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

Harold N. Gabow
University of Colorado at Boulder
and
Tadayoshi Kohno
Counterpane Systems

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 attending teams at Denver General 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 General Hospital. This program has been used each 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 solution space efficiently. This paper describes the design of the program and the graph theoretic ideas involved; computational experience in generating real-world

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: Counterpane Systems, 101 E. Minnehaha Parkway, Minneapolis, MN 55419. e-mail: kohno@counterpane.com

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or permissions@acm.org.

schedules; and 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 braces, e.g., $I \cup e$ is the same as $I \cup \{e\}$.

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 $G = (X \cup Y, E)$, where the vertex set is partitioned into sets $X$ and $Y$ and each edge joins vertices of these two sets.

## 2. PROBLEM FORMULATIONS AND ALGORITHMS

### The real-world problem

The problem is to generate a yearly schedule for Department of Medicine attending teams at the Denver General 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. It is divided into about a dozen divisions (e.g., Cardiology, Pulmonary), the largest 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 teams had 5 doctors.) This means making rounds and managing the care of all patients on the medicine service. Attending is a large commitment of time and energy, and 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 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 attending more than one month have as many free months between attendings as possible because of the time commitment.

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).

The most important measure of quality of a schedule is time off between attending months. To make this precise we define the *clearance* of a schedule to be the minimum number of months off between two consecutive attendings of any given doctor. For instance a schedule has clearance at least 1 exactly when no doctor attends in two consecutive months.

### 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 $d(v, F) = f(v)$ [5]. 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 $E$, each subset of $\mathcal{C}$ called 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 a doctor must attend and also the size of a team in each month, 4. 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 $\{e, f\} \in \mathcal{C}$ where $e$ and $f$ are both incident to the same division and the same month. The clearance constraint corresponds to pairs $\{e, f\} \in \mathcal{C}$ where $e$ and $f$ are incident to the same doctor and their months differ by $c$ or less. The remaining constraints of $\mathcal{C}$ model the idiosyncratic constraints, if any.

The desired schedule is any nonconflicting $f$-factor with the largest possible clearance $c^*$. We find $c^*$ and the corresponding schedule by checking consecutive values of $c = 0, 1, \ldots$ for feasibility; $c^*$ is the largest feasible value.

### Basic algorithm

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

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

Start with any $F \in \mathcal{F}$ and assume $F$ has a conflict. Index the edges of $F$ as $e_i, i = 1, \ldots, r$ so that $F_k$ is a conflict, where in general we write $F_i = \{e_1, \ldots, e_i\}$. Any nonconflicting $f$-factor does not contain $F_k$ and so belongs to exactly one of the disjoint families

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

Process every such $\mathcal{F}_i$ as follows. 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 none 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 (e.g., going from $G$ to $G - F_i$). However in general we must test for conflicts $\mathcal{C}$ using the $f$-factor in the original graph (e.g., $F'$ corresponds to $F' \cup F_{i-1}$). It is clear that the recursive procedure we have sketched discovers a nonconflicting $f$-factor if 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 $v \neq x, y$ 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 $F'$, else no $F'$ exists.

When $G$ is bipartite, as in our scheduling problem, $f$-factors correspond to network flows. We find an augmenting path using the Ford-Fulkerson labelling method [1]. (For general graphs use Edmonds' algorithm [4].) 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 [2]. However this modification does not seem worth programming.)

### Discussion

Our problem involves a large 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 with an obvious 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 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.

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 $e_i = (x, y)$ with $x \in D, y \in M$ we search for an augmenting path from $y$.

### Implementation

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

We store the graph as a collection of adjacency lists for the doctors. This facilitates checking clearance constraints. We number the vertices so doctors in a given division are consecutive. This facilitates checking division constraints. All lists are sequential since we do not modify the graph.

The other constraints vary from year to year and so are not built in. We have always been able to handle these idiosyncratic constraints by simple modifications to the 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 $d, d'$ with $f(d) = f(d') = 1$, $d$ adjacent to the months in $D$'s preference list that differ from $m$ by more than $c$, and $d'$ adjacent to only $m$.

