Conflict graphs in integer programming *

Alper Atamtürk
George L. Nemhauser
Martin W.P. Savelsbergh
School of Industrial and Systems Engineering
Georgia Institute of Technology

March 25, 1998

Abstract

We report on the use of conflict graphs in solving integer programs. A conflict graph represents logical relations between binary variables appearing in an integer program. We develop algorithms and data structures that allow the effective and efficient construction, management, and use of dynamically changing conflict graphs. Our computational experiments show that the techniques presented work very well in practice.
Keywords: Conflict graphs, integer programming, cliques, preprocessing.

1 Introduction

We report on our investigation of the use of conflict graphs in solving integer programs. A conflict graph represents logical relations between binary variables appearing in an integer program. More precisely, a conflict graph has a vertex for each binary variable and its complement, and an edge between two vertices when at most one of the variables represented by the vertices can equal one in a solution.

Conflict graphs are typically constructed using probing techniques based on feasibility considerations. By tentatively setting a binary variable to one of its bounds and examining whether this causes other binary variables to be fixed, we derive logical relations of the form 'if $x_i = 1$, then $x_j = 0$'. Such a logical relation can be represented in the conflict graph by an edge between the vertex associated with x_i and the vertex associated with x_j.

We construct an extended conflict graph by also using probing techniques based on optimality considerations. By computing an upper bound on the optimal solution when two variables are tentatively set to one of their bounds and examining whether this upper bound is smaller than a known lower bound on the optimal solution to the original integer program, we derive logical relations of the form 'no optimal solution can have both $x_i = 1$

*This research is supported, in part, by NSF Grant DMI-9700285 to the Georgia Institute of Technology.

and $x_j = 1$'. Such a logical relation can again be represented in the conflict graph by an edge between the vertex associated with x_i and the vertex associated with x_j.

The primary use of conflict graphs is in the generation of clique inequalities. Any feasible solution to the integer program defines a vertex packing in the conflict graph, i.e., a subset of vertices no two of which are adjacent. As a consequence, the vertex packing polytope associated with a conflict graph forms a relaxation of the convex hull of feasible solutions to the original integer program. Hence, valid inequalities for the vertex packing polytope, such as clique inequalities, are also valid inequalities for the integer program. A clique inequality ensures that at most one vertex in a clique of the conflict graph, i.e., a subgraph in which every pair of vertices is adjacent, is selected in any vertex packing. Conflict graphs may also be used to enhance preprocessing, cover inequality generation, primal heuristics, and branching schemes.

A major computational obstacle to the use of conflict graphs is their size, i.e., the number of vertices and edges. Conflict graphs can be huge. Consider, for example, an instance of a set partitioning problem. Each row of the constraint matrix defines a clique in the conflict graph, which results in a very dense conflict graph. Consequently, it is often impractical (or even impossible) to store the conflict graph explicitly, even for moderately sized instances. As a result, most existing solvers cannot generate clique inequalities for problems with a fair number of generalized upper bound (GUB) constraints, i.e., constraints of the form $\sum x_j \leq 1$, or with a set partitioning substructure. Ironically, clique inequalities are most useful for such problems! We show that with appropriate data structures, it is possible to maintain and manipulate dense conflict graphs efficiently.

Storage is not the only computational issue. If one is not careful, the construction of the conflict graph may also be computationally prohibitive. We discuss various implementation techniques to construct the conflict graph efficiently.

Since each node in the search tree of an LP based branch-and-bound algorithm has its own associated conflict graph, we develop algorithms and data structures that allow the effective and efficient construction, management, and use of dynamically changing conflict graphs throughout the search tree.

Although we introduce several new ideas related to the construction and use of conflict graphs, the emphasis of this paper is on computation. We show that, when carefully implemented, conflict graphs can provide a powerful tool in the solution of a variety of integer programs. Our computational experiments show that the ideas presented in this paper can be incorporated successfully in a general purpose optimizer and that our implementation outperforms state-of-the-art commercial solvers on a wide range of instances.

In the remainder, for ease of presentation, we only consider pure 0-1 integer programs. However, all the techniques described can be applied to any integer program containing 0-1 variables.

In Section 2, we formally introduce conflict graphs and show how to construct them using probing based on feasibility and optimality considerations. In Section 3, we discuss the storage and management of dynamically changing conflict graphs. In Sections 4 and 5, we discuss the use of conflict graphs in preprocessing and cut generation. In Section 6, we

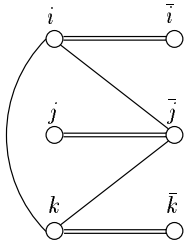

Figure 1: Conflict graph

present the results of our computational experiments. Finally, in Section 7, we conclude with some generalizations currently under investigation.

2 Conflict graphs

A conflict graph represents logical relations between binary variables. It has a vertex for each binary variable and its complement, and an edge between two vertices when at most one of the variables represented by the vertices can equal one in an optimal solution. The four possible logical relations between two binary variables are

$$\begin{array}{lrcrrl}
x_i = 1 \Rightarrow x_j = 0 & \Leftrightarrow & x_i & & +x_j & \leq 1, \\
x_i = 0 \Rightarrow x_j = 0 & \Leftrightarrow & (1 - x_i) & & +x_j & \leq 1, \\
x_i = 1 \Rightarrow x_j = 1 & \Leftrightarrow & x_i & +(1 - x_j) & & \leq 1, \\
x_i = 0 \Rightarrow x_j = 1 & \Leftrightarrow & (1 - x_i) & +(1 - x_j) & & \leq 1.
\end{array}$$

The inequalities represented by the edges of the conflict graph are called the *edge inequalities*. Figure 1 shows the conflict graph representing the three edge inequalities

$$\begin{array}{rrrl}
x_i & +(1 - x_j) & & \leq 1, \\
x_i & & +x_k & \leq 1, \\
& (1 - x_j) & +x_k & \leq 1.
\end{array}$$

Obviously, there is an edge between a variable and its complement, since at most one of them can equal one in any feasible solution. However, the relation between a variable and its complement is stronger: exactly one of them must be one in any feasible solution. This type of relation is shown in the conflict graph by a double line.

Given a 0-1 integer program, we construct a conflict graph using probing techniques based on feasibility and optimality considerations. In the next two subsections, we explain these techniques in more detail.

2.1 Implications from feasibility conditions

Probing refers to setting a binary variable to one of its bounds tentatively and examining the consequences [3, 11]. When we tentatively set a binary variable to one of its bounds

and a subsequent analysis shows that the problem has become infeasible, then we can fix that variable to its opposite bound. Formally, let

$$S = \{x \in B^n : \quad Ax \leq b\} \quad \text{and} \quad S_{x_i=v} = \{x \in S : \quad x_i = v\}$$

for $v \in \{0,1\}$. If $S_{x_i=1} = \emptyset$, then $x_i \leq 0$ is valid for S; similarly if $S_{x_i=0} = \emptyset$, then $x_i \geq 1$ is valid for S. In either case, we have fixed x_i.

