# Generalized Conflict Learning for Hybrid Discrete/Linear Optimization*

Hui Li and Brian Williams

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{huili, williams}@mit.edu

**Abstract.** Conflict-directed search algorithms have formed the core of practical, model-based reasoning systems for the last three decades. At the core of many of these applications is a series of discrete constraint optimization problems and a conflict-directed search algorithm, which uses conflicts in the forward search step to focus search away from known infeasibilities and towards the optimal feasible solution. In the arena of model-based autonomy, deep space probes have given way to more agile vehicles, such as coordinated vehicle control, which must robustly control their continuous dynamics. Controlling these systems requires optimizing over continuous, as well as discrete variables, using linear as well as logical constraints.
This paper explores the development of algorithms for solving hybrid discrete/linear optimization problems that use conflicts in the forward search direction, carried from the conflict-directed search algorithm in model-based reasoning. We introduce a novel algorithm called Generalized Conflict-Directed Branch and Bound (GCD-BB). GCD-BB extends traditional Branch and Bound (B&B), by first constructing conflicts from nodes of the search tree that are found to be infeasible or sub-optimal, and then by using these conflicts to guide the forward search away from known infeasible and sub-optimal states. Evaluated empirically on a range of test problems of coordinated air vehicle control, GCD-BB demonstrates a substantial improvement in performance compared to a traditional B&B algorithm applied to either disjunctive linear programs or an equivalent binary integer programming encoding.

## 1 Introduction

Conflict-directed search algorithms have formed the core of practical, model-based reasoning systems for the last three decades, including the analysis of electrical circuits [1], the diagnosis of thousand-component circuits [5], and the model-based autonomous control of a deep space probe [10]. A conflict, also called nogood, is a partial assignment to a problem′s state variables, representing sets of search states that are discovered to be infeasible, in the process of testing candidate solutions.

* This research is funded by The Boeing Company grant MIT-BA-GTA-1 and by NASA grant NNA04CK91A.

At the core of many of the above applications is a series of discrete constraint optimization problems, whose constraints are expressed in propositional state logic, and a set of conflict-directed algorithms, which use conflicts to focus search away from known infeasibilities and towards the optimal feasible solution.

In the arena of model-based autonomy, deep space probes have given way to more agile vehicles, including rovers, airplanes and legged robots [20], which must robustly control their continuous dynamics according to some higher level plan. Controlling these systems requires optimizing over continuous, as well as discrete variables, using linear as well as logical constraints. In particular, [22] introduces an approach for model-based execution of continuous, non-holonomic systems, and demonstrates this capability for coordinated air vehicle search and rescue, using a real-time hardware-in-the-loop testbed.

In this framework the air vehicle control trajectories are generated and updated in real-time, by encoding the plan′s logical constraints and the vehicles continuous dynamics as a disjunctive linear program (DLP). A DLP [3] generalizes the constraints in linear programs (LPs) to clauses comprised of disjunctions of linear inequalities. A DLP is one instance of a growing class of hybrid representations that are used to encode mixed discrete/linear constraints, such as mixed linear logic programs (MLLPs) [13] and LCNF [16], in addition to the well known mixed integer programs (MIPs) and binary integer programs (BIPs).

In this paper we explore the development of algorithms for solving hybrid discrete/linear optimization problems (HDLOPs) that use conflicts in the forward search direction, similar to the conflict-directed A* algorithm [23]. We introduce an algorithm called Generalized Conflict-Directed Branch and Bound (GCD-BB) applied to the solution of DLPs. GCD-BB extends traditional Branch and Bound (B&B), by first constructing a conflict from each search node that is found to be infeasible or sub-optimal, and then by using these conflicts to guide the forward search away from known infeasible and sub-optimal states.

In the next section we begin by reviewing the DLP formulation. Second, we introduce the GCD-BB algorithm, including B&B for DLPs, generalized conflicts, conflict-directed search and the relaxation method. Third, we evaluate GCD-BB empirically on the test problems generated by the coordinated air vehicle path planner [22]. GCD-BB demonstrates a substantial improvement in performance compared to a traditional B&B algorithm applied to either DLPs or an equivalent BIP encoding. Finally, we conclude and discuss future work.

