Generalized Conflict Learning for Hybrid Discrete/Linear Optimization

by

Hui Li

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Master of Science in Aeronautics and Astronautics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2005

© Massachusetts Institute of Technology 2005. All rights reserved.

Author . . . .
Department of Aeronautics and Astronautics
May 20, 2005

Certified by . . . .
Brian C. Williams
Associate Professor
Thesis Supervisor

Accepted by . . . .
Jaime Peraire
Professor of Aeronautics and Astronautics
Chair, Committee on Graduate Students

Generalized Conflict Learning for Hybrid Discrete/Linear Optimization

by

Hui Li

Submitted to the Department of Aeronautics and Astronautics
on May 20, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science in Aeronautics and Astronautics

Abstract

Conflict-directed search algorithms have formed the core of practical, model-based reasoning systems for the last three decades. In many of these applications there 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 solution. In the arena of model-based autonomy, discrete systems, like deep space probes, have given way to more agile systems, 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 and non-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, generalizing from the conflict-directed search algorithms of 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. We evaluate GCD-BB 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 program encoding.

Thesis Supervisor: Brian C. Williams
Title: Associate Professor

Acknowledgments

First of all, I would like to thank my advisor, Brian Williams, for his guidance and encouragement on my research and working so hard with me to make the thesis deadline.

I would like to thank my caring roommates, Caroline Maier and Jit Kee Chin, especially Caroline, for feeding and taking care of me when I was overwhelmed by work, and cheering me up when I was down. They are not just my roommates; they are my family.

I would like to thank my parents, for their unconditional love and support, and for their care and patience during the time when I was stuck in China for 9 months.

I would like to thank MERS group, for making our lab a comfortable and stimulating part of my life. Especially, Thomas Léauté, for providing test problems for my algorithm, and giving immediate and helpful comments on my thesis, Lars Blackmore, for the insightful discussions we had, the comments he gave on the early draft of my thesis and his jokes; Oliver Martin, for his help on LaTex, our having fun together sailing and flying, and being a good buddy (I wish him the best of luck in his job in California); Martin Sachenbacher, for explaining patiently to me the concepts in CSP and commenting on my papers; John Stedl, for being a cool friend and adding an artistic flavor to the lab;-). I would also like to thank my friend, Joël Alwen, for his help with my code.

My research was funded by The Boeing Company under contract MIT-BA-GTA-1 and by NASA under contract NNA04CK91A.

Contents

1 Introduction **13**

1.1 Problem Statement . . . 15

1.2 Hybrid Discrete/Linear Optimization Problems . . . 15

1.3 Overview of Approach . . . 16

1.3.1 Generalized Conflict Learning . . . 17

1.3.2 Forward Conflict-Directed Search . . . 17

1.3.3 Induced Unit Clause Relaxation . . . 18

1.4 Key Empirical Results . . . 18

1.5 Related Work . . . 19

1.6 Chapter Overview . . . 20

2 Problem Formulation **21**

2.1 Disjunctive Linear Programming . . . 21

2.2 Binary Integer Programming . . . 23

2.3 LCNF . . . 24

2.4 Mixed Logical Linear Programming . . . 25

2.5 A Coordinated Air Vehicle Control Example . . . 26

2.5.1 Problem Statement . . . 27

2.5.2 DLP Encodings . . . 28

3 Technical Background **31**

3.1 Branch and Bound . . . 31

3.1.1 Lower and Upper Bounds . . . 31

3.1.2 An Example of B&B for BIPs 33
3.2 Conflict-Directed A* 34
3.2.1 An Example of CD-A* 35
3.2.2 The Process of CD-A* 35
3.3 Activity Analysis 40

4 The GCD-BB Algorithm 43

4.1 Branch and Bound for DLPs 43
4.2 Generalized Conflict Learning 47
4.2.1 Conflicts 48
4.2.2 Minimal Conflicts 49
4.2.3 Conflict Extraction 50
4.3 Forward Conflict-directed Search 52
4.4 Induced Unit Clause Relaxation 58
4.5 Search Order: Best-first versus Depth-first 63

5 Evaluation and Discussion 65

5.1 Empirical Evaluation 65
5.1.1 The Average LP Size 66
5.1.2 The Total Number of LPs 67
5.1.3 Maximum Queue Size 69
5.2 Discussion 70

List of Figures

1-1 A simple example of a hybrid discrete/linear optimization problem . . 16

2-1 Map of the terrain for the fire-fighting example 27

2-2 Receding horizon continuous planner 27

2-3 Temporally flexible state plan 28

3-1 A simple example of B&B for BIP. Beside each node there is the corresponding relaxed LP to solve. The number in each node is the optimal cost. The optimal solution at each node is in a box. The node with double circles is an incumbent, and the dashed lines represent the subtrees that are pruned. 33

3-2 An OCSP example, with observed inputs and outputs indicated. . . . 35

3-3 A* Search examines all best cost states leading up to the best consistent state. 36

3-4 Conflict-directed A* focuses search using discovered conflicts. a) - d) represent snapshots along a prototypical search. Circles represent states. Filled in circles have been tested for consistency. Regions in grey have been ruled out by conflicts. Only state S9 is consistent. . . 37

3-5 The search tree created by Conflict-directed A* to identify all kernels. Visited nodes that are kernels are check marked, while those that are not are crossed off. 38

3-6 Left: Tree expansion for kernel $\{O_1 = U\}$, producing Candidate 2. Only the best valued child of the root is expanded, not all children. Right: Tree expansion for kernel $\{O_1 = U\}$, producing Candidate 3. When node $O_2 = U$ is expanded, its best child and its next best sibling are created. 39

4-1 The search tree of B&B for DLPs branches by splitting clauses. Each node represents a DLP. 44
4-2 An example of a degenerate unique optimum. 51
4-3 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. 54
4-4 (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. 56
4-5 An example of induced unit clause relaxation: from Eq. 4.4 to Eq. 4.8 62

List of Tables

2.1 Comparison on the worst-case search space: DLP v.s. BIP. Each DLP has n clauses, each being a disjunction of m linear inequalities. 23

5.1 Comparison on the average size of relaxed LPs 67

5.2 Comparison on the number of relaxed LPs 67

5.3 Comparison on the maximum queue size 70

Chapter 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 [28], the diagnosis of thousand-component circuits [8], and the model-based autonomous control of a deep space probe [34]. A conflict, also called a nogood [28, 11, 7], is a partial assignment to a problem′s state variables, representing sets of search states that are discovered to be infeasible, often in the process of testing candidate solutions.

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 an algorithm, called conflict-directed A* [35], 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 [33], deep space probes [32] have given way to more agile vehicles, including rovers, airplanes and legged robots [14], 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 and non-linear as well as logical constraints. In particular, [20] introduces an approach for model-based execution of linear, 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 [1] 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) [16] and LCNF [36], in addition to the well known mixed integer program (MIP) and binary integer program (BIP) representations. We will refer to this class of problems as hybrid discrete/linear optimization problems (HDLOPs).

In this thesis we explore the development of algorithms for solving HDLOPs that use conflicts in the forward search direction, based on the conflict-directed A* algorithm [35], which uses conflicts to solve discrete optimal satisfiability (SAT) problems. 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. Our algorithm is composed of three innovations. First, *generalized conflict learning* efficiently learns conflicts from subproblems that are inconsistent as well as sub-optimal, when solving the subproblems. Second, *forward conflict-directed search* guides the forward step of search away from regions of state space corresponding to known conflicts. Third, *induced unit clause relaxation* forms relaxed subproblems from the set of unit clauses that are induced from the original problem.

With respect to other HDLOP algorithms, GCD-BB is closely related to the LP-SAT algorithm [36] in the way in which it extracts conflicts from LPs, and is similar to activity analysis (AA) [31] in the way in which it encodes sets of sub-optimal states. Note, however, that LPSAT solves SAT problems, not optimization problems; it is a combination of an LP solver and a SAT solver. AA solves non-linear programs (NLPs) using a conflict-based candidate generation algorithm; it does not include discrete choice variables. GCD-BB, however, differs in the way that it uses

this information to guide B&B search towards the optimal solution.

Our experiments on model-based temporal plan execution for cooperative vehicles demonstrated an order of magnitude speed-up over a traditional B&B algorithm applied to either DLPs or an equivalent BIP encoding.

1.1 Problem Statement

This thesis addresses the problem of creating an optimal state trajectory, based on a continuous linear dynamic model and a set of logical constraints. The objective is two-fold: first, to develop an efficient algorithm based on conflict learning to solve hybrid discrete/linear problems that are formulated in DLPs; second, to evaluate and compare the effect of different elements of the algorithm, such as conflict learning, problem encoding, search order, on real-world cooperative vehicle path planning problems.

1.2 Hybrid Discrete/Linear Optimization Problems

Elaborating upon earlier discussions, problems that combine discrete and linear optimization are usually formulated in three ways. First, by introducing integer (or binary) variables and corresponding constraints to real-valued LPs, known as MIPs or BIPs [27, 30, 17]. Second, by augmenting LPs with propositional variables so that the propositional variables can be used to trigger linear constraints. Examples of this formulation include MLLP [16] and LCNF [36]. Note that LCNF formulates a SAT problem, not an optimization problem. Third, through propositional logic formulae in which each proposition is a linear constraint. A CNF instance of this is known as a DLP [1]. Note that this form does not add any discrete variables. The algorithm proposed in this thesis solves problems formulated as DLPs, which combine the expressive power of propositional logic with that of LPs. For example, in Fig.1-1 a vehicle has to go from point A to C, without hitting the obstacle B, while minimizing fuel use. Its DLP formulation is Eq. 1.1.