Determining whether $S_{x_i=v} = \emptyset$ may be as hard as solving the original integer program. Therefore, we resort to easily computable relaxations $S'_{x_i=v} \supseteq S_{x_i=v}$ and try to show that $S'_{x_i=v} = \emptyset$. Let

$$L^r_{x_i=v} = \min\{a_r x \ : \ x \in S'_{x_i=v}\}$$

where a_r is the r^{th} row of A, i.e., $L^r_{x_i=v}$ is the minimum value of the left-hand side of the rth row over all solutions $x \in S'_{x_i=v}$. If $L^r_{x_i=v} > b_r$, then $x_i \leq 0$ is valid for S when $v = 1$, and $x_i \geq 1$ is valid for S when $v = 0$. For an equality constraint the same arguments hold also when $U^r_{x_i=v} < b_r$, where $U^r_{x_i=v} = \max\{a_r x \ : \ x \in S'_{x_i=v}\}$.

Probing can be used to derive edges of the conflict graph by setting each of two variables to one of their bounds. Let $S_{x_i=v_i,x_j=v_j} = \{x \in S : x_i = v_i, x_j = v_j\}$. Then

- if $S_{x_i=1,x_j=1} = \emptyset$ then, $x_i + x_j \leq 1$ is valid for S,
- if $S_{x_i=1,x_j=0} = \emptyset$ then, $x_i + (1 - x_j) \leq 1$ is valid for S,
- if $S_{x_i=0,x_j=1} = \emptyset$ then, $(1 - x_i) + x_j \leq 1$ is valid for S,
- if $S_{x_i=0,x_j=0} = \emptyset$ then, $(1 - x_i) + (1 - x_j) \leq 1$ is valid for S.

The choice of the relaxation $S'_{x_i=v_i,x_j=v_j}$ for $S_{x_i=v_i,x_j=v_j}$ is critical. The stronger relaxation is likely to yield more edges, but also it may be harder to compute. Thus, we face a tradeoff between the size of the conflict graph and the computational effort required to construct it.

One possible relaxation is to use just the bounds on the variables. For constraint r, define B^+_r (B^-_r) to be the index set of variables with positive (negative) coefficients. Then we have

$$L^r_{x_i=v_i,x_j=v_j} = \sum_{k \in B^-_r, k \neq i, k \neq j} a_{rk} + a_{ri} v_i + a_{rj} v_j.$$

A stronger relaxation is obtained by using the bounds of variables as well as the current conflict graph. Let $G = (V, E)$ denote the current conflict graph and consider the set of vertices $U \subseteq V$ representing variables in B^-_r and complements of variables in B^+_r. Let $w_k = |a_{rk}|$ for all $k \in U$. Now, a lower bound L^r, for the minimum value of the left-hand side of the rth row is obtained by solving the weighted vertex packing problem

$$(WVP) \qquad L^r = \sum_{k \in B^+_r} a_{rk} - \max\{wz : z \text{ vertex packing of } G(U)\}.$$

The summation term $\sum_{k \in B_r^+} a_{rk}$ is added to the bound since we use the complements for B_r^+. The weighted vertex packing problem is NP-hard [8]. Therefore, in order to get a lower bound on L^r, we solve a relaxation of the vertex packing problem. The relaxation is based on the observation that if a graph consists of a set of disjoint complete subgraphs, the optimal vertex packing is obtained by picking a largest weight vertex from each complete subgraph. Therefore, we partition the vertices of $G(U)$ into a set of cliques and pick a largest weight vertex from each clique. Algorithm 1 gives a greedy algorithm that implements this idea.

Algorithm 1 Greedy Clique Partitioning

```
1: Index vertices in U so that w_1 ≥ w_2 ≥ ... ≥ w_|U|.
2: Mark vertices in V \ U.
3: for j = 1 to |U| do
4:    if j is not marked then
5:       W ← W + w_j
6:       mark all vertices of a maximal clique containing j.
7:    end if
8: end for
9: L^r ← Σ_{k∈B_r^+} a_rk − W.
```

Example Let

$$4x_1 \ -3x_2 \ +5x_3 \ -3x_4 \ -3x_5 \ +2x_6 \leq 1$$

be the rth row of an IP. Then $B_r^+ = \{1,3,6\}$ and $B_r^- = \{2,4,5\}$. Furthermore, suppose the current conflict graph is defined by the solid edges of the graph in Figure 2. The minimum value of the left-hand side of row r is 1 (by setting $(x_1,\ldots,x_6) = (1,1,0,0,0,0)$ or equivalently by selecting packing $\{2,\bar{3},\bar{6}\}$). Using the greedy clique partitioning algorithm, we obtain the clique partitioning $\{\{\bar{1},2\},\{\bar{3},4,5\},\{\bar{6}\}\}$, which gives a lower bound of 0 on the value of the left-hand side of row r ($(x_1,\ldots,x_6) = (0,0,0,0,0,0)$ or packing $\{\bar{1},\bar{3},\bar{6}\}$).

Once we have a clique partitioning, it is easy to find a lower bound $L^r_{x_i=v_i,x_j=v_j}$. All we have to do is to pick the largest weight vertex in each clique subject to $x_i = v_i, x_j = v_j$. For example $L^r_{x_3=1,x_6=1} = 4 > 1$ given by packing $\{\bar{1},4\}$. Similarly, $L^r_{x_4=1,x_6=1} = L^r_{x_5=1,x_6=1} = 4 > 1$. Thus $S_{x_3=1,x_6=1} = S_{x_4=1,x_6=1} = S_{x_5=1,x_6=1} = \emptyset$. Hence, we can augment the conflict graph edges (3,6), (4,6), and (5,6) (dashed edges in Figure 2).

2.2 Implications from optimality conditions

The edge inequalities obtained in the previous section are derived from feasibility conditions and hence are valid for S. Using the objective function, it is possible to derive stronger inequalities that are not necessarily valid for S but are valid for optimal points in S, i.e., $S^{opt} = \{x \in S \ : \ cx = z^*\}$, where $z^* = \max\{cx : x \in S\}$. Let z_h be the value of any feasible

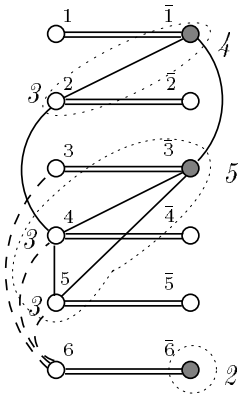

Figure 2: Clique partitioning for L^r

solution. For instance, if $z^*_{x_i=1,x_j=0} = \max\{cx : x \in S_{x_i=1,x_j=0}\} < z_h$ then $x_i + (1 - x_j) \leq 1$ is valid for S^{opt}.