## 2 Disjunctive Linear Programs

A DLP is defined in Eq.1 [3], where $x$ is a vector of decision variables, $f(x)$ is a linear cost function, and the constraints are a conjunction of $n$ clauses, each of which (clause $i$) is a disjunction of ($m_i$) linear inequalities, $C_{ij}(x) \leq 0$.

$$\begin{aligned} &Minimize\ f(x) \\ &Subject\ to \bigwedge_{i=1,\ldots,n} \Big( \bigvee_{j=1,\ldots,m_i} C_{ij}(x) \leq 0 \Big) \end{aligned} \tag{1}$$

A DLP reduces to a standard Linear Program (LP) in the special case when every clause in the DLP is a *unit clause*, that is $m_i = 1, \forall i = 1, \ldots, n$. A clause is a unit clause if it only contains one linear inequality. For a DLP to be feasible, no clause in the DLP should be *violated*. A clause is violated if none of the linear inequalities in the clause is satisfied.

For example, in Fig.1 a vehicle has to go from point A to C, without hitting the obstacle B, while minimizing fuel use. Its DLP formulation is Eq. 2.

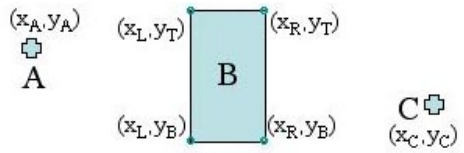



**Fig. 1.** A simple example of a hybrid discrete/linear optimization problem

$$\begin{aligned} &Minimize\ f(x) \\ &Subject\ to\ \ g(x) \leq 0 \\ &\qquad\qquad x_t \leq x_L\ \vee x_t \geq x_R\ \vee y_t \leq y_B\ \vee y_t \geq y_T, \\ &\qquad\qquad \forall t = 1, \ldots, n \end{aligned} \tag{2}$$

Here $\vee$ denotes logical *or*, and $x$ is a vector of decision variables that includes, at each time step $t(= 1, \ldots, n)$, the position, velocity and acceleration of the vehicle. $f(x)$ is a linear cost function in terms of fuel use, and $g(x) \leq 0$ is a conjunction of linear inequalities on vehicle dynamics, and the last constraint keeps the vehicle outside obstacle B, at each time step $t$.

Note that HDLOPs can also be formulated in other ways: BIPs [17, 15, 14], MLLPs [13] and LCNF [16]. Our GCD-BB algorithm, though introduced in the context of DLPs, can be generalized to other formulations. Our focus is on the generalization of forward conflict-directed search to these hybrid problems, not on the DLP encoding in particular.

## 3 The GCD-BB Algorithm

The GCD-BB algorithm builds upon B&B and incorporates three key innovative features: first, Generalized Conflict Learning learns *conflicts* comprised of constraint sets that produce either infeasibility or sub-optimality; second, Forward Conflict-Directed Search guides the forward step of the search away from regions of state space corresponding to known conflicts; and third, Induced Unit Clause Relaxation uses unit propagation to form a relaxed problem and reduce the size of its unassigned problem. In addition, we compare different search orders:

Best-first Search (BFS) versus Depth-first Search (DFS). In the following subsections, we develop these key features of GCD-BB in detail, including examples and pseudo code.

### 3.1 Branch and Bound for DLPs

**Alg. 1** BB-DLP($DLP$)

```
 1: upperBound ← +∞
 2: timestamp = 0
 3: put DLP into a FILO queue
 4: while queue is not empty do
 5:    node ← remove from queue
 6:    node.relaxedSolution ← solveLP(node.relaxedLP)
 7:    if node.relaxedLP is infeasible then
 8:       continue {node is deleted}
 9:    else if node.relaxedValue ≥ upperBound then
10:       continue {node is deleted}
11:    else
12:       expand = False
13:       for each clause in node.nonUnitClauses do
14:          if Clause-Violated?(clause, node.relaxedSolution) then
15:             expand ← True
16:             break
17:          end if
18:       end for
19:       if expand = False then
20:          upperBound ← node.relaxedValue {a new incumbent was found}
21:          incumbent ← node.relaxedSolution
22:       else
23:          put Expand-Node(node, timestamp) in queue
24:          timestamp ← timestamp + 1
25:       end if
26:    end if
27: end while
28: if upperBound < +∞ then
29:    return incumbent
30: else
31:    return INFEASIBLE
32: end if
```

GCD-BB builds upon B&B, which is frequently used by BIPs and MIPs, to solve problems involving both discrete and continuous variables. Instead of exploring the entire feasible set of a constrained problem, B&B uses bounds on the optimal cost, in order to avoid exploring subsets of the feasible set that it can prove are sub-optimal, that is, subsets whose optimal solution is not better

than the *incumbent*, which is the best solution found so far. The algorithm for B&B applied to DLPs is Alg. 1.