We generate conflict partitions (1) and associated edge-indexings in a way that dovetails with the graph data structure. Specifically all edges in the current $f$-factor $F$ are moved to the beginning of their adjacency lists. This induces an implicit indexing of the edges of $F$, since the vertices are already ordered. The edges of $F$ are scanned in order until an edge $e_k$ creating a division, clearance or idiosyncratic conflict is found. The corresponding partition (1) is used.

The advantage of this strategy is programming simplicity: The graph corresponding to any $\mathcal{F}_i$ of (1) (taking into account all recursive calls) consists of all the edges

that occur after $e_i$ in the adjacency structure. Thus no work is needed to construct the graph for recursive calls. (More details on this are given in Fig. 1 below.) A potential disadvantage of the strategy is that the partition of (1) has more sets than necessary. For example a clearance conflict comes from just two edges, so we could get $k = 2$ in (1). But our strategy takes the conflict set to be all the edges of $F$ up to the last edge of the conflict pair. Hence more subproblems are generated. (Specifically if the conflicting edges are $e_j$ and $e_k$, $j < k$, the first possibility gives families $\mathcal{F}[\emptyset,\ e_j]$ and $\mathcal{F}[e_j,\ e_k]$, while our strategy gives $\mathcal{F}[\{e_1, \ldots, e_{i-1}\}, e_i]$ for $1 \leq i \leq k$.) From a greedy point of view this is bad. However it is unclear what the global consequences of the strategy are (since a priori any of our families can be 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 be early in the list (i.e., lower numbered). As a simple example consider a doctor with only one feasible schedule at the clearance $c$ being tested. If this doctor is deep in the list he will generate unresolvable conflicts until, by luck or by exhaustive search, the attending months he needs have not been fixed to earlier doctors in the list. Thus such a doctor should be at the head of the list. The same holds for small sets of doctors that collectively generate similar conflicts. Following this intuition we list large divisions early, as well as divisions with highly constrained doctors. This ordering has been done by hand in a quick pass over the given input file.

```
boolean procedure DFS(d) {
  reached_doc[d] = true;
  for each unmatched edge (d, m) ∋ not reached_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 {
      reached_mth[m] = true;
      for matched edges (m, d') ∋ not reached_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 reached_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 reached_mth array marks months that should not be visited because they have been reached in the search.

The search for an augmenting path is written as a recursive depth-first search, for programming simplicity. See Fig. 1. The rest of the implementation follows mechanically from the above decisions. Even though our design was motivated to make creating subproblems easy, some bookkeeping is needed to reconstruct the graph after a recursive call.

### 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 small size encountered at Denver General 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.

### 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 [4] 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 two axioms that are motivated by the two most common matroids, the family of forests of a graph and finite collections of vectors in vector spaces. 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 if

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

We 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 has a laminar family $\mathcal{F}$. 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 $B$ has

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

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 $I$ is $\{f \in I : I \cup e - f \in \mathcal{I}\}$. For a laminar matroid the fundamental circuit of $e$ is $I \cap S$ for the minimal set $S \in \mathcal{F}$ that contains $e$ and has $|I \cap S| = f(S)$. (If there are lower bounds, assuming $I$ is a base the fundamental circuit is $I \cap S - \cup\{T \in \mathcal{F} : e \notin T,\ |I \cap T| = \ell(T)\}$ for $S$ defined 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 $\mathbf{M}_i$. 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, works by finding augmenting paths. The difference from standard augmenting paths of network flow is that two consecutive elements in the path, instead of being adjacent edges, are ground set elements, one belonging to the fundamental circuit of the other. Also an augmenting path cannot have a "shortcut."

### 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 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 matroid. We now illustrate how a matroid algorithm can capture more constraints than the basic algorithm.

Recall that 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.

It is possible to model division constraints in the graph itself, rather than in the constraint set $\mathcal{C}$. Instead of $f$-factors we can use network flows. We design the flow network to model the division constraints, using an extra node for each division in each month. We refrained from doing this because it enlarges the graph and complicates the code, e.g., the graph is no longer bipartite.

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 with 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 $\mathbf{M}_2$. The laminar family has a set $S_m$ containing every edge incident to $m$, for each month $m = 1, \ldots, 12$. Each such $S_m$ is partitioned into sets $S_{m,j}$ for divisions $j$, where $S_{m,j}$ contains every edge incident to month $m$ and to a doctor in division $j$. The bounding function has $f(S_m) = 4$, $f(S_{m,j}) = 1$. Any valid matroid intersection automatically satisfies the division constraint. We use the matroid algorithm to find a maximum cardinality nonconflicting intersection.

The main difference between this matroid algorithm and the basic algorithm is 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, an augmenting path containing $e, f$ as consecutive elements has $f$ in the fundamental circuit of $e$, for the current matroid intersection $I$. For $\mathbf{M}_1$ this means $e$ and $f$ are adjacent, as in an ordinary augmenting path. For $\mathbf{M}_2$ let $e = (d, m)$. If $I$ contains