Since it is nontrivial to compute $z^*_{x_i=1,x_j=0}$, we consider approximating it. Let $z_{x_i=1,x_j=0}$ denote the optimal value of the linear programming (LP) relaxation of $\max\{cx : x \in S_{x_i=1,x_j=0}\}$. If $z_{x_i=1,x_j=0} < z_h$, then $x_i + (1 - x_j) \leq 1$ is valid for S^{opt}. Note that if $z_{x_i=1,x_j=0} = -\infty$, then $x_i + (1 - x_j) \leq 1$ is valid even for S. We refer to solving the LP relaxation of the problem subject to a pair of variables fixed at one of their bounds and examining the consequences as *LP probing*. However, finding edge inequalities using LP probing requires the solution of many linear programs. In order to reduce the computational effort, we consider approximations that provide upper bounds on $z_{x_i=v_i,x_j=v_j}$.

Let $\bar{x}$ be an optimal solution to the LP relaxation of the original problem with objective value z. If $\bar{x}_i = v_i, \bar{x}_j = v_j$ for $v_i, v_j \in \{0, 1\}$, then $z_{x_i=v_i,x_j=v_j} = z$. Hence, we do not need to consider probing pairs that are at one of their bounds at these values. One approximation can be obtained by fixing only one variable at a bound and using the reduced cost of the second one. Let $z_{x_i=1}$ ($z_{x_i=0}$) be the value of an optimal solution $\hat{x}$ to LP subject to $x_i = 1$ ($x_i = 0$) and z_h be the value of a heuristic solution. Then,

- if $\hat{x}_j = 0$ and $z_{x_i=1} - \hat{c}_j < z_h$ then $x_i + x_j \leq 1$ is valid for S^{opt},
- if $\hat{x}_j = 1$ and $z_{x_i=1} + \hat{c}_j < z_h$ then $x_i + (1 - x_j) \leq 1$ is valid for S^{opt},
- if $\hat{x}_j = 0$ and $z_{x_i=0} - \hat{c}_j < z_h$ then $(1 - x_i) + x_j \leq 1$ is valid for S^{opt},
- if $\hat{x}_j = 1$ and $z_{x_i=0} + \hat{c}_j < z_h$ then $(1 - x_i) + (1 - x_j) \leq 1$ is valid for S^{opt}.

Note that in this method each LP instance differs by two bounds and solving these problems with the dual simplex is much easier than solving the initial LP. Nevertheless, one needs to solve n plus the number of fractional variables LPs, which may be prohibitive if many dual pivots are needed to solve each problem. Therefore we suggest yet another approximation which does not require solving any linear programs but makes use of the

optimal basis of the initial LP. These approximations are equivalent to performing a single dual simplex pivot; however no time consuming basis change is performed. Let B the index set of an optimal basis for LP, N be the index set of nonbasic variables. Also let c_B, c_N, x_B, x_N be the corresponding partitioning of c and x into basic and nonbasic variables. We are interested in the change of the optimal value when nonbasic variables x_i and x_j are set to v_i and v_j, respectively. Letting $t_i = v_i - \bar{x}_i, t_j = v_j - \bar{x}_j$, we consider

$$\Delta z = c_B \Delta x_B + c_N \Delta x_N = \bar{c}_i t_i + \bar{c}_j t_j.$$

Since dual feasibility is maintained in the presence of the additional equalities $x_i = v_i$, $x_j = v_j$, but primal feasibility may be lost, Δz gives a lower bound on the change. Hence we have

$$z_{x_i=v_i, x_j=v_j} \leq z + \bar{c}_i t_i + \bar{c}_j t_j \leq z.$$

It is possible to obtain a stronger upper bound by performing a ratio test to calculate the change in one dual simplex pivot. Let $\hat{x}_B = \bar{x}_B - \bar{a}^i t_i - \bar{a}^j t_j$ and $Q = \{q \in B : \hat{x}_q < 0 \text{ or } \hat{x}_q > 1\}$. If $Q = \emptyset$, we have $z_{x_i=v_i, x_j=v_j} = z + \bar{c}_i t_i + \bar{c}_j t_j$. Otherwise, let $\bar{N} = N \setminus \{i, j\}$ and define

$$P_q = \left\{ p \in \bar{N} : \begin{array}{l} \bar{a}^p_q < 0, \bar{x}_p = 0 \text{ or } \bar{a}^p_q > 0, \bar{x}_p = 1, \text{ if } \hat{x}_q < 0, \\ \bar{a}^p_q > 0, \bar{x}_p = 0 \text{ or } \bar{a}^p_q < 0, \bar{x}_p = 1, \text{ if } \hat{x}_q > 1. \end{array} \right.$$

If $P_q = \emptyset$ for some $q \in Q$, then the LP subject to $x_i = v_i$, $x_j = v_j$ is infeasible and we have an edge inequality valid for S. Otherwise, let

$$f_q = \begin{cases} \hat{x}_q, & \text{if } \hat{x}_q < 0 \\ \hat{x}_q - 1, & \text{if } \hat{x}_q > 1 \end{cases}$$

and

$$\Delta^* = \max_{q \in Q} \min_{p \in P_q} |\frac{f_q}{\bar{a}^q_p} \bar{c}_p|. \tag{1}$$

Then we obtain a stronger upper bound

$$z_{x_i=v_i, x_j=v_j} \leq z + \bar{c}_i t_i + \bar{c}_j t_j - \Delta^*.$$

So, if $z + \bar{c}_i t_i + \bar{c}_j t_j - \Delta^* < z_h$, we have an edge inequality valid for S^{opt}.

In order to improve this bound, we exploit the fact that our variables are binary. Since $f_q/\bar{a}^q_p$ is the change in the value of nonbasic variable x_p if it enters the basis, if $f_q/\bar{a}^q_p > 0$, $x_p = 1$, and if $f_q/\bar{a}^q_p < 0$, $x_p = 0$. So we obtain,

$$\bar{\Delta}^* = \max_{q \in Q} \min_{p \in P_q} (|\bar{c}_q| \max\{1, |\frac{f_q}{\bar{a}^q_p}|\}). \tag{2}$$

Hence, if $z + \bar{c}_i t_i + \bar{c}_j t_j - \bar{\Delta}^* < z_h$, we have an edge inequality valid for S^{opt}. Tomlin [12] used this line of argument in calculating penalties for choosing a branching variable. Since

both x_i and x_j are nonbasic, an edge inequality found in this way is not violated by the LP solution. However, such edges may help preprocessing or may be violated later in the search tree. Similar bounds can be obtained when one of x_i and x_j is basic.