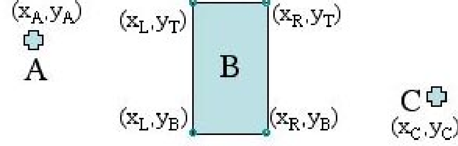

Figure 1-1: A simple example of a hybrid discrete/linear optimization problem

$$
\begin{aligned}
&Minimize\ f(x) \\
&Subject\ to\ \ g(x) \leq 0 \\
&\qquad x_t \leq x_L \ \vee x_t \geq x_R \ \vee y_t \leq y_B \ \vee y_t \geq y_T, \\
&\qquad \forall t = 1, \ldots, n
\end{aligned}
\tag{1.1}
$$

Here $\vee$ denotes logical or, and x is a vector of decision variables that includes, at each time step $i(= 1, \ldots, n)$, 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 constraints on vehicle dynamics, and the last constraint keeps the vehicle outside obstacle B, at each time step i.

A real-world example of DLP encodings for the coordinated air vehicle control problem is given in [20]. We use these encodings as test problems, to empirically evaluate the GCD-BB algorithm. These encodings are discussed in Chapter 2.

1.3 Overview of Approach

We introduce a novel algorithm for efficiently solving DLPs called Generalized Conflict-Directed Branch and Bound. It extends the B&B algorithm [19] using logical inference to help identify relaxed LP problem constraints. In addition, it generalizes, from problem instances, the source of each discovered infeasibility and sub-optimality, called a *conflict*. Conflicts are used to guide forward search by pruning the state space. The GCD-BB algorithm has three key features: Generalized Conflict Learning, Forward Conflict-Directed Search and Induced Unit Clause Relaxation, which are summarized in the following subsections through examples.

1.3.1 Generalized Conflict Learning

Conflicts traditionally are used to summarize discrete variable assignments that are inconsistent. Generalized Conflict Learning learns conflicts comprised of linear constraint sets, rather than variable assignments. In addition, the constraint sets are those that produce sub-optimality as well as infeasibility. More specifically, during the search process, whenever a subproblem is identified as infeasible or sub-optimal, a minimal subset of constraints that cause the infeasibility or sub-optimality is extracted, using efficient methods. For example, the constraint set $\{x \leq 0, y \geq 7, x \geq 5\}$ produces infeasibility and we extract the minimal subset of it, $\{x \leq 0, x \geq 5\}$, that causes the infeasibility. An example of a sub-optimal subproblem is in Eq. 1.2, assuming the optimal value of the best solution to the overall problem found so far, called the incumbent, is -10.

$$\begin{aligned} Minimize\ & x + y \\ Subject\ to\ \ & x \geq 0 \\ & y \geq 0 \\ & x + y \leq 5 \end{aligned} \tag{1.2}$$

Its optimal solution is $\{x = 0, y = 0\}$ and the optimal value is worse than the incumbent. Hence the subproblem is sub-optimal, and we extract the minimal subset of the constraints, $\{x \geq 0, y \geq 0\}$, that causes the sub-optimality.

1.3.2 Forward Conflict-Directed Search

Forward Conflict-Directed Search heuristically guides the forward step of search away from regions of state space denoted by known conflicts. Backward search methods also use conflicts to direct search, such as dependency-directed backtracking [28], backjumping [12], conflict-directed backjumping [25], dynamic backtracking [13] and LPSAT [36]. These backtrack search methods use conflicts both to select backtrack points and as a form of dynamic programming when testing candidates. In contrast,

methods like conflict-directed A* [26, 35] use conflicts in the forward search, to move away from known "bad" states. Thus not only is one conflict 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.

1.3.3 Induced Unit Clause Relaxation

Induced Unit Clause Relaxation forms a relaxed problem from a subset of the unit clauses that are induced from the original problem. Previous research [15] 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 ($0 \le x \le 1$). 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. The latter approach creates a weaker relaxation than the continuous relaxation of BIP, but it benefits from avoiding the addition of binary variables and constraints, which increases the dimensionality of the search problem. Our approach starts with the direct DLP relaxation and overcomes 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. Our relaxation method also avoids adding binary variables and constraints, which can significantly increase the dimensionality of the search problem.

1.4 Key Empirical Results

The key results of our empirical study are the following. First, the algorithm that performs the best is the one that uses 1) DLP encodings, 2) conflict-directed forward search, and 3) either best-first search with infeasibility conflict learning or 4) depth-first search with sub-optimality and infeasibility conflict learning. Second, this algorithm achieves an order of magnitude speed-up over BIP-BB.

1.5 Related Work

This thesis builds upon the Conflict-Directed Clausal LP Branch and Bound method [18], which uses B&B for DLPs and learns infeasible states as conflicts to guide search. The improvements in this thesis over the Conflict-Directed Clausal LP Branch and Bound algorithm are the following. First, we extract conflicts more efficiently, that is, as a by-product of solving an LP problem instead of solving a number of additional LP problems. Second, we generalize the concept of a conflict to include sub-optimality; as a result, larger subspaces can be pruned during search. Finally, we perform a more thorough empirical study, comparing the effect of each element of the algorithm, such as problem encoding, search order, search method and conflict learning.

GCD-BB is also closely related to LPSAT [36]. LPSAT determines satisfiability of a hybrid logic LP problem, rather than extracting the optimal solution, as in GCD-BB. LPSAT is a combination of the CASSOWARY [5] LP solver and the RELSAT [2] SAT solver. It searches over propositional variables, while a variable assignment may "trigger" the inclusion of a linear constraint. It learns an inconsistent partial variable assignment, called a minimal conflict set, and uses it to prune multiple subtrees. The difference from our algorithm is the following. First, our definition of a conflict is more general in that it includes sub-optimality as well as infeasibility. Second, we use the conflicts to guide the forward step of search, instead of during backjumping. Last, LPSAT terminates with any feasible solution, not the globally optimal solution.

The concept of search directed by conflicts draws from Conflict-Directed A* [35]. This method considerably speeds up the search process by generalizing individual infeasibilities into regions of the state space that must contain only infeasible states and by using them to guide the forward step of search. As mentioned earlier, the concept of conflict learning from sub-optimality draws from activity analysis (AA) [31], which reasons using qualitative abstractions of sub-optimal subspaces in non-linear optimization, in order to guide the numerical methods away from subspaces with the same abstractions. GCD-BB is different from AA in that AA extracts conflicts from the Karush-Kuhn-Tucker (KKT) conditions, while we learn conflicts

from solved subproblems that are sub-optimal.

Van Hentenryck′s work [29, 6] on interval and local methods for non-linear optimization problems is related to our algorithm in that they guide search for solutions using consistency checking, constraint propagation and approximations. Neumaier′s survey [22] studies the utility and complexity of sub-optimality pruning for non-linear optimization.

1.6 Chapter Overview

The rest of this thesis is organized as follows. Chapter 2 studies the problem formulation and reviews the cooperative air vehicle coordination problem solved by [20], as well as its encoding. Chapter 3 reviews the technical background of GCD-BB, including Branch & Bound, Conflict-Directed A* and Activity Analysis. Chapter 4 introduces the main algorithm, Generalized Conflict-Directed Branch & Bound (GCD-BB). It develops the three key elements of the algorithm in detail through examples and pseudo code, examines other options for each algorithmic element, and then discusses analytically the advantage of our approach. Chapter 5 describes the experiments, shows the results of comparing different methods, and analyzes empirically the advantage of our approach, followed by a discussion.

Chapter 2

Problem Formulation

Recall from Chapter 1 that problems combining discrete and linear optimization are usually formulated in three ways: (1) MIP and BIP; (2) MLLP and LCNF; or (3) DLP. The algorithm proposed in this thesis solves problems formulated as DLPs. 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. In this chapter, we introduce the concept of a DLP, discuss its relationship to BIP, MLLP and LCNF encodings, and review the cooperative air vehicle coordination problem solved by [20] and used as a benchmark in Chapter 5, along with its DLP encoding.

2.1 Disjunctive Linear Programming

In general, a DLP takes the form shown in Eq. 2.1, 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, $C_{ij}(x) \leq 0$. A DLP reduces to a standard 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 constraint. For a DLP to be feasible, every clause in the DLP must be *resolved*. A clause is resolved if at least one of the linear inequalities in the

clause is satisfied.

$$\begin{aligned} &Minimize\ f(x) \\ &Subject\ to \bigwedge_{i=1,\ldots,n} (\bigvee_{j=1,\ldots,m_i} C_{ij}(x) \leq 0) \end{aligned} \tag{2.1}$$

Any DLP can be converted to a BIP, by adding one binary variable for each linear inequality that appears in a non-unit clause of the DLP and adding one linear constraint for each such clause. The general form of the BIP converted from Eq. 2.1 using the big M method is shown in Eq. 2.2.

$$\begin{array}{lll} Minimize & & f(x) \\ Subject\ to & \bigwedge_{i\in\{1,\ldots,n|m_i>1\}} & \left(\begin{array}{c} \wedge_{j=1,\ldots,m_i} C_{ij}(x) \leq M(1-b_{ij}) \\ \sum_{j=1,\ldots,m_i} b_{ij} \geq 1 \end{array} \right) \\ & \wedge_{i\in\{1,\ldots,n|m_i=1\}} C_{ij} \leq 0 & \end{array} \tag{2.2}$$

On the other hand, any BIP can also be converted to a DLP. In the most general case, it is done by enumerating all possible assignments to the binary variables in each constraint and explicitly making each of those assignments that results in a unique linear constraint a disjunct in one clause. Each constraint in the BIP is turned into a clause of the DLP: the constraint that involves no binary variable corresponds to a unit clause, and the constraint that involves n binary variables corresponds to a clause with at most 2^n disjuncts.