Alg. 1 is special for DLPs, in mainly function Clause-Violated? and function Expand-Node. Clause-Violated? checks if any clause is violated by the relaxed solution. Note that a node in the search tree represents a set of unselected clauses and a set of selected unit clauses. At each node in the search tree, the selected unit clause set and the objective function form the relaxed LP to be solved[1]. While the search tree of B&B for BIPs branches by assigning values to the binary variables, in Expand-Node, B&B for DLPs branches by splitting clauses; that is, a tree node is expanded by selecting one of the DLP clauses, and then selecting one of the clauses′ disjuncts for each of the child nodes. More detailed pseudo code can be found in [21].

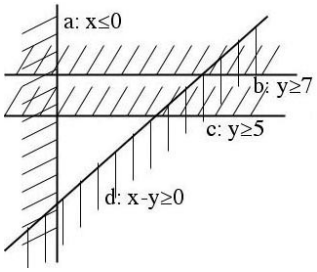
### 3.2 Generalized Conflict Learning

Underlying the power of B&B is its ability to prune subsets of the search tree that correspond to relaxed subproblems that are identified as inconsistent or sub-optimal, as seen in line 7 and 9 in Alg.1. Hence two opportunities exist for learning and pruning. We exploit these opportunities by introducing the concept of generalized conflict learning, which extracts a description from each fathomed subproblem that is infeasible or sub-optimal. This avoids exploring subproblems with the same description in the future. To accomplish this we add functions Extract-Infeasibility and Extract-Suboptimality after line 7 and 9 in Alg. 1, respectively. It is valuable to have each conflict as compact as possible, so that the subspace that can be pruned is as large as possible.

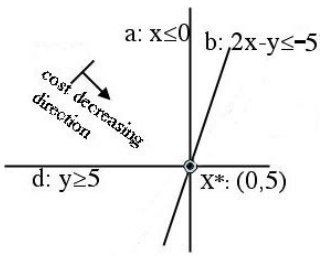
In the related fields of model-based reasoning and discrete constraint satisfaction, conflict-directed methods, such as dependency-directed backtracking [1], backjumping [2], conflict-directed backjumping [8] and dynamic backtracking [7], dramatically improve the performance of backtrack (BT) search, by learning the source of each inconsistency discovered and using this information, called a conflict (or nogood), to prune additional subtrees that the conflict identifies as inconsistent. Similarly nogood learning is a standard technique for improving BT search, in CSP [6][19] and in SAT solvers [12].

**Definition of a Conflict** In the context of DLPs, each *conflict* can be one of two types: an *infeasibility conflict*, or a *sub-optimality conflict*. An infeasibility conflict is a set of inconsistent constraints of an infeasible subproblem. An example is the constraint set {a,b,c,d} in Fig. 2(a). A sub-optimality conflict is a set of active constraints of a sub-optimal subproblem. An inequality constraint $g_i(x) \leq 0$ is active at a feasible point $\tilde{x}$ if $g_i(\tilde{x}) = 0$. An example of a sub-optimality conflict is the constraint set {a,b,d} in Fig. 2(b).

[1] $p\prime$ is a relaxed LP of an optimization problem $p$, if the feasible region of $p\prime$ contains the feasible region of $p$, and they have the same objective function. Therefore if $p\prime$ is infeasible, then $p$ is infeasible. Assuming minimization, if $p\prime$ is solved with an optimal




(a) Constraint set {a,b,c,d} is inconsistent. It is an infeasibility conflict.




(b) The optimal solution is $X^*$. Constraints a, b and d are all active. Set {a,b,d} is a sub-optimality conflict.

**Fig. 2.** Examples of conflicts

**Definition of a Minimal Conflict** A conflict is *minimal* if none of its proper subsets is a conflict. For example, the constraint set {a,c,d} in Fig. 2(a) is a minimal conflict, as it is an inconsistent constraint set and every proper subset of it is consistent. Constraint set {a,d} in Fig. 2(b) is also a minimal conflict. Note that there can be more than one minimal conflict (possibly with different cardinalities) involved in one infeasibility or sub-optimality, and a minimal conflict is not guaranteed to have the *minimum* cardinality. We extract minimal conflicts instead of any conflicts, since minimal conflicts can prune larger portion of the state space. However, we do not try to extract the minimum conflict of a subproblem, because it is NP-complete.

**Implementation** Its important to extract minimal conflicts instead of any conflicts so that larger portion of the state space can be pruned. We use methods based on the duality theory to extract minimal conflicts for infeasibility and sub-optimality. They are efficient because only one additional LP is incurred for each conflict, regardless of the number of constraints in the conflict. More specifically, minimal infeasibility sets are in 1-1 correspondence with the extreme rays of the cone formed by the modified dual of the original LP [24].