```
if some matched edge (m, d') has d' in d's division then {
  if not reached_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 after the "advance" comment, as the first statement of the **for** loop, in Fig. 1. The reached_mth array ensures the augmenting path is shortcut-free, but still allows the path to visit a month up to 4 times.

an edge $g = (d', m)$ where $d'$ is in the same division as $d$, then we must have $f = g$. In the opposite case $f$ is essentially chosen as usual: If $I$ does not contain 4 edges incident to $m$ then an augmenting path has been found. If $I$ contains 4 edges incident to $m$ then $f$ can be any of these edges.

The advantage of the 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 and the 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 is shortcut-free.

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.

### 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 no feasible schedule. In typical instances the basic algorithm ran over 20 minutes and the matroid algorithm was under 4 minutes (17M recursive calls versus 3.3M). 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 ordinary constraints continued to hold, ignoring whether a doctor was scheduled as normal or backup.

To get a quick solution we treated backup doctors as contributing another constraint to $\mathcal{C}$. We added a small amount of code to check that constraint (i.e., each month had 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 and 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 be 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 ideas in a purer setting. 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 models the central constraint of the doctor scheduling problem.

### NP-completeness

Itai et. al. [2] 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 matching problem corresponding to a given instance of SAT is illustrated in Fig. 3. The item nodes of the graph 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 clauses.

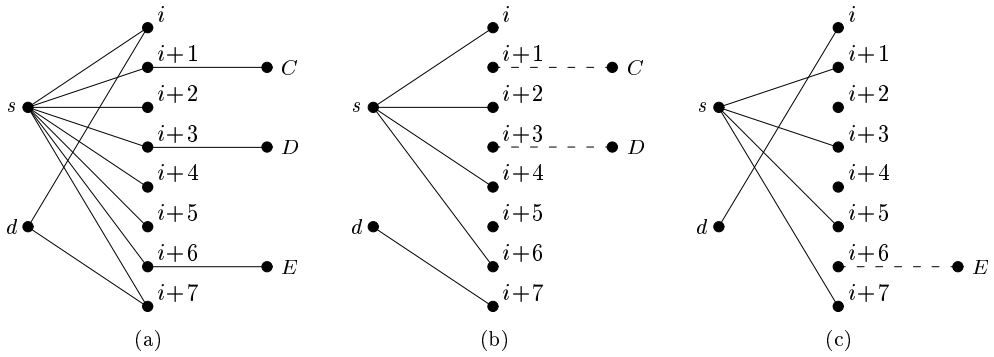


Fig. 3. (a) Edges in $G$ corresponding to variable $x$, where $x$ occurs in clauses $C$ and $D$, $\overline{x}$ occurs in $E$. Time slots $i$ through $i+7$ correspond to $x$. (b) Matched edges (solid) and potentially matched edges (dashed) when $x = T$. (c) The same for $x = F$. In the entire graph, one dashed edge for each clause gets matched.

The edges corresponding to a typical variable are illustrated in Fig. 3(a). A variable $x$ that, together with its complement, occurs in a total of $t$ clauses corresponds to $2t+2$ consecutive time slot nodes. These time slots are adjacent to the selector $s$ of $x$ and to no other selector. The first and last of these time slots are adjacent to the decision node $d$ of $x$ and to no other decision node. Each clause node is adjacent to one time slot for each literal in the clause. The bounding function has $f(v) = 1$ for $v \neq s$ and $f(s) = t+1$ (for selectors $s$).