2.2 Binary Integer Programming

As a different encoding, the example in Fig.1-1 can be formulated as a BIP (Eq. 2.3), where M is an arbitrarily large positive number.

$$\begin{aligned}
Minimize\ & f(x) \\
Subject\ to\ & g(x) \leq 0 \\
& x_t - x_L \leq M(1 - b_{t1}) \\
& x_t - x_R \geq M(b_{t2} - 1) \\
& y_t - y_B \leq M(1 - b_{t3}) \\
& y_t - y_T \geq M(b_{t4} - 1) \\
& \sum_{j=1,\ldots,4} b_{tj} \geq 1 \\
& b_{tj} \in \{0, 1\},\ \forall j = 1, \ldots, 4 \\
& \forall t = 1, \ldots, n
\end{aligned} \tag{2.3}$$

Table 2.1: Comparison on the worst-case search space: DLP v.s. BIP. Each DLP has n clauses, each being a disjunction of m linear inequalities.

n\m	**2**		**4**		**8**		**12**	
	DLP	BIP	DLP	BIP	DLP	BIP	DLP	BIP
1	2	4	4	16	8	256	12	4096
2	4	16	16	256	64	65536	144	1.7E+7
4	16	256	256	65536	4096	4.3E+9	20736	2.8E+14
8	256	65536	65536	4.3E+9	1.7E+7	1.8E+19	4.3E+9	7.9E+28
12	4096	1.7E+7	1.7E+7	2.8E+14	6.9E+10	7.9E+28	8.9E+12	2.2E+43

The transformation from a DLP to a BIP shows that different encodings can have profoundly different sizes of the complete search tree. In particular, suppose a DLP has n clauses, each being a disjunction of m linear constraints. Then its B&B search tree has m^n leaf nodes. On the other hand, the B&B search tree for the equivalent BIP using the big M method has $2^{m \cdot n}$ leaf nodes. Table 2.1 shows a comparison of

the worst-case search space size of DLPs and BIPs, as a function of the number of leaf nodes in the complete search tree. BIP has several methods to formulate HDLOPs besides the big M method, however, they share the same disadvantage in terms of growth in search tree size. Note that a method operating on a larger search tree is not necessarily slower than a method on a smaller search tree. What matters is the effectiveness of the method at search tree pruning, which controls the amount of the tree visited. However, tree size is one useful indicator of problem difficulty. Our experimental results in Chapter 5 demonstrate the effectiveness of the pruning method.

2.3 LCNF

The LCNF formulation [36] represents another way to combine propositional logic with metric constraints. The key to LCNF is the concept of triggers: each propositional variable may *trigger* a metric constraint, and this constraint is enforced whenever the *trigger variable* is assigned *true*.

An LCNF problem is a five-tuple $< R, V, \Delta, \Sigma, \Gamma >$ in which R is a set of real-valued variables, V is a set of propositional variables, Δ is a set of linear equality and inequality constraints over variables in R, Σ is a propositional formula in CNF over variables in V, and Γ is a mapping from V to Δ and establishes the constraint triggered by each propositional variable. LCNF is exactly MLLP without discrete variables h or the objective function f or the restriction from arbitrary propositional formula to CNF. Eq. 2.4 is an example introduced in [36]. *Italicized* variables are boolean-valued; other variables are real-valued.

$$
\left.\begin{array}{rcc}
MaxLoad & \rightarrow & (\text{load} \leq 30) \\
MaxFuel & \rightarrow & (\text{fuel} \leq 15) \\
MinFuel & \rightarrow & (\text{fuel} \geq 7 + \text{load}/2) \\
AllLoaded & \rightarrow & (\text{load} = 45)
\end{array}\right|
\begin{array}{c}
MaxLoad \\
MaxFuel \\
Deliver \\
\neg Move \vee MinFuel \\
\neg Move \vee Deliver \\
\neg GoodTrip \vee Deliver \\
\neg GoodTrip \vee AllLoaded
\end{array}
\tag{2.4}
$$

For the obstacle example in Fig.1-1 in Chapter 1, the LCNF formulation is Eq. 2.5. Any DLP can be converted to LCNF, by assigning each linear inequality a propositional variable, as seen in Eq. 2.5. On the other hand, any LCNF can also be converted to a DLP, as LCNF is in CNF form.

$$
\begin{array}{ll}
Minimize & f(x) \\
Subject\ to & \left.\begin{array}{c}
l \rightarrow (g(x) \leq 0) \\
l_{i1} \rightarrow (x_i \leq x_L) \\
l_{i2} \rightarrow (x_i \geq x_R) \\
l_{i3} \rightarrow (y_i \leq y_B) \\
l_{i4} \rightarrow (y_i \geq y_T)
\end{array}\right| \quad l \wedge (l_{i1} \vee l_{i2} \vee l_{i3} \vee l_{i4}) \\
& \forall i = 1, \ldots, n
\end{array}
\tag{2.5}
$$

2.4 Mixed Logical Linear Programming

Mixed Logical Linear Programming (MLLP) is another approach to formulating optimization problems that have both discrete and continuous elements. It is more general and expressive than LCNF. It has been applied to chemical engineering network synthesis problems, warehouse location problems and flow shop scheduling problems, and demonstrated a better performance than MIP on those problems [16]. It extends MIP by introducing logic-based modeling. Rather than require that a feasible solution satisfy a fixed set of inequalities, an MLLP model can contain several alternative sets

of inequalities. The logical formulas govern which sets must be satisfied by a feasible solution.

As introduced in [16], the MLLP formulation has the form in Eq. 2.6.

$$\begin{aligned} &\mathit{Minimize}\ f(x) \\ &\mathit{Subject\ to}\ \ p_j(y,h) \rightarrow (A^j x \geq a^j), j \in J \mid \ q_i(y,h), i \in I. \end{aligned} \tag{2.6}$$

Each constraint has a logical part, to the right of the vertical bar, and a continuous part, to the left. The logical part consists of formula $q_i(y,h)$ over propositional variables $y = (y_1, \ldots, y_n)$ and discrete variables $h = (h_1, \ldots, h_m)$, which takes on values in a finite domain. For example, $q_i(y,h)$ could be $(y_1 \vee y_2) \wedge (h_1 \neq h_2)$. The continuous part of the constraint associates logical formula $p_j(y,h)$ with systems $A^j x > a^j$ of linear inequalities. A system $A^j x > a^j$ is enforced when $p_j(y,h)$ is true. In general, the formula p_j and q_i may take on any form that is convenient for the purpose at hand, provided that their truth value is a function of the values of propositions y and discrete variables h.

Any DLP can be converted to a MLLP, by assigning each linear inequality a propositional variable. On the other hand, any MLLP can also be converted to a DLP, by converting the relation between $q_i(y,h)$ and $p_j(y,h)$ to CNF. A potential problem with this conversion is that it can be computationally infeasible. For example, if $q = \bigvee_{i=1,\ldots,20}(\bigwedge_{j=1,\ldots,5} p_{ij})$, then the CNF transformation is $q = \bigwedge_{k=1,\ldots,N}(\bigvee_{l=1,\ldots,20} p_{kl})$, where N can be as large as 5^{20}.

2.5 A Coordinated Air Vehicle Control Example

GCD-BB is developed to solve the coordinated air vehicle control problems in [20]. We introduce an example of this problem here, and report benchmark results in Chapter 5.

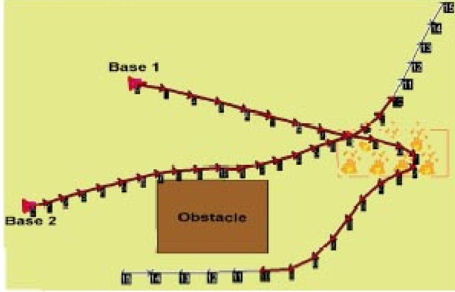

Figure 2-1: Map of the terrain for the fire-fighting example

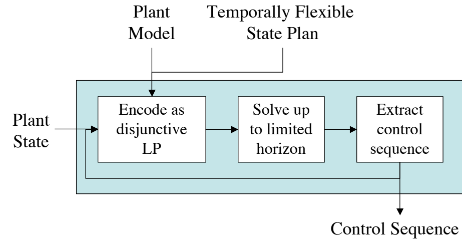

Figure 2-2: Receding horizon continuous planner

2.5.1 Problem Statement

The example introduced in [20] consists of two fixed-wing unmanned aerial vehicles (UAVs), whose state variables are their 2-D Cartesian positions and velocities. The UAVs are in an environment (Fig. 2-1) involving a reported fire that the team has to extinguish. To accomplish the task, the UAVs must navigate around unsafe regions, such as obstacles, and drop water on the fire. Once the fire is extinguished, they must also take pictures in order to assess the damage. A mission state plan specifies the desired evolution of the states of a dynamic system over time. The state plan for this fire-fighting mission is shown in Fig. 2-3:

> *Vehicles v_1 and v_2 must start at their respective base stations. v_1, a water tanker UAV, must reach the fire region and remain there for 5 to 8 time units, while it drops water over the fire. v_2, a reconnaissance UAV, must reach the fire region after v_1 is done dropping water and must remain there for 2 to 3 time units, in order to take pictures of the damage. The overall plan execution must last no longer than 20 time units.*

The problem of state execution is to generate a control trajectory that evolves the state of the vehicles according to the state plan (Fig. 2-3). State execution involves continuously planning control trajectory over a finite horizon, and then executing that trajectory.

As shown in Fig. 2-2, [20] solves the continuous planning problem up to a limited *planning horizon*, in order to generate a control sequence. It then executes that

sequence up to a shorter *execution horizon*, and solves the planning problem again at that time to adapt to disturbances. This approach achieves tractability by restricting the planner to a small planning window and also allows for on-line, robust adaptation to disturbances through continuous replanning. Finally, this short horizon and adaptation compensates for inaccuracies resulting from the linearization of the vehicle dynamics. [20] encodes the temporal state plan (Fig. 2-3) and the dynamics of the system within each limited horizon as a disjunctive linear program (DLP), so that the program can be solved iteratively to obtain a time or fuel optimal trajectory in the plant state space. Several variants of our GCD-BB algorithm have been tested to solve a range of these benchmark DLPs and have demonstrated significant improvement over BIP-BB.

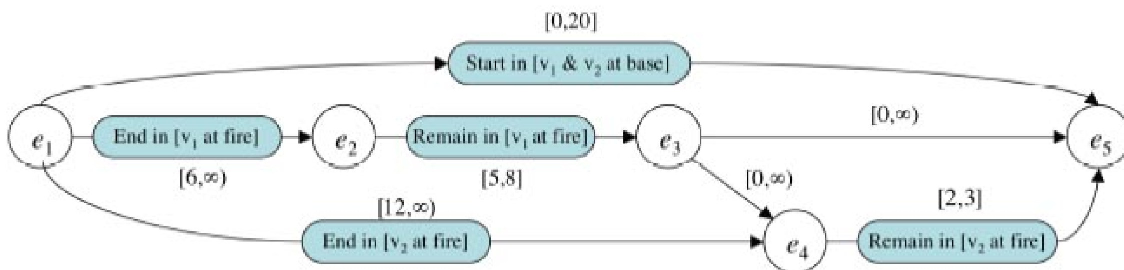

Figure 2-3: Temporally flexible state plan

2.5.2 DLP Encodings

In [20] plans and system dynamics are encoded in DLPs. The DLP encoded constraints are of two types: state plan constraints and plant model constraints, which include obstacle avoidance and system dynamics. The following example constraints are quoted from [20].