For sub-optimality, we use the dual method of LP to extract minimal conflicts. According to Complementary Slackness [11] from linear optimization theory, the non-zero terms of the optimal dual vector correspond to the set of active constraints (assuming with cardinality $k$) at the optimal solution (assuming with dimension $n$) of the LP. When the optimal solution is non-degenerate, it is guaranteed that $k \leq n$ and the active constraint set is the minimal sub-optimality conflict; when there is degeneracy, we take any $min\{k, n\}$ constraints from the active constraint set to form the minimal sub-optimality conflict.

---

value $v$, the optimal value of $p$ is guaranteed to be greater than or equal to $v$. B&B uses relaxed problems to obtain lower bounds of the original problem.

Once extracted, the minimal conflict is stored in a conflict database, *conflictDB*, indexed by a timestamp that marks its discovery time.

### 3.3 Forward Conflict-directed Search

We use forward conflict-directed search to guide the forward step of search away from regions of the feasible space that are ruled out by known conflicts. Backward search methods also use conflicts to direct search, such as dependency-directed backtracking [1], backjumping [2], conflict-directed backjumping [8], dynamic backtracking [7] and LPSAT [16]. These backtrack search methods use conflicts both to select backtrack points and as a cache to prune nodes without testing consistency. In contrast, methods like conflict-directed A* [23] use conflicts in forward search, to move away from known bad states. Thus not only one conflict is used to prune multiple subtrees, but also several conflicts can be combined as one compact description to prune multiple subtrees. We generalize this idea to guiding B&B away from regions of state space that the known conflicts indicate as infeasible or sub-optimal. Our experimental results show that forward conflict-directed search significantly outperforms backtrack search with conflicts on a range of cooperative vehicle plan execution problems.

In terms of implementation, we replace function Expand-Node in Alg. 1 with function General-Expand-Node (Alg. 2). When there is no unresolved conflict[2], the normal Expand-Node is used, and when unresolved conflicts exist, forward conflict-directed search is performed. Forward conflict-directed search (Forward-CD-Search as in Alg. 2) includes three steps: 1) Generate-Constituent-Kernels, 2) Generate-Kernels (Alg. 3) and 3) Generate-And-Test-DLP-Candidates (Alg. 5). An example is shown in Fig. 3.

**Alg. 2** General-Expand-Node($node, timestamp$,conflictDB)

```
1: conflictSet ← conflictDB(timestamp)
2: if conflictSet is empty then
3:     Expand-Node(node, timestamp)
4: else
5:     Forward-CD-Search(node, conflictSet)
6: end if
```

A *constituent kernel* is a minimal description of the states that resolve a conflict. In the context of DLPs, a constituent kernel of a conflict is a linear inequality that is the negation of a linear inequality in the conflict. For example, one constituent kernel of the minimal infeasibility conflict in Fig. 2(a) is $\{x - y \leq 0\}$[3].

[2] A node *resolves* conflict $C$ if at least one of the $C'$s disjuncts is explicitly excluded in the relaxed LP of the node.

[3] It is not the strictly correct negation of $x - y \geq 0$, but in the context of linear programming, it is correct and convenient.

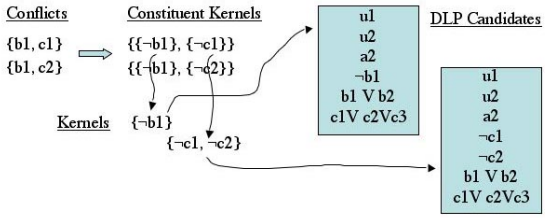



**Fig. 3.** Each conflict is mapped to a set of constituent kernels, which resolve that conflict alone. Kernels are generated by combining the constituent kernels using minimal set covering. A DLP candidate is formed for each kernel, and is checked for consistency.

In [23] *kernels* are generated to resolve all known conflicts, by combining the constituent kernels using minimal set covering. It views minimal set covering as a search and uses A* to find the kernel containing the best utility state. In the context of DLPs, we similarly build up a kernel by incrementally selecting constituent kernels (which are linear inequalities) from discovered conflicts using minimal set covering. However, we do not use A* search to identify the best kernel. In order to evaluate the heuristic during A* search, we would need to solve an LP at each step as we build the kernels; this can be very costly. Instead GCD-BB generates a DLP candidate with each kernel, as shown in Fig. 3, and prunes the DLPs that are propositionally unsatisfiable, using a fast unit propagation test before solving any relaxed LP, as shown in Alg. 5.