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 $d$ uniquely determines the matched edges incident to the corresponding selector $s$. Fig. 3(b)–(c) illustrate the matching corresponding to a satisfiable formula. □

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. It has $f(p) = m - c$ for $c$ the number of clauses.

## Algorithms

We implemented the basic algorithm and a matroid algorithm for the general clearance $c$ matching problem. We now describe the matroids in the latter.

Both matroids have ground set $E$. Matroid $\mathbf{M}_1$ models the degree constraints of the time slots (i.e., it is a partition matroid with a block for each time slot $i$, with bound $f(i)$). $\mathbf{M}_2$ 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 a way that each $S_{i,j}$ consists of pairwise conflicting edges, i.e., $(i,s)$, $(i,s') \in S_{i,j}$ implies $|s - s'| \leq c$. The bounding function has $f(S_i) = f(i)$ and $f(S_{i,j}) = 1$. Our program constructs the $S_{i,j}$ greedily: It repeatedly chooses 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 when two conflicting edges belong to different subsets $S_{i,j}$. However intuitively it is clear that this model prunes the solution space better than the $f$-factor model.

## 4. EXPERIMENTS

We conducted a number of experiments on the programs for clearance matching. All tests were run on a Pentium 200MHz MMX computer with 32 megabytes of RAM under the Linux kernel 2.0.30. All algorithms were implemented by one person, in C, using the same coding style. Our main measurements were user time and number of recursive calls. The latter equals the total number of subproblems generated. Full details of the experiments are in [3]; selected results are presented here.

### Test families

The first set of test families was obtained using our NP-hardness reduction. Generators from the Second DIMACS Implementation Challenge (`ftp://dimacs.rutgers.edu/pub/challenge`) were used to get difficult satisfiability problems. These problems were transformed into complete and perfect clearance 1 matching problems using the reductions of Section 3. We experimented with SAT instances having 5 to 10 variables and clauses numbering in the double-digits or low hundreds. An instance with 10 variables and 134 clauses transformed to a graph with 155 items, 752 slots and 1906 edges; an instance with 15 variables and 50 clauses transformed to a graph with 81 items, 328 slots and 806 edges.

We also studied a simple family of random instances. The family is parameterized by the number of items, number of time slots, degree constraint function $f$, and a lower bound on the number of edges. A random instance is chosen by first generating $f(i)$ random edges for each item $i$; then generating a minimal number of random edges to ensure each slot $t$ has at least $f(t)$ edges; and finally generating more random edges to ensure the lower bound is met. We took $f(t)$ constant on time slots $t$, usually $f(t) = 1$.

### Random instances

Both algorithms were efficient on random graphs with a perfect c-matching. On 251 such graphs the mean time was less than 2 (1) seconds for the basic (matroid) algorithm. A small number of graphs took disproportionate amounts of time, but usually the matroid solver found a c-matching quicker.

Of 54 random graphs with no perfect c-matching, all but 7 were immediately rejected by the basic algorithm because there was no $f$-factor. All 54 were immediately rejected by the matroid algorithm because there was no perfect matroid intersection.

### Orderings in SAT instances

As mentioned in Section 2 the ordering of the vertices in our data structure effects the efficiency of the search. For clearance matching the algorithm always finds a conflict involving the lowest numbered item with a clearance conflict. We studied

a family of orderings for problems obtained from SAT instances. We now describe the family.

In all orderings the clause nodes $c$ and the padding node $p$ (if present) have the highest numbers, since their matched edges are determined by the decision nodes $d$ and selectors $s$. The order for the $d$ and $s$ nodes is described by a name that has the following form:

$$\{ds, sd, DS, SD\}[-, +][\text{_nop}]$$

The first part of an order name is one of the 4 strings listed in curly braces. $ds$ means for each SAT variable the decision node immediately precedes the selector node. $DS$ means the ordering starts with the decision nodes; they are followed by the selector nodes, in the same order. $sd$ and $SD$ are defined analogously.

If the middle part of the name is $-$ ($+$) the variables are ordered by nonincreasing (nondecreasing) degree. If the middle part is omitted the variables are ordered as in the given SAT instance.

If the final part of the name is _nop the reduction does not use a padding node; the default is that a padding node is used.