> *As for state plan constraints, consider the activity of imposing* $\boldsymbol{s} \in D_\forall$ *at the time* $T(e_E)$ *when event* e_E *is scheduled, where* $\boldsymbol{s}$ *is the vector of state variables, taking on values from the state space* $S \subset \mathbb{R}^n$*, and* $D_\forall$ *is the domain in* S *described by linear constraints on the state variables. In the*

fire-fighting scenario, this would be the constraint imposing v_2 *to be in the fire region at the event* e_4. *The general encoding is presented in Eq. 2.7, and translates to the fact that, either there exists a time instant of index* t *in the planning window that is* ΔT*-close to* $T(e_E)$ *and for which* $\boldsymbol{s}_t \in D_\forall$, *or event* e_E *must be scheduled outside of the current planning window.*

$$\begin{array}{cc} \bigvee_{t=0...N_t} & \left\{ \begin{array}{cc} & T(e_E) \geq T_0 + (t - \frac{1}{2})\Delta T \\ \wedge & T(e_E) \leq T_0 + (t + \frac{1}{2})\Delta T \\ \wedge & \mathbf{s}_t \in D_\forall \end{array} \right\} \\ \vee & T(e_E) \leq T_0 - \frac{\Delta T}{2} \\ \vee & T(e_E) \geq T_0 + (N_t + \frac{1}{2})\Delta T \end{array} \tag{2.7}$$

As for plant model constraints, Eq. 2.8 constrains $\mathbf{s}_t$ to be outside of each unsafe region for all t, where the unsafe regions are described by polyhedra. In the fire-fighting scenario, this corresponds to constraints encoding obstacle avoidance.

$$\bigwedge_{t=1...N_t} \bigvee_{i=1...n_{\mathcal{P}_S}} \mathbf{a}_i^T \mathbf{s}_t \geq b_i \tag{2.8}$$

Chapter 3

Technical Background

Recall that GCD-BB solves DLPs by combining Branch and Bound (B&B) [19] with the ideas drawn from Conflict-Directed A* search [35] and Activity Analysis [31]. We review each of these in the following sections.

3.1 Branch and Bound

Branch and Bound is frequently used to solve problems involving both discrete and continuous variables, such as MIPs and BIPs. It uses a 'divide and conquer' approach to explore the set of feasible integer solutions. However, instead of exploring the entire feasible set of a constrained problem, it uses bounds on the optimal cost, in order to avoid exploring subsets of the feasible set that it can prove are *sub-optimal*. A subproblem F_i of problem F is sub-optimal if the optimal solution to F_i is not better than the *incumbent*, which is the best feasible solution to F found so far. The generic B&B algorithm [3] is shown in Alg. 1.

3.1.1 Lower and Upper Bounds

It is important to have a relatively efficient way, for every F_i of interest, to compute a lower bound $lb(F_i)$ for its optimal cost. The basic idea is that while the optimal

Alg. 1 Branch-Bound(problem F)

```
1: incumbent U = +∞
2: select a subproblem F_i
3: if F_i is infeasible then
4:    delete F_i {prune the infeasible subproblem}
5: else
6:    compute the lower bound lb(F_i)
7:    if lb(F_i) ≥ U then
8:       delete F_i {prune the sub-optimal subproblem}
9:    else if the solution to F_i satisfies all the constraints of F then
10:      U ← lb(F_i)
11:   else
12:      break F_i into subproblems
13:   end if
14: end if
```

cost in a subproblem may be difficult to compute exactly, a lower bound may be more easily obtained. A general method to obtain such a bound is to use the optimal cost of the LP relaxation. p' is a relaxed LP of an optimization problem p, if the feasible region of p' contains the feasible region of p, and they have the same objective function. Therefore, if p' is infeasible, then p is infeasible. Assuming that we are performing minimization, if p' is solved with an optimal value v, the optimal value of p is guaranteed to be greater than or equal to v.

In the course of the B&B algorithm, we occasionally find that the optimal solution to a certain relaxed subproblem is also a solution to the original problem. This solution is used to maintain an upper bound U on the optimal cost of the original problem. In particular, U is the cost of the best feasible solution encountered thus far, called the *incumbent*. Given an incumbent and lower bounds formed from relaxed subproblems, tree pruning within B&B is based on the following observation. If the lower bound $lb(F_i)$ of a subproblem satisfies $lb(F_i) \geq U$, then this subproblem need not be considered further. This is because the optimal solution to the subproblem is no better than the incumbent.

3.1.2 An Example of B&B for BIPs

Fig. 3-1 shows a simple example of how B&B solves the BIP problem in Eq. 3.1.

$$\begin{aligned} \textit{Minimize} \quad & 3x_1 + 8x_2 \\ \textit{Subject to} \quad & x_1 + x_2 \geq 0.5 \\ & x_1, x_2 \in \{0, 1\} \end{aligned} \tag{3.1}$$

For BIP problems, B&B recursively partitions them into subproblems by assigning 0 and 1 to each binary variable, as shown in Fig. 3-1. A subproblem is defined by substituting for an assigned binary variable in the original problem. Subproblems are relaxed by turning binary variables into real-valued variables with domain $[0, 1]$. In Fig. 3-1 the relaxed subproblems are written next to their corresponding nodes $A-C$. The purpose of a relaxation is to generate a simple problem, whose solution offers a lower bound (assuming for minimization problems) on the solution to the original problem, to solve. For BIPs, a relaxed problem is an LP over real-valued variables, which can be solved using simplex.

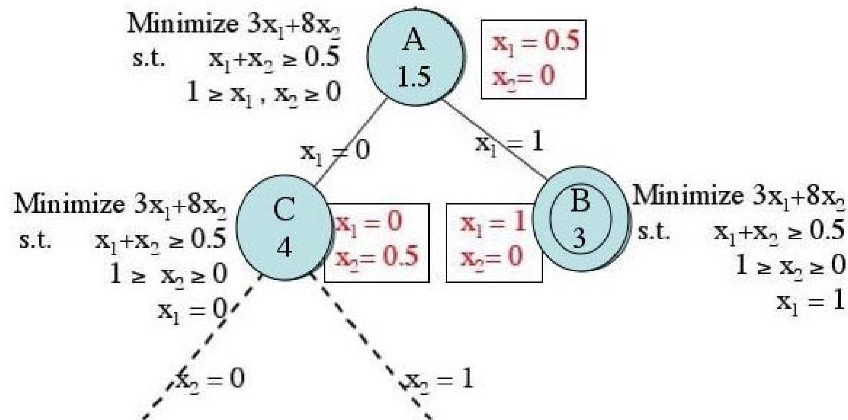

Figure 3-1: A simple example of B&B for BIP. Beside each node there is the corresponding relaxed LP to solve. The number in each node is the optimal cost. The optimal solution at each node is in a box. The node with double circles is an incumbent, and the dashed lines represent the subtrees that are pruned.

B&B typically explores a tree in depth first order. In the example (Fig. 3-1), once

the relaxed LP at node B is solved, its solution is found to be an incumbent. Node C is supposed to be expanded because its solution does not satisfy the binary variable x_2, however, it needs not to continue to break into subproblems as its lower bound is greater than the incumbent (3). Hence the subtree is pruned, and at the end of the search the incumbent proves to be the optimal solution.

3.2 Conflict-Directed A*

The Conflict-Directed A* (CD-A*) algorithm [35] utilizes the concept of a conflict to guide the search process to a solution of the optimal constraint satisfaction problem (OCSP). An OCSP is a multi-attribute decision problem whose decision variables are constrained by a set of finite domain constraints. For example, the task of identifying the most likely, consistent diagnoses of a circuit introduced in [35] is an OCSP. The circuit consists of three OR gates and two AND gates, as shown in Fig. 3-2 taken from [35]. Each component is in one of two possible modes, good (G) or broken (U). The decision variables are component mode variables, each over domain $\{G, U\}$. The relation of the components described in Fig. 3-2 defines the finite domain constraints on the decision variables. The attribute utilities are the component failure probabilities and are combined by multiplication, as we assume component failures are independent. If OR gates fail with probability 1% and AND gates with probability .5%, then the solution to the OCSP is $\{O1 = U, O2 = G, O3 = G, A1 = G, A2 = G\}$. A* search [10] uses an admissible heuristic to estimate the utility or cost of a state in the search space, and tests a sequence of candidate solutions in decreasing order of utility or increasing order of cost. The admissible heuristic corresponds to solving a relaxed problem in B&B. CD-A* differs from A* in that it uses the sources of *conflict*, identified within each inconsistent candidate, to jump over related candidates in the sequence. A conflict is any partial variable assignment that violates the OCSP constraints.

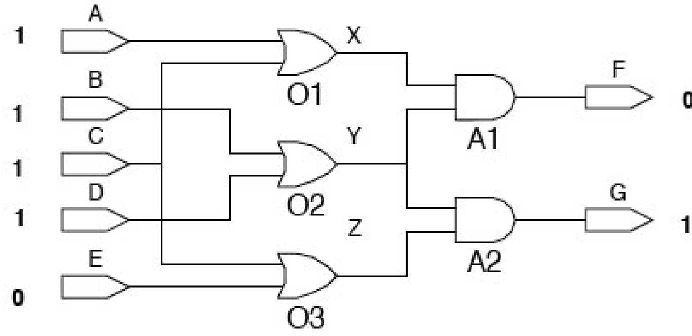

Figure 3-2: An OCSP example, with observed inputs and outputs indicated.