One interesting option is probing a fractional variable at one of its bounds and a nonbasic variable at its current LP value. In order to do so, we either set the lower bound of the fractional variable to 1, or set the upper bound to 0. Calculating single dual pivot bounds for this case is quite simple since the current LP solution violates the bound of a single basic variable. Furthermore, edge inequalities found in this manner are necessarily violated by the current LP solution. Suppose we probe x_q such that $0 < \bar{x}_q < 1$. Then, using the terminology developed above, $\hat{x} = \bar{x}$ and $Q = \{q\}$. As we probe $x_q = 0$ ($x_q = 1$), we update $u_q = 0$ ($l_q = 1$). Let $p^* \in P_q$ be an index giving $\bar{\Delta}^*$ in (2). If $z + \bar{\Delta}^* < z_h$, then we have $x_q \geq 1$ is valid for S^{opt} if $l_q = u_q = 0$, and $x_q \leq 0$ is valid for S^{opt} if $l_q = u_q = 1$. Otherwise, let $\bar{\Delta}'$ be the second smallest value of (2). Note that $\bar{\Delta}'$ is a lower bound in the change of the objective value when x_q is set to one of its bounds and x_{p^*} is fixed at its current LP value. If $z + \bar{\Delta}' < z_h$ then the following edge inequalities are valid for S^{opt}:

- if $l_q = u_q = 1$ and $\bar{x}_{p^*} = 1$ then $x_q + x_{p^*} \leq 1$,
- if $l_q = u_q = 1$ and $\bar{x}_{p^*} = 0$ then $x_q + (1 - x_{p^*}) \leq 1$,
- if $l_q = u_q = 0$ and $\bar{x}_{p^*} = 1$ then $(1 - x_q) + x_{p^*} \leq 1$,
- if $l_q = u_q = 0$ and $\bar{x}_{p^*} = 0$ then $(1 - x_q) + (1 - x_{p^*}) \leq 1$.

We remark again that even if there is no feasible solution at hand, it is possible to find such edge inequalities if $P_q = \emptyset$. In that case, the edge inequalities are valid for S.

2.3 Edge equalities

So far, we have concentrated on edges in the conflict graph that represent inequalities of the form $x_i + x_j \leq 1$. However, allowing equality edges to represent equalities of the form $x_i + x_j = 1$ can lead to quick detection of further edge inequalities and equalities and fixing more variables. Let $G = (V, E)$ be a conflict graph, and $E^= \subseteq E$ be the set of edges of G that represent equalities, then we have the following trivial propositions:

Proposition 2.1 *If $(i,j) \in E^=$, $(j,k), (i,l) \in E$, then $(k,l) \in E$.*

$$\left.\begin{array}{rrrrl} x_i & +x_j & & & = 1 \\ x_i & & +x_l & & \leq 1 \\ & x_j & & +x_k & \leq 1 \end{array}\right\} \Rightarrow x_k + x_l \leq 1.$$

Proposition 2.2 *If $(i,j), (l,k) \in E^=$, $(i,k), (j,l) \in E$, then $(i,k), (j,l) \in E^=$.*

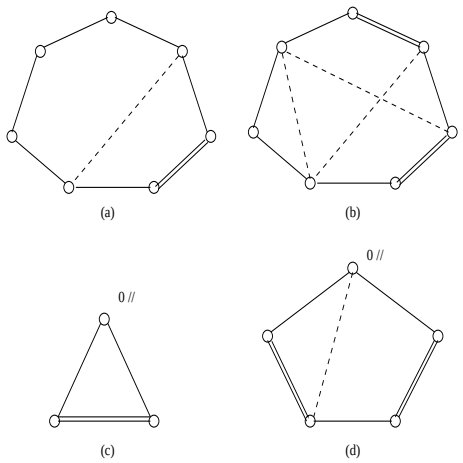

Figure 3: Edge equalities

$$\left.\begin{array}{rrrrr} x_i & +x_j & & & = 1 \\ & & x_l & +x_k & = 1 \\ x_i & & & +x_k & \leq 1 \\ & x_j & +x_l & & \leq 1 \end{array}\right\} \Rightarrow \begin{array}{l} x_i + x_k = 1 \\ x_j + x_l = 1. \end{array}$$

Proposition 2.3 *If* $(i,j) \in E^{=}$, $(i,k),(j,k) \in E \setminus E^{=}$, *then* $x_k \leq 0$.

$$\left.\begin{array}{rrrr} x_i & +x_j & & = 1 \\ x_i & & +x_k & \leq 1 \\ & x_j & +x_k & \leq 1 \end{array}\right\} \Rightarrow x_k \leq 0.$$

We can use these propositions iteratively to obtain stronger packing relaxations of an integer programming problem. We scan end vertices of edges in $E^{=}$ and apply one of the propositions until this is no longer possible.

Example Here we give some examples to illustrate the use of edge equalities. In Figure 3, equality edges are represented by double lines, and edges found by applying the above propositions are represented by dashed lines. Observe that in Figure 3 (b) an inserted edge triggers the third insertion. In Figures 3 (c) and (d) one of the variables can be fixed to its bound and eliminated from the problem.

3 Data structures and implementation

A careful implementation is required to use a conflict graph in a general purpose integer optimizer. A general purpose integer optimizer has to be able to handle a variety of problems ranging from set partitioning problems, which have very dense conflict graphs, to arbitrary mixed integer problems, which typically have much sparser conflict graphs.

When a conflict graph is very dense, keeping the conflict graph in memory becomes practically impossible. Therefore, we do not explicitly keep conflict graph edges that are derived from a GUB constraint, since the existence of such an edge can be inferred by checking whether two variables appear together in a GUB constraint. Instead, we maintain all the GUB constraints in a separate data structure that supports fast checking of whether two variables appear together in one of the GUB constraints. Our experience that preprocessing and reduced cost fixing are usually very effective on instances with many GUB constraints and can reduce the size of the instances significantly also affected the choice of an appropriate data structure for the storage of GUB constraints. The data structure for the storage of GUB constraints should also support fast deletion of constraints and variables. To accommodate fast deletion of constraints and variables as well as fast checking of whether two variables appear together in one of the GUB constraints, we use a two dimensional linked list structure to store GUB constraints. Each element in the two-dimensional linked list structure contains the row and column index of a nonzero entry in a GUB constraint, and pointers to elements to its right, left, up and down. An example is given in Figure 4.

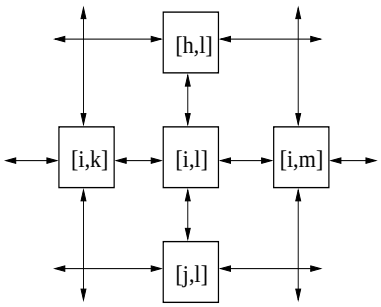

Figure 4: Data structure to store GUB constraints

Note that storing the column indices of the nonzero entries in a GUB constraint is sufficient to be able to determine whether two variables appear in the same GUB constraint, and it is also avoids expensive double arithmetic operations and comparisons carried out during probing, which makes probing practically feasible for instances with a large number of GUB constraints.