As shown in Generate-Kernels (Alg. 3), we use minimal set covering to generate the kernels. Fig. 4(b) demonstrates Generate-Kernels by continuing the example from Fig. 3. In particular, in Fig. 4(b) the tree branches by splitting on constituent kernels. In this example, each node represents a set of chosen constituent kernels: the root node is an empty set, and the leaf node on the right is $\{\neg c_1, \neg c_2\}$. At each node, consistency is checked (line 8 in Alg. 3), and then Generate-Kernels checks whether any of the existing kernels is a subset of the current node (line 10). If this is the case, there is no need to keep expanding the node, and it is removed. In this event, the leaf node is marked with an $X$ in Fig. 4(b); otherwise, Generate-Kernels checks whether any conflict is unresolved at the current node (line 16): if yes, the node is expanded by splitting on the constituent kernels of the unresolved conflicts (line 17); otherwise, the node is added to the kernel list, while removing from the list any node whose set of constraints is a superset of another node (line 19). The node at the far left of Fig. 4(b) resolves all the conflict and, therefore, is not expanded.

Finally, a timestamp is used to record the time that a node is created or a conflict is discovered. We use timestamps to ensure that each node resolves all conflicts, while avoiding repetition. This is accomplished through the following rules: 1. if {conflict time = node time}, there is no need to resolve the conflict when expanding the node. For example, in Fig. 4(a), node $c_3$ and its children (if any) are guaranteed to resolve the two conflicts $\{b_1, c_1\}$ and $\{b_1, c_2\}$. 2. If

**Alg. 3** Generate-Kernels($constituentKernelSet$)

```
1: root ← {}
2: root.unresolved ← constituentKernelSet {initializes node.unresolved}
3: put root in a queue
4: kernelSet ← {}
5: nodeDelete ← False {the flag to determine whether to delete a node}
6: while queue is not empty do
7:    node ← remove from queue
8:    if Consistent?(node) then
9:       for each E in kernelSet do
10:         if E ⊆ node then
11:            nodeDelete ← True {checks whether any of the existing kernels is a
               subset of the current node}
12:            break
13:         end if
14:      end for
15:      if nodeDelete = False then
16:         if Unresolved-Conflict?(node, node.unresolved) then
17:            put Expand-Conflict(node, node.unresolved) in queue {checks whether
               any conflicts are unresolved by node}
18:         else
19:            Add-To-Minimal-Sets(kernelSet, node) {avoids any node that is a
               superset of another in kernelSet}
20:         end if
21:      end if
22:   end if
23: end while
24: return kernelSet
```

**Alg. 4** Add-To-Minimal-Sets($Set, S$)

```
1: for each E in Set do
2:    if E ⊂ S then
3:       return Set
4:    else if S ⊂ E then
5:       remove E from Set
6:    end if
7: end for
8: return Set ∪ {S}
```

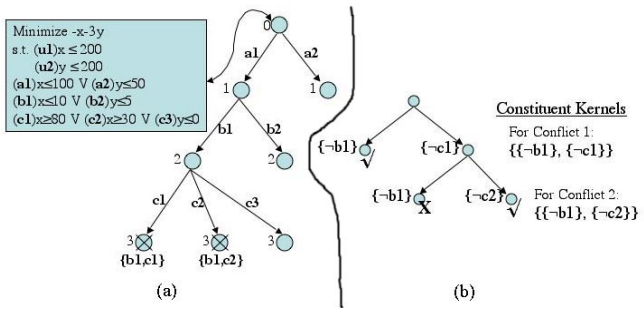



**Fig. 4.** (a) A partial tree of B&B for DLPs. The creation time of each node is shown on the left of the node. Two conflicts are discovered at the bottom. (b) The search tree for minimal set covering to generate kernels from constituent kernels.

**Alg. 5** Generate-And-Test-DLP-Candidate($kernelSet, DLP$)

1: $S \leftarrow DLP.unitClauses$
2: **for** each $kernel$ in $kernelSet$ **do**
3: **if** Consistent?($S \cup kernel$) **then**
4: $DLP.unitClauses \leftarrow S \cup kernel$ {checks whether $kernel$ is consistent with the unit clause set of $DLP$}
5: add $DLP$ in DLPList
6: **end if**
7: **end for**
8: return DLPList

{conflict time > node time}, we expand the node in order to resolve the conflict using the conflict's constituent kernels. For example, node $b_2$ and $a_2$ are to be expanded using Forward-CD-Search. 3. If {conflict time < node time}, the conflict is guaranteed to be resolved by an ancestor node of the current node, and therefore, needs not to be resolved again.

### 3.4 Induced Unit Clause Relaxation

Relaxation is an essential tool for quickly characterizing a problem when the original problem is hard to solve directly; it provides bounds on feasibility and the optimal value of a problem, which are commonly used to prune the search space. Previous research [18] typically solves DLPs by reformulating them as BIPs, where a relaxed LP is formed by relaxing the binary constraint ($x \in \{0, 1\}$) to the continuous linear constraint ($0 \leq x \leq 1$).