3.2.1 An Example of CD-A*

For example, consider Fig. 3-4 [35], which illustrates the search process of CD-A*. Whereas A* (as in Fig. 3-3 [35]) would search every single state in increasing heuristic cost order, CD-A* is able to identify regions that share infeasibilities and skip over all the states in these regions after exploring a single inconsistent state. In Fig. 3-4, CD-A* first selects the state with the lowest cost, S_1, which proves inconsistent. This inconsistency generalizes to Conflict 1, which eliminates states $S_1 - S_3$ (Fig. 3-4a). CD-A* then tests state S_4 which resolves Conflict 1. However, S_4 also proves inconsistent, and generalizes to Conflict 2, eliminating states $S_4 - S_7$ (Fig. 3-4b). Similarly, Conflict 3 (Fig. 3-4c) is generalized from inconsistent state S_8. Finally, the search tests state S_9 as consistent and returns it as an optimal solution (Fig. 3-4d).

3.2.2 The Process of CD-A*

CD-A* interleaves best-first generation and test. CD-A* generates as a candidate, the best valued decision state that resolves all discovered conflicts, by expanding a search tree which makes assignments to a set of decision variables of the OCSP. It tests each candidate S for consistency against the OCSP constraints. When S tests inconsistent, the inconsistency is generalized to one or more conflicts, denoting states that are inconsistent in a manner similar to S. The candidate is tested using any suitable CSP algorithm that extracts conflicts. CD-A* prunes discovered conflicts that are subsumed by other discovered conflicts, and then generates the next best

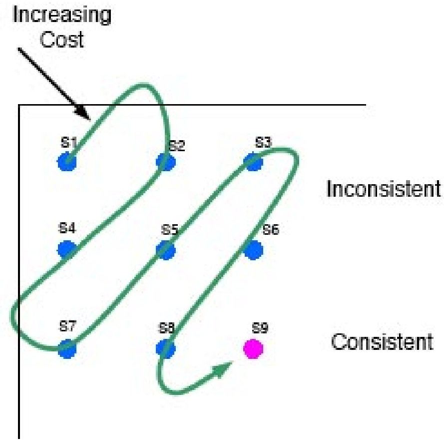

Figure 3-3: A* Search examines all best cost states leading up to the best consistent state.

candidate S' that resolves all conflicts discovered thus far. The process repeats until the desired leading solutions are found or all states are eliminated.

The key to CD-A* is the ability to efficiently generate, at each iteration, the next best candidate resolving all known conflicts. This is accomplished by mapping known conflicts to partial assignments, called *kernels*, and by extracting the kernel containing the best utility state. Each kernel describes a set of states that resolve the known conflicts. The mapping from conflicts to kernels consists of two steps. The first step generates *constituent kernels*. A *constituent kernel* is a *minimal* description of all states that resolve a particular conflict. The second step generates kernels, by computing the minimal set covering of the constituent kernels. In order to find the kernel with the best utility state, CD-A* views minimal set covering as a search and uses A* search to find the best kernel. The search tree for the example in Fig. 3-2 is shown in Fig. 3-5, where Conflict 1 is $\{A_1 = G, O_1 = G, O_2 = G\}$ and Conflict 2 is $\{A_1 = G, A_2 = G, O_1 = G\}$. A description of the CD-A* search tree is taken from [35].

> A tree node is expanded by selecting the constituent kernels of a conflict that is unresolved by that node, and by creating a child for each constituent kernel of that conflict. For example, the root node does not resolve Conflict 1 or 2. Selecting Conflict 1, the children of the root are

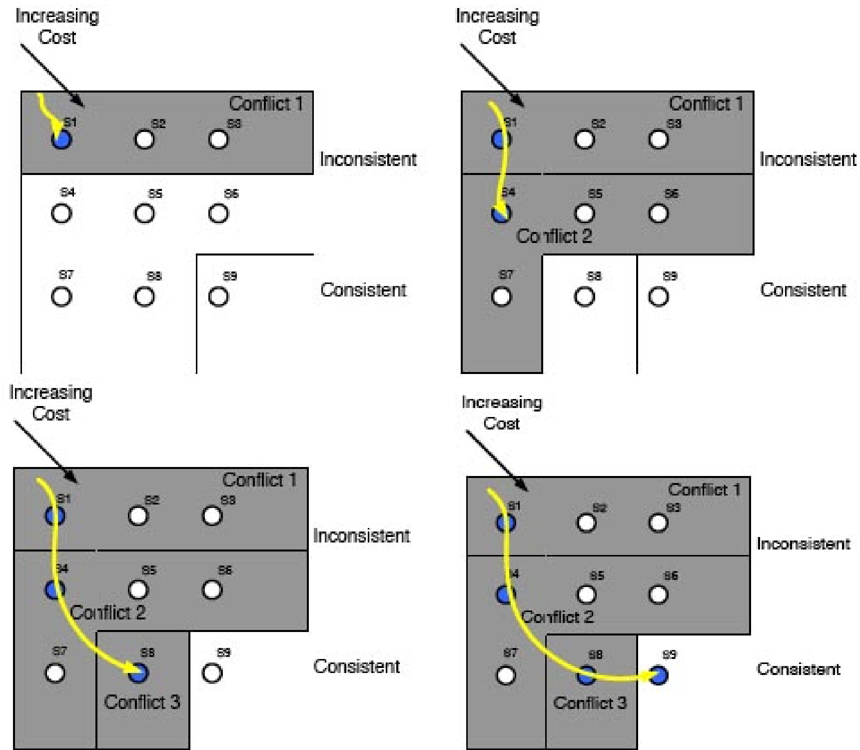

Figure 3-4: Conflict-directed A* focuses search using discovered conflicts. a) - d) represent snapshots along a prototypical search. Circles represent states. Filled in circles have been tested for consistency. Regions in grey have been ruled out by conflicts. Only state S9 is consistent.

$\{O_2 = U\}$, $\{O_1 = U\}$ and $\{A_1 = U\}$. Nodes are eliminated when non-minimal, such as the first and third leaves at the bottom left of the tree. Next, consider how the best candidate is extracted from a kernel. We generate the best candidate by assigning the remaining unassigned variables. To accomplish this we exploit a property called mutual, preferential independence (MPI). MPI says that to find the best candidate we assign each variable its best utility value, independent of the values assigned to the other variables. For example, initially there are no conflicts and the best kernel is the root node $\{\}$. For this kernel, Candidate 1 assigns the most likely value, G, to every variable, hence all components are working. Continuing the process, when Candidate 2 is generated (left, Fig. 3-6), only Conflict 1 has been discovered, hence the kernels correspond to the

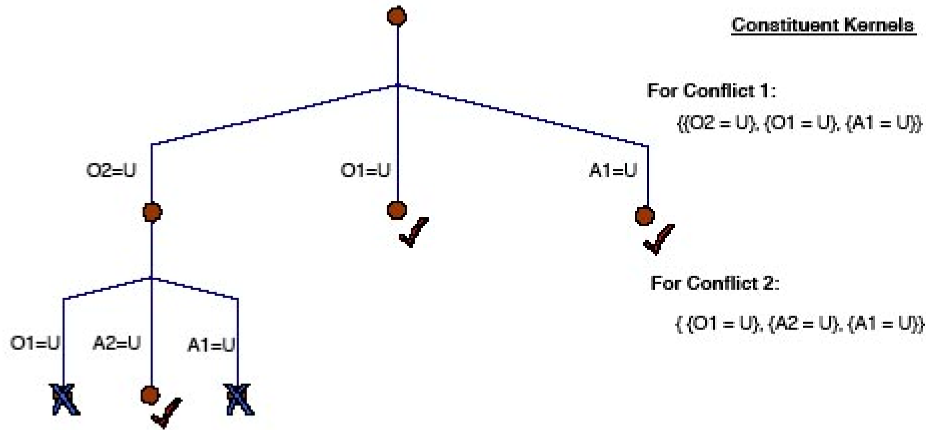

Figure 3-5: The search tree created by Conflict-directed A* to identify all kernels. Visited nodes that are kernels are check marked, while those that are not are crossed off.

constituent kernels of Conflict 1. Kernel $\{O_2 = U\}$ contains the most likely candidate. Its estimated probability combines the probability of $\{O_2 = U\}$, .01, with an optimistic estimate (i.e., admissible heuristic) of the best probability of the unassigned variables. By MPI, this heuristic selects the best utility value for each unassigned variable, .97, resulting in .0097, for the best candidate of $\{O_2 = U\}$.

A key property of the search is that it only expands the best valued child of $\{\}$, which is $\{O_2 = U\}$, rather than all children. This is valid because MPI guarantees that $\{O_2 = U\}$ contains a state whose utility is at least as good as that of every state contained by the other children, such as $\{O_1 = U\}$. The best kernel must be $\{O_2 = U\}$, or one of its descendants. $\{O_2 = U\}$ resolves the known conflicts, and hence is a kernel. To maximize utility, the kernels best candidate assigns G to the remaining components, that is, Candidate 2 has only O_2 broken.

When Candidate 3 is generated (right, Fig. 3-6), Conflict 1 and 2 have been discovered. Node $\{O_2 = U\}$ does not resolve Conflict 2, and is expanded by creating its best child $\{O_2 = U, O_1 = U\}$. This is a kernel, whose best candidate has probability $.01 \times .098 = .00098$.