Edges derived by probing are stored in a separate data structure. The data structure we have chosen allows for easy addition of new edges and easy access to the edges incident to a given vertex. For each vertex v the array entry `last[`v`]` contains the index, say k, of the last edge having v as one of its end-vertices, or 0 if there are no edges incident to v. The other end-vertex of this edge can be found in `adj[`k`]`. The index of another edge incident to v can be found in `next[`k`]`. If there are no other edges incident to v, then `next[`k`]` is 0. The following two C functions show how to add edge (i, j) and traverse adjacent vertices of vertex i.

```
ADD_EDGE (i, j) {
  n = n+1; adj[n] = j; next[n] = last[i]; last[i] = n;
  n = n+1; adj[n] = i; next[n] = last[j]; last[j] = n;
}

ADJACENT (i) {
  for (k = last[i]; k != 0; k = next[k]) {
    /*
     * adj[k] is adjacent to vertex i
     */
  }
}
```

We illustrate the data structure by means of an example in Figure 5. In this example, edges have been added in the order $(1,3),(1,4),(3,2),(1,2)$.

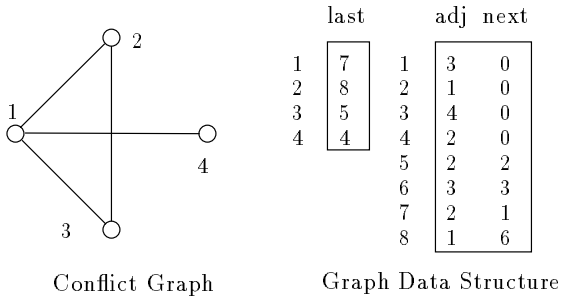

Figure 5: Conflict graph data structure

When one or more variables are fixed, new logical relations between pairs of variables may be derived. In particular, new logical implications may be derived at each node of the search tree since branching is typically done by fixing variables. Such logical implications are only locally valid, since they depend on the branching decision. Locally valid conflict graphs do not pose any theoretical difficulties. However, it is more complicated to manage dynamically changing conflict graphs. We need a data structure that allows inheriting of edges by child nodes from their parent node and that supports addition of new edges at an arbitrary node of the search tree. Furthermore, when a vertex of the conflict graph is scanned, only incident edges that are valid for the current node and its ancestors should be visited.

The data structure for the storage of edges found by probing can be extended to accommodate these requirements as follows. We store conflict graph edges in blocks, with one block for each node in the search tree. The list of edges incident to a vertex for a particular node of the search tree is linked to the list of edges incident to that same vertex in the parent node. In Figure 6 we give an illustration. The conflict graph has four vertices. The solid edge is valid at node a, dotted edges are found to be valid at node b and dashed edges are found at node c. At node c, vertex 3 is adjacent to vertices 2 and 4 via valid edges.

These edges are stored at position indices 12 and 2. The next index pointer at 12 skips edges valid at node b.

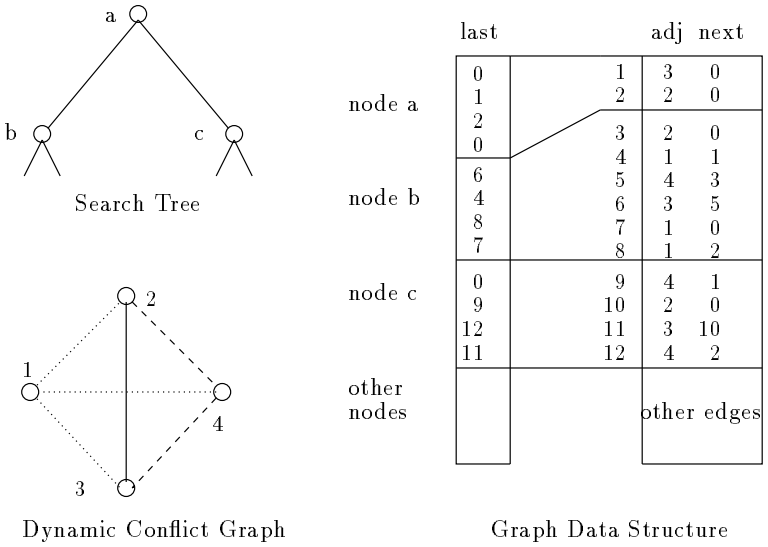

Figure 6: Dynamic conflict graph structure

Note that the chosen data structure does not support fast deletion of edges. Therefore, edge deletion is only carried out when absolutely necessary.

4 Preprocessing

Preprocessing refers to a set of simple reformulations performed on a problem instance to enhance the solution process. These reformulations try to identify infeasibilities and redundancies, to tighten bounds on variables, and to improve coefficients of constraints. Preprocessing may reduce the size of an instance as well as the integrality gap. Several papers have appeared on this subject; see e.g. [4, 11] for the specifics of these reformulation techniques.

At the heart of these techniques is the computation of lower and upper bounds on the value of the left-hand side of constraints. For "less than or equal to" constraints, the lower bounds are used to improve variable bounds and derive conflict graph edges, and the upper bounds are used to improve constraint coefficients.

The conflict graph can be used to strengthen these techniques, because it can be used to enhance the bound computations. In Section 2, we discussed how to use a conflict graph in the computation of lower bounds. It is easy to see that upper bounds can be computed similarly.

Example Consider the inequality

$$-90x_1 - 75x_2 - 120x_3 + 30x_4 \leq -79.$$

By using standard reduction techniques (Euclidean reduction and coefficient improvement [4, 11]), it can be strengthened to

$$-6x_1 - 5x_2 - 8x_3 + 2x_4 \leq -6.$$

Note that $L_{x_1=0,x_3=0} = -5 > -6$; thus the conflict edge $(\bar{1}, \bar{3})$ is valid. Using $(\bar{1}, \bar{3})$, we obtain an improved upper bound $U_{x_2=1} = -9$, which can be shown to lead to the strengthening of the inequality to

$$-3x_1 - x_2 - 4x_3 + x_4 \leq -3.$$

We now briefly discuss an efficient implementation of probing. In probing, we tentatively set variables to one of their bounds and try to identify infeasibilities and fixed variables in the resulting reduced problem. When probing a specific variable, say $x_i = v_i$, it is clearly enough to process only the set of constraints in the reduced problem that are affected by tentatively setting $x_i = v_i$. Initially, this set consists of only the constraints in which x_i appears. However, if there already exist logical implications of the form $x_i = v_i \Rightarrow x_j = v_j$, then all the constraints in which x_j occurs have to be added to the set of constraints affected by tentatively setting $x_i = v_i$ as well. Furthermore, each time a variable, say x_k, in the reduced problem can be fixed, i.e., a new implication of the form $x_i = v_i \Rightarrow x_k = v_k$ is detected, all the constraints in which x_k occurs also have to be added to the set of constraints affected by tentatively setting $x_i = v_i$. Consequently, we need a data structure that maintains a set of constraint indices and supports insertion and (random) deletion operations. We have implemented this data structure using a stack and a bit vector. When an element is inserted in the set it is pushed onto the stack and its associated bit is set to 1. The bit vector is used for membership testing. Deletion is implemented by popping an element off the stack and setting its associated bit to 0. We process constraints until either an infeasibility is detected or the set of affected constraints becomes empty.