An alternative way of creating a relaxed LP is to operate on the DLP encoding directly, by removing all non-unit clauses from the DLP. Prior work argues for the reformulation of DLP as BIP relaxation, with the rationale that it maintains some of the constraints of the non-unit clauses through the continuous

relaxation from binary to real-valued variables, in contrast to ignoring all the non-unit clauses. However, this benefit is at the cost of adding binary variables and constraints, which increases the dimensionality of the search problem.

Our approach starts with the direct DLP relaxation. We overcome the weakness of standard DLP relaxation (loss of non-unit clauses) by adding to the relaxation unit clauses that are logically entailed by the original DLP. In the experiment section we compare our induced unit clause relaxation with the BIP relaxation and show a profound improvement on a range of cooperative vehicle plan execution problems.

**Alg. 6** Induce-Unit-Clause($DLP$)

1: $\{DLP$.unitClauses, $DLP$.nonUnitClauses$\} \leftarrow$
Unit-Propagation($\{DLP$.unitClauses, $DLP$.nonUnitClauses$\}$)
2: $DLP$.relaxedLP $\leftarrow <DLP$.objective, $DLP$.unitClauses$>$
3: return $DLP$

In terms of implementation, as seen in Alg.6 and the example in Fig. 5, Induce-Unit-Clause performs unit propagation among the unit and non-unit clauses to induce more unit clauses and simplify a DLP. A relaxed LP is also formed by combining the objective function and the unit clause set (line 2).

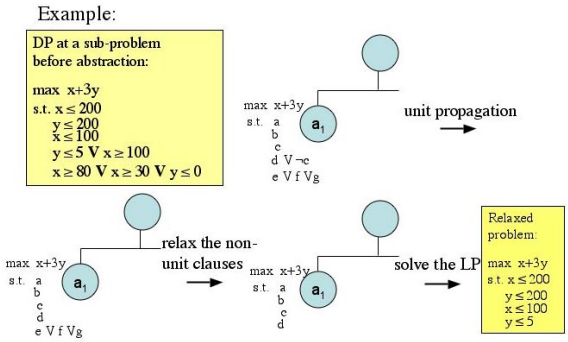



**Fig. 5.** A simple example of induced unit clause relaxation

### 3.5 Search Order: Best-first versus Depth-first

Given a fixed set of heuristic information, [4] shows that best-first search is the most efficient algorithm in terms of time efficiency. Intuitively, this is because BFS does not visit any node whose heuristic value is worse than the optimum, and all nodes better than the optimum must be visited to ensure that

the optimum is not missed. However, BFS can take dramatically more memory space than DFS. Nevertheless, with conflict learning and forward conflict-directed search, the queue of the BFS search tree can be significantly reduced. Our experimental results show that on a range of test problems BFS can take memory space similar to DFS, while taking significantly less time to find the optimum.

An additional issue for GCD-BB is that the concept of sub-optimality is rooted in maintaining an incumbent. Hence, it can be applied to DFS but not to BFS. To evaluate these tradeoffs, our experiments in the next section compare the use of BFS and conflict learning from infeasibility only, with DFS and conflict learning from both infeasibility and from suboptimality.

## 4 Experimental Performance Analysis

This section provides experimental results of the GCD-BB algorithm, compared with the benchmark B&B algorithm applied either to DLPs or to an equivalent BIP encoding, on a range of test problems of coordinated air vehicle control [22]. We also compare the effect of several algorithmic variants, in particular, BFS versus DFS, infeasibility conflict learning versus sub-optimality conflict learning and forward search versus backtrack search. While each algorithmic variant terminates with the same optimal solution, GCD-BB achieves an order of magnitude speed-up over BIP-BB. In addition, the difference in performance increases as the problem size increases.

As the bulk of the computational effort expended by these algorithms is devoted to solving relaxed LP problems, the total number and average size of these LPs are representative of the total computational effort involved in solving the HDLOPs. Note that extracting infeasibility conflicts and sub-optimality conflicts can be achieved as by-products of solving the LPs, and therefore does not incur any additional LP to solve. We use the total number of relaxed LPs solved and the average LP size as our LP solver and hardware independent measures of computation time. To measure memory space use, maximum queue size is used.

We programmed BIP-BB, GCD-BB and its variations in Java. All used the commercial software CPLEX as the LP solver. Test problems were generated using the model-based temporal planner [22], performing multi-vehicle search and rescue missions. This planner takes as input a temporally flexible state plan, which specifies the goals of a mission, and a continuous model of vehicle dynamics, and encodes them in DLPs. The GCD-BB solver generates an optimal vehicle control sequence that achieves the constraints in the temporal plan. For each Clause/Variable set, 15 problems were generated and the average was recorded in the tables.