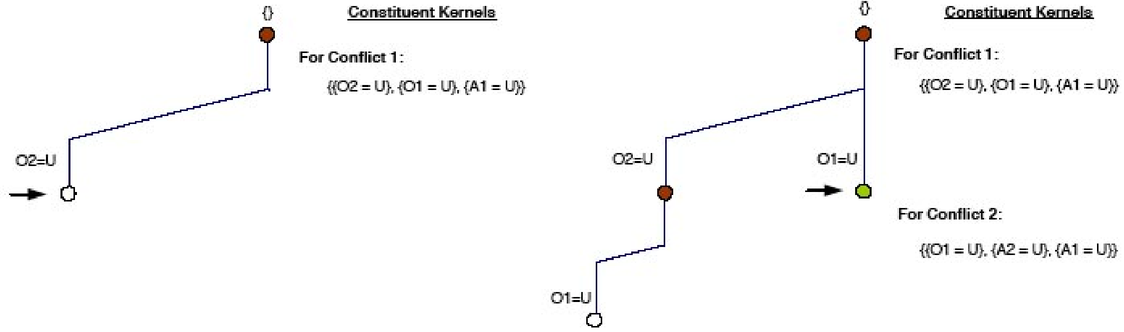

Figure 3-6: Left: Tree expansion for kernel $\{O_1 = U\}$, producing Candidate 2. Only the best valued child of the root is expanded, not all children. Right: Tree expansion for kernel $\{O_1 = U\}$, producing Candidate 3. When node $O_2 = U$ is expanded, its best child and its next best sibling are created.

> At this point it is no longer valid to just expand the best child of $\{O_2 = U\}$. Conflict 2 pruned out one or more of the states below node $\{O_2 = U\}$, hence we are no longer guaranteed that $\{O_2 = U\}$ contains a state that is as good as its sibling – this sibling may now contain the next best kernel. To achieve completeness we also expand its next best sibling, which is $\{O_1 = U\}$, with probability .0097. The next best sibling has higher probability than the best child, and hence the sibling is selected next. It is a kernel, and produces candidate 3, which is our most likely diagnosis.

Kernel generation of CD-A* described above is key to the forward conflict-directed search in GCD-BB. The forward conflict-directed search maps conflicts to kernels, by generating constituent kernels and computing the minimal set covering of the constituent kernels to form kernels. The difference is that best-first search is not used to identify the best kernel, because in order to do so we need to solve an LP for each kernel, which is very costly. Instead, we identify all the kernels that resolve all known conflicts, and prune those that are propositionally unsatisfiable before solving any LPs.

3.3 Activity Analysis

Activity Analysis (AA), as proposed in [31], is a technique that applies to the pervasive family of linear and non-linear, constrained optimization problems. It draws from the power of two seemingly divergent perspectives - the global conflict-based approaches of combinatorial satisfying search, and the local gradient-based approaches of continuous optimization - combined with the underlying insights of engineering monotonicity analysis [24, 23].

AA is used to help solve non-linear optimization problems. It strategically cuts away subspaces that it identifies as sub-optimal, and guides the numerical methods to the remaining subspaces. The power of eliminating large sub-optimal subspaces is derived from QKKT, an abstraction in *qualitative vector algebra* of the foundational Karush-Kuhn-Tucker (KKT) condition of optimization theory. The underlying algorithm achieves simplicity and completeness by introducing the concept of generating *prime implicating assignments* of linear, qualitative vector equations. Finally, AA can be considered as automating the underlying principle about monotonicity used by the simplex method to examine only the vertices of the linear feasible space. It then generalizes and applies this principle to non-linear programming problems.

The KKT conditions provide a set of vector equations that are satisfied for a feasible point x^* exactly when that point is stationary:

$$\begin{aligned} \bigtriangledown f(x^*) + \lambda^T \bigtriangledown h(x^*) + \mu^T \bigtriangledown g(x^*) &= 0^T \quad (KKT1) \\ \mu^T g(x^*) &= 0 \quad (KKT2) \\ \mu &\geq 0 \quad (KKT3) \end{aligned} \tag{3.2}$$

A key property of KKT is that it identifies *active* inequality constraints. Intuitively, a constraint $[g_i]$ is *active* at a point x when x is on the constraint boundary and the direction of decreasing objective, $\bigtriangledown f$, is pointing into the boundary. When this is true, μ_i is positive. Hence the basic approach of AA is to determine by looking at signs of μ, that the stationary points lie at the intersection of the constraint boundaries. The regions of the design space where optima can possibly lie is the

regions with only the stationary points. The regions with no stationary points are then sub-optimal regions to be eliminated.

The process of AA is the following. First, compute the signs of Jacobians $\bigtriangledown f$, $\bigtriangledown g$ and $\bigtriangledown h$, and expand QKKT1 by expanding matrix sums and products. Second, compute prime assignments P_i's and minimal set covering of the prime assignments $P_i \rightarrow P$, while deleting inconsistent assignments. Third, extract minimal sets of positive μ_i from P, and map the positive μ_i to $g_i(x) = 0$. Finally, formulate and return a new optimization problem.

Consider the hydraulic cylinder example in [31], the problem is formulated as Eq. 3.3. After instantiating QKKT1 (Step 1 of the AA process), prime assignments are computed (Eq. 3.4). The minimal set covering of $P(1)-(5)$ is $\{\{\lambda_1 = \hat{-}, \lambda_2 = \hat{+}, \mu_1 = \hat{+}, \mu_4 = \hat{+}\}, \{\lambda_1 = 0, \lambda_2 = \hat{+}, \mu_1 = \hat{+}, \mu_2 = \hat{+}, \mu_4 = 0\}\}$. Extracting the minimal sets of positive μ results in $\{\mu_1 = \hat{+}, \mu_4 = \hat{+}\}$ and $\{\mu_1 = \hat{+}, \mu_2 = \hat{+}, \mu_3 = \hat{+}\}$. Therefore two subspaces are found that could contain the optima, one subspace where g_1 and g_4 become strict equalities, and a second where all but g_4 become strict equalities. The new optimization problem produced is, to find x^* such that $x^* = argmin_{x \in F} f(x), F \in \{F_1, F_2\}$, where $F_1 =< \{g_2, g_3\}, \{h_1, h_2, g_1, g_4\} >$ and $F_2 =< \{g_4\}, \{h_1, h_2, g_1, g_2, g_3\} >$.

$$
\begin{aligned}
&Minimize && i + 2t \\
&Subject\ to && s - \frac{pi}{2t} = 0, && (h_1 = 0) \\
& && f - \frac{\pi i^2}{4} p = 0, && (h_2 = 0) \\
& && F - f \leq 0, && (g_1 \leq 0) \\
& && T - t \leq 0, && (g_2 \leq 0) \\
& && p - P \leq 0, && (g_3 \leq 0) \\
& && s - S \leq 0, && (g_4 \leq 0)
\end{aligned}
\tag{3.3}
$$

$$
\begin{aligned}
\{\lambda_1 = \hat{+}\} \bigvee \{\lambda_2 = \hat{+}\} \quad & P(1) \\
\{\lambda_1 = \hat{-}\} \bigvee \{\mu_2 = \hat{+}\} \quad & P(2) \\
\{\lambda_2 = 0, \mu_1 = 0\} \bigvee \{\lambda_2 = \hat{+}, \mu_1 = \hat{+}\} \quad & P(3) \\
\{\lambda_1 = 0, \mu_4 = 0\} \bigvee \{\lambda_1 = \hat{-}, \mu_4 = \hat{+}\} \quad & P(4) \\
\bigvee \{\lambda_1 = 0, \lambda_2 = 0, \mu_3 = 0\} \bigvee & \\
\{\lambda_1 = \hat{+}, \lambda_2 = \hat{-}\} \bigvee \{\lambda_1 = \hat{+}, \mu_3 = \hat{+}\} & \\
\{\lambda_1 = \hat{-}, \lambda_2 = \hat{+}\} \bigvee \{\lambda_2 = \hat{+}, \mu_3 = \hat{+}\} \quad & P(5)
\end{aligned}
\tag{3.4}
$$

Originally the problem has a 3 dimensional space to explore resulting from 5 variables and 2 equality constraints. AA rules out the interior and boundaries except some intersections. The first remaining subspace corresponds to a line, and the second remaining space is a point. Therefore, the complexity of the problem is significantly reduced.

GCD-BB is similar to AA in that it maps sub-optimal states to minimal set covering, and uses the covering to formulate and return a new optimization problem. However, GCD-BB differs from AA in that GCD-BB combines states from several sub-optimal problems through minimal set covering and uses the covering to expand a node in the B&B search tree, rather than being restricted in one optimization problem as in AA.

Chapter 4

The GCD-BB Algorithm

Recall that the GCD-BB algorithm builds upon B&B and incorporates three key innovative features: first, Generalized Conflict Learning, which learns abstractions (*conflicts*) comprised of constraint sets that produce either infeasibility or sub-optimality; second, Forward Conflict-Directed Search, which guides the forward step of the search away from regions of state space corresponding to known conflicts, and third, Induced Unit Clause Relaxation, which uses unit propagation to form a relaxed problem. In addition, we compare the influence of different search orders: Best-first Search (BFS) versus Depth-first Search (DFS). In the following sections, we develop these key features of GCD-BB in detail, including examples and pseudo code.

4.1 Branch and Bound for DLPs

B&B, an algorithm to solve problems involving both discrete and continuous variables, is frequently used by algorithms to solve BIPs. Instead of branching by assigning 0 and 1 to each binary variable, as for BIPs, 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 disjuncts of the clause for each of the child nodes. An example search tree is shown in Fig. 4-1. The node on the bottom left is created from its parent node by selecting the disjunct $g_{21}(x) \leq 0$ from its clause.

BB-DLP (Alg. 2) results from applying the generic B&B algorithm (Alg. 1) to

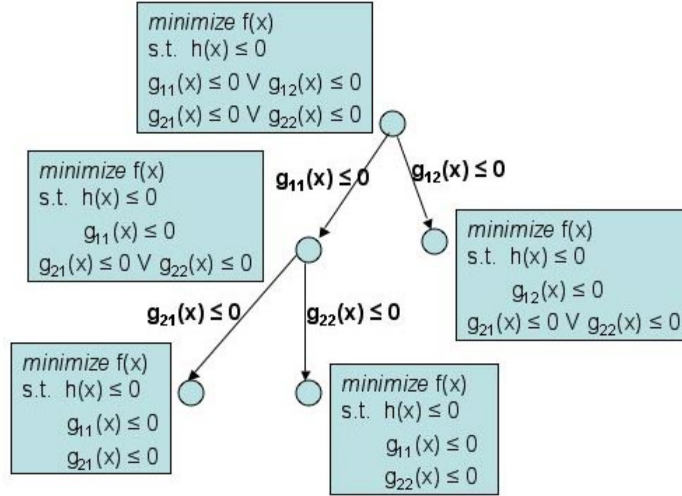

Figure 4-1: The search tree of B&B for DLPs branches by splitting clauses. Each node represents a DLP.

