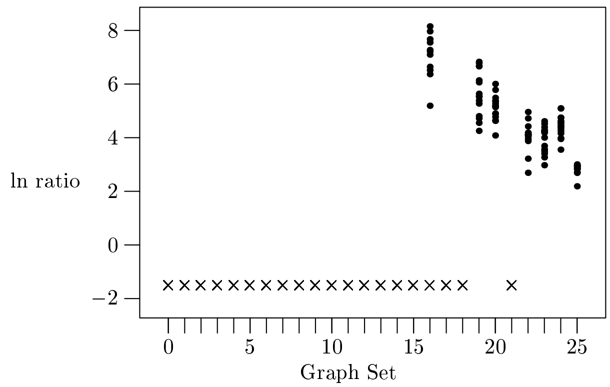


(a)

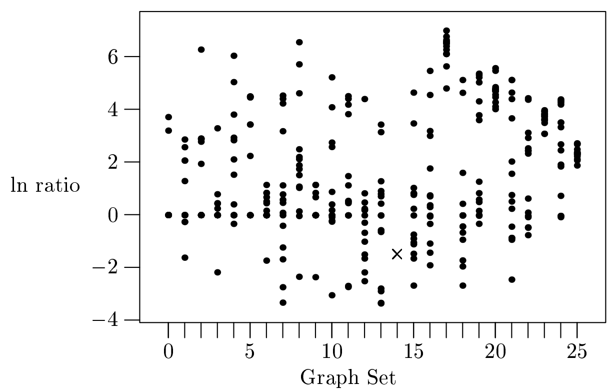


(b)

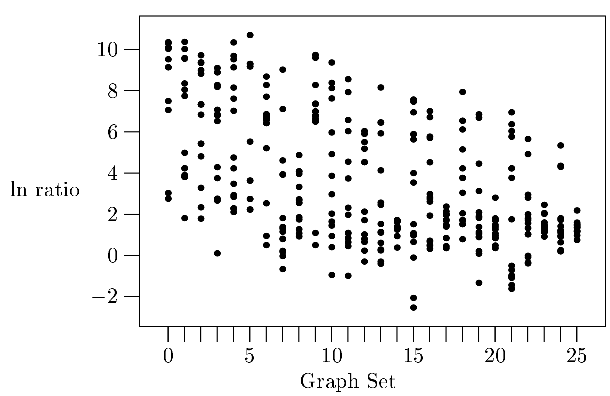
Fig. 16. Natural log of the ratio of the number of recursive calls on *SAI* instances required by (a) the basic algorithm over the basic MTF algorithm (b) the matroid algorithm over the matroid MTF algorithm.



(a)



(b)



(c)

Fig. 17. Natural log of the ratio of the number of recursive calls: (a) basic algorithm over basic MTF algorithm (b) matroid algorithm over matroid MTF algorithm (c) basic MTF algorithm over matroid MTF algorithm.