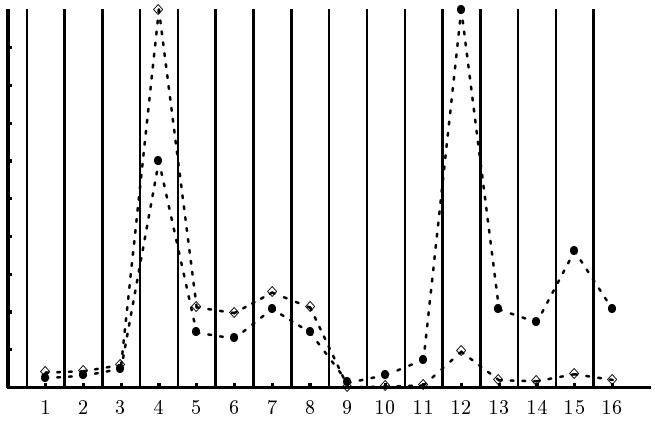


Fig. 4. Plot of the user time ($\diamond$) and number of recursive calls ($\bullet$) the matroid algorithm used when solving clearance matching instances reduced from a satisfiable Boolean formula with 5 variables and 34 clauses. The maximum displayed user time is 54.24 seconds and the maximum number of recursive calls needed was 65439. 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.

By eyeballing plots like Fig. 4 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 1. To explain this observation suppose the selector node $s_i$ for the $i$th variable is un-c-matchable. In the $ds$ ordering we must only backup our fixed set of matched edges to the item node directly in front of $s_i$ (i.e., the corresponding decision node $d_i$) and try again. However in $DS$ to toggle the matched edge for $d_i$ we must backup over all the matched edges for selector nodes $s_j$, $j < i$ and decision nodes $d_k$, $k > i$. This leads to many more futile subproblems.

| 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 1. Statistics of the ratios of the time required by the basic algorithm on satisfiable Boolean formulas. The time for $DS$ is in the numerator.

| 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 2. 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 time for $ds$ is in the numerator.

We also observed groups of orderings with essentially the same performance on unsatisfiable instances, e.g., $ds$, $sd$, and $SD$. See Table 2. A possible intuitive explanation is that the $d$'s serve little purpose other than ensuring that the $s$'s can match to slots in only one of two configurations. These orders give essentially the same ordering of the selectors and so are equivalent.

### Basic versus matroid algorithm

Since unsatisfiable Boolean formulas require searching the whole space we expect the matroid algorithm to perform at its best on them. The time advantage observed was 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.

### 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.

Consider an invocation of the basic algorithm on the family $\mathcal{F}_i$ of (1). Recall that it finds a new $f$-factor $F' \cup F_{i-1}$ not containing $e_i$. Index the edges of $F'$ as $f_j$, $j = i, \ldots, r$. Let $k$ be the smallest index such that $F_{i-1} \cup \{f_i, \ldots, f_k\}$ contains a conflict. Intuitively the item node $x$ that is an endpoint of edge $f_k$ may be the source of the conflict. That is, given that edges $F_{i-1}$ are matched it may be impossible to

match edges incident to $x$ without violating the clearance constraint. To investigate this possibility immediately, we swap $x$ and another item node to make $x$ and $f_k$ come right after the frozen edges $F_{i-1}$. If the guess is correct we avoid generating the subproblems for edges $f_j$, $j < k$ that the basic algorithm generates. We call this modification the *move-to-front* heuristic.

We tested move-to-front versions of the basic and matroid algorithms on our random instances. Both algorithms generally found solutions faster (and with fewer recursive calls). As an example of the possible extreme benefits we mention a c-matchable instance 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 unsatisfiable Boolean formulas using the move-to-front matroid algorithm, the $DS$ ordering was no longer worse than the others. (On 85 instances the time ratio was 1.03 for $ds$ and similar for the others.) 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.

On random instances with all slot nodes $t$ having $f(t) = 4$, the move-to-front matroid algorithm was best.

## 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. An interesting question for future work is to characterize the small number of data sets that violate our general conclusions, e.g., when is matroid or move-to-front *less* efficient?

### Acknowledgment

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

### REFERENCES

[1] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows.* Prentice Hall, Englewood Cliffs, New Jersey, 1993.

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

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

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

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