DLPs. Each node in the search tree represents a DLP. The root represents the original DLP, and its descendents represent subproblems of the original DLP. In the algorithm, each node has an associated subproblem, its relaxation and its relaxed solution. More precisely, each node is comprised of: 1) the objective function of the DLP (*node*.objective), for example, "*minimize* $f(x)$" in the root node in Fig. 4-1, 2) the unit clause set of the DLP (*node*.unitClauses), for example, $h(x) \leq 0$ in the root node, and 3) the non-unit clause set of the DLP (*node*.nonUnitClauses), for example, $\{g_{11}(X) \leq 0 \vee g_{12}(x) \leq 0,\ g_{21}(X) \leq 0 \vee g_{22}(x) \leq 0\}$. As with traditional B&B, a node also includes a relaxed LP (*node*.relaxedLP) formed from the DLP using some relaxation method (Section 4.4), the optimal solution to the relaxed LP (*node*.relaxedSolution) and its optimal value (*node*.relaxedValue).

GCD-BB performs a novel relaxation of a DLP subproblem through propositional logic. In particular, the relaxation is constructed from the original DLP as follows: each linear inequality in the original DLP, either in the unit clause set or in the non-unit clause set, is associated with a propositional symbol. Unique inequalities have unique symbols, repeated ones share the same symbol, and if one inequality is the negation of another then $\neg$ is added. For example, $\{x \leq 200, x \geq 200 \vee y \leq 100, y \leq$

$100 \vee x + 2y \geq 10\}$ can be represented as $\{a, \neg a \vee b, b \vee c\}$. The propositional symbols will be used in the next three sections, when the form of inequalities is not needed for our algorithm.

Alg. 2 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 Violated-Clause?(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
```

Similar to standard B&B, BB-DLP performs search in depth-first order (line 3 Alg. 2). At each node a relaxed LP is solved (line 6). If a node is feasible (line 9) and better than the incumbent (line 11), it is tested (line 13-18) whether further expansion

Alg. 3 Violated-Clause?(*clause*, *solution*)

```
1: for each disjunct in clause do
2:    if solution satisfies disjunct then
3:       return False
4:    end if
5: end for
6: return True
```

is needed, using Violated-Clause? (Alg. 3). For example, if the node does not need expansion (line 19), it is identified as the new incumbent; otherwise, Expand-Node (Alg. 4) is called, which creates the children of the node and places them in the queue (line 23).

Finally, GCD-BB needs to know the order in which nodes are created, in order to perform forward conflict-directed search (Section 4.3). To support this a timestamp is used to mark the creation time of a node (*node*.timestamp). Every time node expansion occurs, the timestamp is incremented by 1 (line 24). Note that timestamps are also maintained for conflicts, as introduced in Section 4.2 and then later exploited in Section 4.3.

Alg. 4 Expand-Node(*node*, *timestamp*)