We use an implication stack to store all the implications used in a single probing iteration (probing a specific variable). At the beginning of a probing iteration, all known implications, i.e., all edges in the conflict graph incident to the vertex representing the variable being probed, and all implications that can be inferred from the GUB constraints, are pushed onto the implication stack and the size of the stack is stored. All new implications found during the probing iteration are also pushed onto the stack. At the end of the probing iteration, if no infeasibility is detected, the new implications become edges of the conflict graph. Keeping the initial implications in the stack, allows us to restore the variable bounds at the end of the probing iteration quickly.

Next, we observe that when we probe a specific variable, say $x_i = v_i$, and find a new implication of the form $x_i = v_i \Rightarrow x_j = v_j$, then we have also found a new implication for each variable x_k for which we already have an implication of the form $x_k = v_k \Rightarrow x_i = v_i$,

namely $x_k = v_k \Rightarrow x_j = v_j$. Consequently, it may be beneficial to probe variable x_k again. We use a data structure identical to the one introduced above to maintain the set of variables that still need to be probed. Initially, this set consists of all the variables in the problem. We probe variables until the set becomes empty.

Finally, it should also be observed that many of the preprocessing and probing techniques can be implemented to run much faster for constraints with 0-1 coefficients only. Furthermore, pure 0-1 rows allow additional preprocessing techniques, such as row domination, see e.g. [1, 5, 7].

5 Cut generation

Any feasible solution to S defines a vertex packing in the conflict graph. Therefore, the vertex packing polytope associated with the conflict graph contains the convex hull of feasible solutions to S. Hence, valid inequalities for the vertex packing polytope, such as clique and odd-hole inequalities, define valid inequalities for the convex hull of feasible solutions to S.

Therefore, we may use the conflict graph to try to find valid inequalities that cut off the current LP solution $\bar{x}$. Formally, we look for $P \subseteq V$ such that

$$\Sigma_{j \in P} w_j z_j > \alpha(G(P)),$$

where

$$w_j = \begin{cases} \bar{x}_j, & j \text{ original} \\ 1 - \bar{x}_j, & j \text{ complement} \end{cases}$$

is the weight of vertex j, $z_j \in \{0, 1\}$ and $\alpha(G(P))$ is the stability number of the subgraph induced by P. We limit ourselves to clique inequalities, i.e., inequalities with $\alpha(G(P)) = 1$. Clique inequalities have been shown to be facet defining [10] for the vertex packing polytope. The separation problem for the class of clique inequalities is $\max_{P \subset V} \{\Sigma_{j \in P} w_j z_j : \alpha(G(P)) = 1\}$ and is a maximum weighted clique problem on the conflict graph.

There are at least three ways to implement cut generation based on clique inequalities derived from the conflict graph:

1. (a) Construct the conflict graph.
   (b) Generate a set of maximal cliques and store them in a clique table.
   (c) During the solution process, check the clique table for violated clique inequalities.
2. (a) Construct the relevant part of the conflict graph on-the-fly, i.e., the subgraph associated with the vertices with positive weight.
   (b) Generate maximal cliques on-the-fly and check them for violation.
3. (a) Construct the conflict graph.

(b) Generate maximal cliques on the relevant part of the conflict graph on-the-fly and check them for violation.

The advantage of the first approach is that one does everything only once, thus avoiding duplication of effort. However, there are two disadvantages that arise for instances with many GUB constraints: (1) since the conflict graph is large, identifying (all) maximal cliques may be computationally prohibitive, and (2) for the same reason storing (all) maximal cliques may be impossible. Nevertheless, this approach seems to be prevalent in general purpose solvers such as CPLEX, MINTO and OSL. Our computational results in Section 6 indicate that, this approach may not be a good choice for many problems.

The second approach circumvents the disadvantages of the first approach by doing everything on-the-fly: constructing the relevant part of the conflict graph as well as identifying violated cliques. This approach does not require storage of a clique table and, since the conflict graphs are smaller, the identification of violated cliques is faster. However, the disadvantage is that we construct a partial conflict graph over and over again. Hoffman and Padberg [5] have used this approach successfully for set partitioning problems. However, for general integer programming problems, building the conflict graph "on the fly" is not practical, since small conflict graphs may not suffice and finding implications between pairs of variables is more complex and time consuming than in set partitioning problems.

We have chosen to use the third approach, which is a compromise between the two extreme solutions. We construct the complete conflict graph once. Since the maximum weighted clique problem is NP-hard [8], we resort to a heuristic separation algorithm that performs a partial search. However, running even a heuristic separation algorithm on the complete conflict graph can be very time consuming on large graphs. Therefore, we apply a two stage approach. First we consider only vertices with positive weight. If we find a violated clique inequality, we then extend the clique to a maximal one using the remaining vertices, thus lift the clique inequality to get a high dimensional face of the vertex packing polytope. In Section 6, we empirically show the efficacy of this approach.

6 Computational results

In this section, we report on the various computational experiments conducted to test the effectiveness and efficiency of the implementation techniques and algorithms described in the previous sections. Our implementation is embedded in MINTO (version 3.0). MINTO [9] is a software system that solves mixed-integer linear programs by a branch-and-bound algorithm with linear programming relaxations. All experiments were done on an IBM RS/6000 Model 590 workstation with one hour CPU time limit.

Our data set consists of problems with varying characteristics. The first five problems are various instances from MIPLIB 3.0 [2]. The next five are real-life production planning and resource allocation problems. The next set of five problems are instances of a time-indexed formulation of a single machine scheduling problem. Due to the structure of the time-indexed formulation, we anticipated that the clique inequalities generated would be

very helpful in the solution of these problems. Finally, the last five are maximum clique problems from the DIMACS Challenge on Cliques, Coloring, and Satisfiability [6].