Table 1 records the number of relaxed LPs solved by each algorithm. In both the DLP BFS and the DLP DFS cases, the algorithm with conflict learning performs significantly better than the one without conflict learning. In addition, the difference increases with the test problem size. The backtrack algorithm, based

**Table 1.** Comparison on the number of relaxed LPs

| **Clause/ Variable** | | **80/ 36** | **700/ 144** | **1492/ 300** | **2456/ 480** |
|---|---|---|---|---|---|
| **BIP-BB** | | 31.5 | 2009 | 4890 | 8133 |
| **DLP BFS** | without Conflict Learning | 24.3 | 735.6 | 1569 | 2651 |
| | Infeasibility Conflict | 19.2 | 67.3 | 96.3 | 130.2 |
| | Conflict-directed Backtrack | 23.1 | 396.7 | 887.8 | 1406 |
| **DLP DFS** | without Conflict Learning | 28.0 | 2014 | 3023 | 4662 |
| | Infeasibility Conflict | 22.5 | 106.0 | 225.4 | 370.5 |
| | Conflict-directed Backtrack | 25.9 | 596.9 | 1260 | 1994 |
| | Infeasibility+Suboptimality Conflict | 22.1 | 76.4 | 84.4 | 102.9 |
| | Suboptimality Conflict | 25.8 | 127.6 | 363.7 | 715.0 |

on dependency-directed backtracking [1], uses infeasibility conflicts as a cache to check consistency of a relaxed LP before solving it. We observe that in both the BFS and the DFS cases, the forward algorithm performs significantly better than the backward algorithm. In order to show the reason for using our DLP relaxation instead of the continuous relaxation of BIP, we compare row "BIP-BB" with row "DLP DFS without Conflict Learning", and DLP performs significantly better than BIP. To address the tradeoffs of BFS and DFS, we observe that in terms of time efficiency, BFS performs better than DFS, in both the "without Conflict Learning" and the "Infeasibility Conflict" cases. Finally, "BFS Infeasibility Conflict" performs similar to "DFS Infeasibility+Suboptimality Conflict"; for large test problems, DFS performs better than BFS.

**Table 2.** Comparison on the average size of relaxed LPs

| **Clause/ Variable** | | **80/ 36** | **700/ 144** | **1492/ 300** | **2456/ 480** |
|---|---|---|---|---|---|
| **BIP-BB** | | 90 | 889 | 1909 | 3911 |
| **DLP BFS** | without Conflict Learning | 72 | 685 | 1460 | 2406 |
| | Infeasibility Conflict | 70 | 677 | 1457 | 2389 |
| | Conflict-directed Backtrack | 72 | 691 | 1461 | 2397 |
| **DLP DFS** | without Conflict Learning | 76 | 692 | 1475 | 2421 |
| | Infeasibility Conflict | 74 | 691 | 1470 | 2403 |
| | Conflict-directed Backtrack | 75 | 692 | 1472 | 2427 |
| | Infeasibility+Suboptimality Conflict | 73 | 691 | 1470 | 2403 |
| | Suboptimality Conflict | 74 | 692 | 1471 | 2410 |

As seen in Table 2, the average size of LPs solved in BIP is much larger than that of the LPs solved for DLPs, and the difference grows larger as the problem size increases. Experiments also show that the average size of LPs solved by each DLP algorithm variant is similar to one another.

Maximum queue size of the search tree of each algorithm is recorded in Table 3. Our goal is to compare the memory use of BFS algorithms with that of DFS algorithms. BFS without Conflict Learning takes significantly more memory space than any other algorithm. Compared with DFS without Conflict Learning, its maximum queue size is from 68% to 90% larger. However, it is notable that using conflict learning, the memory taken by BFS is reduced to the same level as DFS.