We have conducted five experiments. First and most important, we wanted to see if the use of conflict graphs was effective and if our implementation was robust and efficient. To do so, we solved all the instances with MINTO and compared the performance against the commercial integer programming solver CPLEX (version 4.0). Since we wanted to compare, among other things, the effectiveness and efficiency of the clique generation implementations, we turned CPLEX' clique generation on. (CPLEX performed much better on the instances in our test set with clique generation on.) The results are summarized in Table 1. In this table, we report the number of clique cuts generated, optimal value of LP relaxation at the root node, the best solution found, the percentage gap between the best upper bound and the best lower bound at termination, the number of nodes explored in the search tree and the total CPU time elapsed in seconds both for MINTO and CPLEX. Our techniques are clearly effective. In all instances the bound obtained at the root node is better, and in all but one instance the number of nodes evaluated is smaller, often significantly smaller. Except for two relatively easy instances, cpu times were better too, which shows that our implementation is efficient. The implementation appears to be robust as well, since it handled 'air04' and 'air05' without any problems. These are instances of set partitioning problems that lead to dense conflict graphs.

Next, we conducted several experiments to determine the effectiveness of the various components. MINTO incorporates other classes of cuts besides clique cuts. In order to understand how much of the performance can be attributed to clique cuts, we ran MINTO on the same set of instances with clique generation turned off. In Table 2, we present the results. This table shows that for almost all of the instances, performance degrades drastically without the clique cuts.

Conflict graphs are generated during preprocessing. Therefore the level of preprocessing plays an important role in the size of the generated conflict graph, and consequently in the overall performance. The third experiment was done to evaluate the effect of different levels of preprocessing. In Table 3, we summarize our results for levels: 0) no preprocessing, 1) simple preprocessing, 2) preprocessing and limited probing with conflict graph generation, 3) preprocessing and full probing with conflict graph generation. Although there are exceptions, in general, a higher level of preprocessing increases the size of the conflict graph, reduces the number of evaluated nodes, and reduces the solution time.

	MINTO 3.0						CPLEX 4.0					
problem	clqs	zroot	IP value	gap	nodes	cpu	clqs	zroot	IP value	gap	nodes	cpu
air04	358	55645.41	56137	0.00	443	2327	0	55535.44	56307	0.91	1315	3600
air05	341	25974.58	26374	0.00	1003	2367	0	25877.61	26540	1.36	3178	3600
dcmulti	15	185443.38	188182	0.00	4635	121	0	184034.37	188182	0.00	2597	21
mitre	85	115155.00	115155	0.00	1	204	0	114782.66	143400	24.53	42113	3600
vpm2	1	12.71	13.75	0.00	7958	173	0	9.89	14.00	8.71	605514	3600
edf1	76	869.11e+8	869.11e+8	0.00	14	726	166	869.08e+8	8.691e+10	0.00	40	526
edf2	58	869.07e+8	869.07e+8	0.00	29	469	152	869.05e+8	8.691e+10	0.00	48	527
rlp2	60	14.00	19	0.00	255	23	0	10.21	19	0.00	19701	188
steel1	193	-475.47	-472.14	0.00	3635	260	32	-477.23	-472.14	0.00	167317	2510
steel2	1067	-611.07	-571.44	6.94	348	3600	190	-611.85	-553.96	10.45	38846	3600
R3010_1	60	9654.50	9665	0.00	9	71	0	9614.00	9668	0.19	48050	3600
R3010_2	22	7128.00	7128	0.00	1	26	0	7095.33	7128	0.07	84506	3600
R3010_3	74	9073.25	9102	0.00	29	121	0	9054.40	9102	0.00	26855	1680
R3010_7	7	10677.50	10688	0.00	7	66	0	10636.91	10688	0.00	32040	1953
R3010_15	99	12122.60	12139	0.00	103	240	0	12100.08	12145	0.12	20705	3600
brock200_2	3558	-21.83	-8	163.81	21	3600	2114	-27.37	-7	197.86	813	3600
c-fat200-1	460	-12.99	-12	0.00	1	381	5257	-70	-12	333.33	105	3600
c-fat200-2	440	-24.99	-24	0.00	1	246	7125	-82.50	-24	0.00	232	1621
p_hat300-1	1982	-16.18	-6	169.72	1	3600	4143	-32.91	-6	445.17	29	3600
san200_0.7_2	1296	-18.00	-18	0.00	19	819	1043	-22.29	-18	0.00	1262	2781

Table 1: Comparison with CPLEX

	with clique cuts					without clique cuts			
problem	clique cuts	zroot	gap	nodes	time	zroot	gap	nodes	time
air04	358	55645.41	0.00	443	2327	55535.44	0.00	1032	3198
air05	341	25974.58	0.00	1003	2367	25877.61	0.00	1914	3006
dcmulti	15	185443.38	0.00	4635	121	185443.38	0.00	4287	110
mitre	85	115155	0.00	1	204	115155	0.00	1	183
vpm2	1	12.71	0.00	7958	173	12.71	0.00	10484	231
edf1	76	869.11e+8	0.00	14	726	869.07e+8	0.01	500	3600
edf2	58	869.07e+8	0.00	29	469	869.04e+8	0.01	1043	3600
rlp2	60	14.00	0.00	255	23	14.00	0.00	155	14
steel1	193	-475.47	0.00	3635	260	-473.72	0.00	33312	1716
steel2	1067	-611.07	6.94	348	3600	-611.85	9.96	590	3600
R3010_1	60	9654.50	0.00	9	71	9614.00	0.00	227	130
R3010_2	22	7128.00	0.00	1	26	7095.33	0.00	17	38
R3010_3	74	9073.25	0.00	29	121	9054.40	0.00	89	90
R3010_7	7	10677.50	0.00	7	66	10636.91	0.00	134	132
R3010_15	99	12122.60	0.00	103	140	12100	0.00	364	358
brock200_2	3553	-21.83	163.81	21	3600	-98	527.78	3436	3600
c-fat200-1	460	-12.99	0.00	1	381	-88.50	0.00	309	1120
c-fat200-2	440	-24.99	0.00	1	246	-91.50	0.00	269	927
p_hat300-1	1982	-16.18	169.72	1	3600	-147.50	1850.00	124	3600
san200_0.7_2	1296	-18.00	0.00	19	819	-13	638.46	10175	3600

Table 2: Effect of clique cuts

problem	level	zinit	zroot	cg edges	clique cuts	nodes	time
air04	0	55535.44	55645.37	0	268	156	3074
	1	55535.44	55645.41	0	239	168	2358
	2	55535.44	55645.41	185053	358	443	2327
	3	55535.44	55645.46	212321	227	155	2197
dcmulti	0	183975.54	184034.45	0	0	7072	152
	1	184034.37	184071.24	0	0	5303	112
	2	184569.16	185443.35	171	15	4635	121
	3	184569.16	185443.35	171	15	4635	123
mitre	0	114740.52	114787.42	0	0	9	3600
	1	114782.47	115155.00	0	0	1	230
	2	114878.34	115155.00	43129	85	1	204
	3	115141.50	115155.00	29625	38	1	183
rlp2	0	10.21	12.72	0	0	545	59
	1	12.64	14.00	0	0	247	15
	2	14.00	14.00	7227	60	255	23
	3	14.00	14.00	327	9	43	8
steel1	0	-612.64	-481.88	0	0	4747	537
	1	-489.44	-477.01	0	0	19704	1024
	2	-486.66	-475.47	2722	193	3635	260
	3	-486.66	-475.47	2722	193	3635	263
vpm2	0	9.89	11.20	0	0	149379	3600
	1	10.27	11.31	0	0	72094	1676
	2	11.26	12.71	3	1	7958	173
	3	11.26	12.71	3	1	8571	176

Table 3: Effect of preprocessing

The next experiment was conducted to evaluate the effect of generating conflict graph edges dynamically throughout the search tree. With any integer programming technique that is added to the basic LP based branch-and-bound algorithm to enhance the performance, a tradeoff is made between the effort (time spent on the technique) and effect (reduction in number of nodes evaluated). However, for techniques that produce only locally valid information, it is harder for such a tradeoff to turn out to be beneficial, since the technique only affects the subtree rooted at the node at which the information is generated, which may be a small part of the overall search tree. In Table 4 we compare the results obtained using a static conflict graph, which is generated once at the root node, with those obtained using a dynamic conflict graph, where we extend the conflict graph at nodes in the search tree at depths less than or equal to five. There is no clear winner. While the use of dynamic conflict graphs reduced the number of nodes evaluated in some of the problems, it increased the number of nodes evaluated in some others. Our intuition is that it only pays to use dynamic conflict graphs for tightly constrained problems, like set partitioning problems.

Finally, we performed an experiment to determine the effect of the use of optimality edges in the conflict graph. The results are shown in Table 5. Note that optimality edges

	static cg				dynamic cg			
problem	clique cuts	gap	nodes	time	clique cuts	gap	nodes	time
air04	358	0.00	443	2327	249	0.00	243	2443
air05	341	0.00	1003	2367	451	0.00	685	2127
dcmulti	15	0.00	4635	121	4	0.00	4359	159
mitre	85	0.00	1	204	83	0.00	1	208
vpm2	1	0.00	7958	173	1	0.00	8136	247
edf1	76	0.00	14	726	76	0.00	56	753
edf2	58	0.00	29	469	58	0.00	35	483
rlp2	60	0.00	255	23	59	0.00	199	18
steel1	193	0.00	3635	260	247	0.00	3632	398
steel2	1067	6.94	348	3600	848	6.17	211	3600
R3010_1	60	0.00	9	71	58	0.00	9	72
R3010_2	22	0.00	1	26	22	0.00	1	26
R3010_3	74	0.00	29	121	84	0.00	33	148
R3010_7	7	0.00	7	66	73	0.00	7	68
R3010_15	99	0.00	103	140	93	0.00	95	229
brock200_2	3553	163.81	21	3600	3537	159.37	19	3600
c-fat200-1	460	0.00	1	381	460	0.00	1	385.16
c-fat200-2	440	0.00	1	246	440	0.00	1	246
p_hat300-1	1982	169.72	1	3600	1982	169.72	1	3600
san200_0.7_2	1296	0.00	19	819	1168	0.00	13	482

Table 4: Effect of a dynamic conflict graph

can only be generated when a feasible solution is available, and that the quality of this feasible solution has a significant impact on the number of optimality edges that will be generated. For each problem we ran the algorithm with the optimal value given at the start and checked to see if the optimality edges have any effect at all in proving the optimality of the given solution. In its current form, optimality edges do not appear to be very effective. They tend to reduce the number of evaluated nodes, but this does not translate into faster solution times.

	without opt. edges			with opt. edges at root			with opt. edges at depth ≤ 5		
problem	clique cuts	nodes	time	clique cuts	nodes	time	clique cuts	nodes	time
air04	245	233	1318	232	213	1379	243	115	1908
air05	197	307	864	190	221	880	154	101	954
edf1	62	27	727	61	37	898	78	35	1705
R3010_15	80	21	128	80	21	174	77	15	126
steel1	182	4368	327	182	4053	348	232	4053	354

Table 5: Effect of optimality edges

7 Extensions to mixed conflict graphs

We are currently investigating mixed conflict graphs in which we represent logical relations between binary variables as well as logical relations between binary and continuous variables. By tentatively setting a binary variable to one of its bounds and examining whether this causes tighter bounds on *continuous* variables, we derive logical relations of the form 'if $x_i = 1$, then $y_j \leq u'_j$'. Such a logical relation can be represented in the conflict graph by an edge with weight $u_j - u'_j$, where u_j is the original upper bound on the value of variable y_j, between the vertex associated with x_i and the vertex associated with y_j. Any feasible solution to the integer program defines a mixed vertex packing in the mixed conflict graph. Consequently, the mixed vertex packing polytope associated with the mixed conflict graph forms a relaxation of the convex hull of feasible solutions to the original integer program. Hence, valid inequalities for the mixed vertex packing polytope are also valid inequalities for the integer program. We are studying the mixed vertex packing polytope to find classes of strong valid inequalities that can be used in the solution of mixed integer problems.

References

[1] A. Atamtürk, G. L. Nemhauser, and M. W. P. Savelsbergh. A combined Lagrangian, linear programming, and implication heuristic for large-scale set partitioning problems. *Journal of Heuristics*, 1:247–259, 1995.

[2] R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. P. Savelsbergh. An updated mixed integer programming library: MIPLIB 3.0. Technical Report TR98-03, Department of Computational and Applied Mathematics, Rice University, 1996. Available from `URL http://www.caam.rice.edu/~bixby/miplib/miplib.html`.

[3] M. Guignard and K. Spielberg. Logical reduction methods in zero-one programming. *Operations Research*, 29:49–74, 1981.

[4] K. Hoffman and M. Padberg. Improving LP-representations of zero-one linear programs for branch-and-cut. *ORSA Journal on Computing*, 3:121–134, 1991.

[5] K. Hoffman and M. Padberg. Solving airline crew-scheduling problems by branch-and-cut. *Management Science*, 39:667–682, 1993.

[6] D. S. Johnson and M. A. Trick, editors. *Cliques, Coloring, and Satisfiability.* American Mathematical Society, 1996. DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, Vol. 26.

[7] E. L. Johnson. Modeling and strong linear programs for mixed integer programming. In S. W. Wallace, editor, *Algorithms and Model Formulations in Mathematical Programming*, pages 1–43. Springer-Verlag, 1989. NATO ASI Series, Vol. F51.

[8] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[9] G. L. Nemhauser, M. W. P. Savelsbergh, and G. S. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15:47–58, 1994.

[10] M. Padberg. On the facial structure of set packing polyhedra. *Mathematical Programming*, 5:199–215, 1973.

[11] M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994.

[12] J. A. Tomlin. An improved branch-and-bound method for integer programming. *Operations Research*, 19:1070–1075, 1971.