**Table 3.** Comparison on the maximum queue size

| **Clause/ Variable** | | **80/ 36** | **700/ 144** | **1492/ 300** | **2456/ 480** |
|---|---|---|---|---|---|
| **BIP-BB** | | 8.4 | 30.8 | 46.2 | 58.7 |
| **DLP BFS** | without Conflict Learning | 19.1 | 161.1 | 296.8 | 419.0 |
| | Infeasibility Conflict | 6.4 | 18.3 | 38.4 | 52.5 |
| | Conflict-directed Backtrack | 15.6 | 101.7 | 205.1 | 327.8 |
| **DLP DFS** | without Conflict Learning | 6.1 | 18.7 | 25.1 | 30.3 |
| | Infeasibility Conflict | 6.5 | 21.4 | 45.0 | 57.3 |
| | Conflict-directed Backtrack | 6.1 | 18.4 | 23.5 | 28.1 |
| | Infeasibility+Suboptimality Conflict | 6.5 | 21.4 | 33.0 | 40.9 |
| | Suboptimality Conflict | 6.5 | 21.6 | 38.7 | 47.0 |

## 5 Discussion

This paper presented a novel algorithm, Generalized Conflict-Directed Branch and Bound, that efficiently solves DLP problems through a powerful three-fold method, featuring *generalized conflict learning, forward conflict-directed search* and *induced unit clause relaxation*. The key feature of the approach reasons about infeasible or sub-optimal subsets of state space using conflicts, in order to guide the forward step of search, by moving away from regions of state space corresponding to known conflicts. Our experiments on model-based temporal plan execution for cooperative vehicles demonstrated an order of magnitude speed-up over BIP-BB.

## References


1. Stallman, R. and Sussman, G.J.: Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. J. of Artificial Intelligence. **9** (1977) 135196
2. Gaschnig, J.: Experimental Case Studies of Backtrack vs. Waltz-type vs. New Algorithms for Satisfying Assignment Problems. Proceedings of The 2nd Canadian Conference on AI. (1978)
3. Balas, E.: Disjunctive programming. Annals of Discrete Math. **5** (1979) 3-51

4. Dechter, R. and Pearl, J.: Generalized Best-first Search Strategies and the Optimality of A*. J. of ACM. **32** (1985) 506-536
5. de Kleer, J. and Williams, B.: Diagnosis with Behavioral Modes. Proceedings of IJCAI. (1989)
6. Dechter, R.: Enhancement Schemes for Constraint Processing: Backjumping, Learning and Cutset Decomposition. J. of Artificial Intelligence. **41** (1990) 273-312
7. Ginsberg, M.: Dynamic Backtracking. J. of Artificial Intelligence Research. **1** (1993) 25-46
8. Prosser, P.: Hybrid Algorithms for the Constraint Satisfaction Problem. J. of Computational Intelligence. **9(3)** (1993) 268-299
9. Williams, B. and Cagan, J.: Activity Analysis: The Qualitative Analysis of Stationary Points for Optimal Reasoning. Proceedings of AAAI. (1994)
10. Williams, B. and Nayak, P.: A Model-based Approach to Reactive Self-Configuring Systems. Proceedings of AAAI. (1996)
11. Bertsimas, D. and Tsitsiklis, J.N.: Introduction to Linear Optimization. Athena Scientific. (1997)
12. Bayardo, R. J. and Schrag, R. C.: Using CSP Look-back Techniques to Solve Real-world SAT Instances. Proceedings of AAAI. (1997)
13. Hooker, J.N. and Osorio, M.A.: Mixed Logical/Linear Programming. J. of Discrete Applied Math. **96-97** (1999) 395-442
14. Kautz, H. and Walser, J.P.: State space planning by integer optimization. Proceedings of AAAI. (1999)
15. Vossen, T. and Ball, M. and Lotem, A. and Nau, D.: On the Use of Integer Programming Models in AI Planning. Proceedings of IJCAI. (1999)
16. Wolfman, S. and Weld, D.: The LPSAT Engine & Its Application to Resource Planning. Proceedings of IJCAI. (1999)
17. Schouwenaars, T. and de Moor, B. and Feron, E. and How, J.: Mixed Integer Programming for Multi-Vehicle Path Planning. Proceedings of European Control Conference. (2001)
18. Hooker, J.N.: Logic, Optimization and Constraint Programming. INFORMS J. on Computing. **14** (2002) 295-321
19. Katsirelos, G. and Bacchus, F.: Unrestricted Nogood Recording in CSP Search. Proceedings of CP. (2003)
20. Hofmann, A. and Williams, B.: Safe Execution of Temporally Flexible Plans for Bipedal Walking Devices. Plan Execution Workshop of ICAPS. (2005)
21. Li, H.: Generalized Conflict Learning for Hybrid Discrete Linear Optimization. Master's Thesis, M.I.T. (2005)
22. Léauté, T. and Williams, B.: Coordinating Agile Systems Through The Model-based Execution of Temporal Plans. Proceedings of AAAI. (2005)
23. Williams, B. and Ragno, R.: Conflict-directed A* and its Role in Model-based Embedded Systems. J. of Discrete Applied Math. (to appear 2005)
24. J. Gleeson and J. Ryan: Identifying minimally inconsistent subsystems of inequalities. ORSA J. Computing 2 (1990)