```
1: for each clause in node.nonUnitClauses do
2:    if Violated-Clause?(clause, node.relaxedSolution) then
3:       add clause to sortList {sortList contains violated clauses in increasing order
         of their number of disjuncts}
4:    end if
5: end for
6: selectedClause ← sortList(first)
7: for each disjunct in selectedClause do
8:    child.unitClauses ← node.unitClauses + disjunct
9:    child.nonUnitClauses ← node.nonUnitClauses - selectedClause
10:   child.timestamp ← timestamp + 1
11:   add child to childList
12: end for
13: return childList
```

For the function Violated-Clause? (Alg. 3), we define a clause to be *violated* by a relaxed solution, if the solution satisfies none of the linear constraints (disjuncts) in

the clause.

Consider the function Expand-Node (Alg. 4). In the spirit of the most constrained variable heuristic of CSPs [4], Expand-Node splits on a violated clause with the smallest number of disjuncts. To find this clause, we maintain a list, sortList (line 1-5), which contains the violated clauses in increasing order of their number of disjuncts. Therefore, sortList(first) is the clause with the least number of disjuncts. For each disjunct in the selected clause, a child is created (line 7-12). Finally, the list of children are returned.

B&B for DLPs differs from B&B for BIPs in three respects. First, it forms relaxed LPs from unit clauses (see Section 4.4), rather than extending the domain of binary variables to the real-valued interval [0,1]. Second, the condition for performing node expansion is the existence of violated clauses, rather than unsatisfied (real-valued) binary variables. Finally, it expands nodes by splitting clauses, rather than assigning binary variables 0 and 1.

4.2 Generalized Conflict Learning

In the related field of discrete constraint satisfaction, conflict learning methods, such as dependency-directed backtracking [28], backjumping [12], conflict-directed backjumping [25] and dynamic backtracking [13], dramatically improve the performance of backtrack (BT) search, by learning the source of each inconsistency discovered, and by using this information, called a *conflict* (or *nogood*), to prune additional subtrees that the conflict identifies as inconsistent.

To apply conflict learning to B&B, we note that B&B prunes subtrees corresponding to relaxed subproblems that are either infeasible or sub-optimal (line 8 and 11 in Alg. 2). Hence two opportunities exist for learning and pruning. We generalize conflict learning and pruning, in contrast to previous work, in that conflicts are extracted from both sub-optimal and infeasible subproblems. To accomplish this we add functions Extract-Infeasibility (Alg. 5) and Extract-Suboptimality (Alg. 6) after line 7 and 10 in BB-DLP (Alg. 2), 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.

4.2.1 Conflicts

In the context of DLP, each *conflict* can be one of two types: an *infeasibility conflict*, or a *sub-optimality conflict*.

Definition 4.2.1 Given a DLP $= < \mathbf{x}, f, C >$, where $\mathbf{x} \in \mathbb{R}^n$ is a vector of variables, f is the minimizing objective function over $\mathbf{x}$ and C is the constraint set over $\mathbf{x}$, $\alpha \subseteq$ disjuncts(C), where disjuncts(C) is the set of all the linear inequalities of C, is an *infeasibility conflict* of the DLP if $\neg\exists\mathbf{x}$ s.t. $\alpha(x)$ is satisfied.

For example, in the DLP shown in Eq. 4.1, the unit clause set $\{x \leq 200, y \leq 200, x \leq 10, x \geq 80\}$ is an infeasibility conflict, since the constraints are not satisfiable for any value of x.

Definition 4.2.2 Given a DLP $= < \mathbf{x}, f, C >$ and $\mathbf{x}^* \in \mathbb{R}^n$ s.t. $C(\mathbf{x}^*)$ is satisfied, $\alpha \subseteq$ disjuncts(C) is a *sub-optimality conflict* of the DLP if $\forall \mathbf{y} \in \mathbb{R}^n$ s.t. $\alpha(\mathbf{y})$ is satisfied, $f(\mathbf{y}) \geq f(\mathbf{x}^*)$.

For example, the DLP shown in Eq. 4.2 be a subproblem of the DLP shown in Eq. 4.3. The best feasible solution to the DLP in Eq. 4.3 found so far has value smaller than -100, and -100 is the optimal value of the relaxed LP of Eq. 4.2. The unit clause set $\{x \leq 200, y \leq 200, x \leq 100, y \leq 0\}$ is then a sub-optimality conflict.

$$\begin{aligned} Minimize \;\; & -x - 3y \\ Subject\ to \;\; & x \leq 200 \\ & y \leq 200 \\ & x \leq 10 \\ & x \geq 80 \\ & x \leq 100 \vee y \leq 50 \end{aligned} \tag{4.1}$$

$$
\begin{aligned}
Minimize \; & -x - 3y \\
Subject\ to \; & x \leq 200 \\
& y \leq 200 \\
& x \leq 100 \\
& y \leq 0 \\
& x \leq 10 \lor y \leq 5 \lor y \leq 4
\end{aligned} \tag{4.2}
$$

$$
\begin{aligned}
Minimize \; & -x - 3y \\
Subject\ to \; & x \leq 200 \\
& y \leq 200 \\
& x \leq 100 \lor y \leq 50 \\
& y \leq 0 \lor x \geq 80 \lor x \geq 30 \\
& x \leq 10 \lor y \leq 5 \lor y \leq 4
\end{aligned} \tag{4.3}
$$

4.2.2 Minimal Conflicts

Similar to the above definitions, a *minimal conflict* can be one of two types: a *minimal infeasibility conflict*, or a *minimal sub-optimality conflict*.

Definition 4.2.3 Given a DLP $= < \mathbf{x}, f, C >$, $\alpha \subseteq$ disjuncts(C) is a *minimal infeasibility conflict* of the DLP if α is an infeasibility conflict and $\neg\exists\delta \subset \alpha$ s.t. δ is an infeasibility conflict.

For example, the constraint set $\{x \leq 10, x \geq 80\}$ in the DLP of Eq. 4.1 is a minimal infeasibility conflict, since it is an infeasibility conflict and any proper subset of it is not an infeasibility conflict.

Definition 4.2.4 Given a DLP $= < \mathbf{x}, f, C >$, $\alpha \subseteq$ disjuncts(C) is a *minimal sub-optimality conflict* of the DLP if α is a sub-optimality conflict and $\neg\exists\delta \subset \alpha$ s.t. δ is a sub-optimality conflict.

Likewise, the constraint set $\{x \leq 100, y \leq 0\}$ in the DLP of Eq. 4.2 is a minimal sub-optimality conflict, with respect to the feasible point $(100, 0)$. Note that there can be more than one minimal conflict (possibly with different cardinalities) involved in one infeasibility or sub-optimality. In addition 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. We do not try to extract the minimum conflict of a subproblem, because the computational cost of searching for this conflict is prohibitive.

4.2.3 Conflict Extraction

To perform generalized conflict learning efficiently, we introduce two novel methods based on the duality theory [3] to extract a subproblem's minimal conflict. Recall that in the Branch and Bound search tree, a relaxed LP is solved at each node. We run the dual simplex method as the LP solver. For infeasibility, when dual simplex terminates unbounded, an extreme ray is discovered with it. The non-zero elements of the extreme ray are used to identify the constraints of the minimal infeasibility conflict. The proof is in the working paper [21].

For sub-optimality, when dual simplex terminates with an optimal solution, we can choose to extract the minimal sub-optimality conflict. To accomplish that, we reduce the constraint matrix so that there are no duplicate constraints and examine the consequent dual solution vector. Suppose the reduced constraint matrix has m rows and n columns. Then the corresponding dual vector has m elements. If there are n or less non-zero elements in the dual vector, all the non-zero elements of the dual vector correspond to the constraints of the minimal sub-optimality conflict; otherwise, *any* n of the non-zero elements are used to identify the minimal conflict.

The principle is explained as follows. According to Complementary Slackness [3] from linear optimization theory, the non-zero terms of the optimal dual vector correspond to the set of *active* constraints S at a given optimal solution to the LP. An inequality constraint $g_i(x) \leq 0$ is active at a feasible point $\tilde{x}$ if $g_i(\tilde{x}) = 0$. According to Definition 4.2.4, all the linear inequalities in a minimal sub-optimality conflict must

be in S. That is, if $g_i(x) < 0$ at the optimum, then removing the constraint does not alter the optimum. A solution $x \in \mathbb{R}^n$ is *degenerate* if more than n constraints are active at x. When x is non-degenerate and unique, exactly n constraints are active at x and removing any one of them alters the solution. When x is non-degenerate and non-unique, there are less than n constraints active at x and removing any one of them alters the solution. When x is degenerate and unique, more than n constraints are active at x and as long as n of them are kept active the solution x remains the same. For example, $(0, 5)$ is a degenerate and unique solution in Fig. 4-2, and any 2 of the constraints form a minimal sub-optimality conflict. The case when x is degenerate and non-unique does not exist for our reduced constraint matrix.

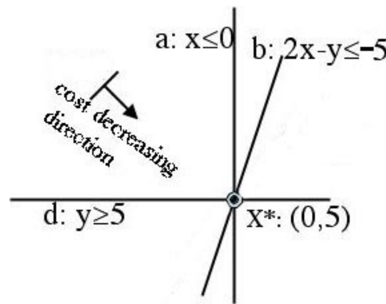

Figure 4-2: An example of a degenerate unique optimum.

Functions Extract-Infeasibility and Extract-Suboptimality are shown in Alg. 5 and Alg. 6, respectively. Note that in Extract-Suboptimality (Alg. 6) $|activeSet|$ can be less than n when the optimum is not unique (line 2). In both functions, an extracted minimal conflict is stored as a set of propositional symbols in a conflict database (conflictDB). A conflict is indexed by a *timestamp*, which marks the conflict′s discovery time.

Alg. 5 Extract-Infeasibility

1: $minimalConflict \leftarrow$ cplex.getIIS()
2: put $minimalConflict$ in conflictDB($timestamp$)

To summarize, this section introduced the concepts of conflicts and minimal conflicts, and described the efficient approach used in GCD-BB to extract minimal con-

Alg. 6 Extract-Suboptimality

1: $activeSet \leftarrow$ cplex.getDuals() {dual vectors are computed using CPLEX}
2: **if** |activeSet| $\leq n$ **then**
3: $minimalConflict \leftarrow$ activeSet
4: **else**
5: put n constraints from activeSet in $minimalConflict$
6: **end if**
7: put $minimalConflict$ in conflictDB($timestamp$)

flicts.

4.3 Forward Conflict-directed Search

We use forward conflict-directed search to heuristically guide the forward step of search away from regions of the state space that are ruled out by known conflicts. Backward search methods also use conflicts to direct search, such as dependency-directed backtracking [28], backjumping [12], conflict-directed backjumping [25], dynamic backtracking [13] and LPSAT [36]. These backtrack search methods use conflicts to select backtrack points and as a cache to prune nodes without testing consistency. We use conflicts in forward search, as in conflict-directed A* (CD-A*) search [35], to move away from known "bad" states. We generalize the approach in CD-A* to guiding B&B away from regions of state space that the known conflicts indicate are infeasible or sub-optimal. To accomplish this, a node is expanded so that each of its children resolves all the node′s unresolved conflicts. A node *resolves* a conflict if at least one of the conflict′s disjuncts is explicitly excluded from the relaxed LP of the node.

In terms of implementation, we replace function Expand-Node in line 24 of BB-DLP (Alg. 2) with function General-Expand-Node (Alg. 7). Our experimental results on a range of cooperative vehicle plan execution problems show that forward conflict-directed search significantly outperforms backtrack search with conflicts (Section 5.1.2).

In General-Expand-Node (Alg. 7), when there is no unresolved conflict, Expand-

Alg. 7 General-Expand-Node($node, timestamp$,conflictDB)

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

Node (Alg. 4) is used, and when unresolved conflicts exist, Forward-CD-Search (Alg. 8) is performed. Next we elaborate upon the concepts in forward conflict-directed search, and then present the detailed algorithm.

Recall from Section 3.2.2, that 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 constraint contained in the conflict. For example, one constituent kernel of the minimal infeasibility conflict in Eq. 4.1 is $\{x \geq 10\}$.

Given the set of constituent kernels, recall that CD-A* generates kernels, each of which resolves 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, a kernel corresponds to a set of linear inequalities. Extending a node with the kernels of the unresolved conflicts guarantees that all known conflicts are resolved by the node and its descendants. For DLPs we build up kernels similar to CD-A*, by combining constituent kernels using minimal set covering. However, unlike CD-A* 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. 4-3, and prunes the DLPs that are propositionally unsatisfiable, using a fast unit propagation test before solving any relaxed LP.

Forward-CD-Search (Alg. 8) includes three steps: 1) Generate-Constituent-Kernels (Alg. 9), 2) Generate-Kernels (Alg. 10) and 3) Generate-And-Test-DLP-Candidates

Alg. 8 Forward-CD-Search($node, conflictSet$)

1: $constituentKernelSet \leftarrow$ Generate-Constituent-Kernels($conflictSet$)
2: $kernelSet \leftarrow$ Generate-Kernels($constituentKernelSet$)
3: return $DLPList \leftarrow$ Generate-And-Test-DLP-Candidates($kernelSet, node$)

(Alg. 14). An example is shown in Fig. 4-3.

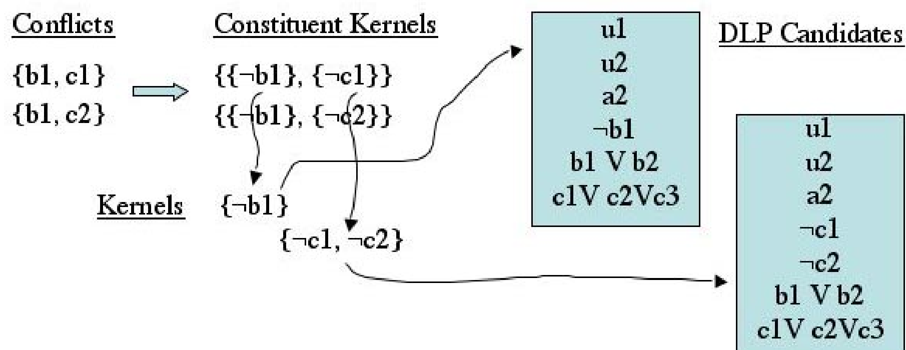

Figure 4-3: Each conflict is mapped to a set of constituent kernels, which resolve that conflict alone. Kernels are generated by combining the constituent kernels using minimal set covering. A DLP candidate is formed for each kernel, and is checked for consistency.

Alg. 9 Generate-Constituent-Kernels($conflictSet$)

1: **for** each c in $conflictSet$ **do**
2: $K_c \leftarrow \{\}$
3: **for** each $constraint$ in c **do**
4: $K_c \leftarrow K_c \cup \{\neg constraint\}$
5: **end for**
6: add K_c to $constituentKernelSet$
7: **end for**
8: return $constituentKernelSet$

To generate constituent kernels for a conflict, Generate-Constituent-Kernels (Alg. 9) forms a set by negating each constraint in the conflict (line 3-5), and then collects the constituent kernel set for every conflict.

Once the constituent kernels for all the conflicts are generated, we use minimal set covering to generate the kernels; this is performed by Generate-Kernels (Alg. 10).

Alg. 10 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. 11 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. 4-4(b) demonstrates Generate-Kernels by continuing the example from Fig. 4-3. In particular, in Fig. 4-4(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. 10), 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. 4-4(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. 4-4(b) resolves all the conflict and, therefore, is not expanded.

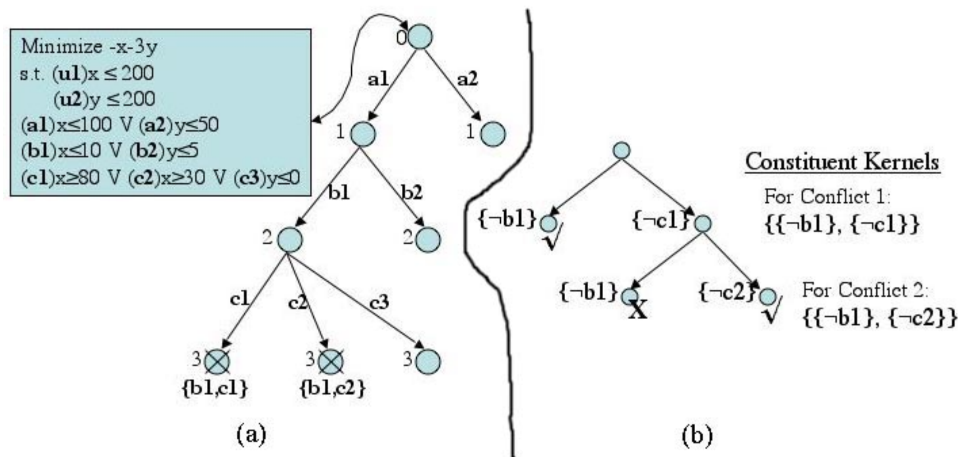

Figure 4-4: (a) A partial tree of B&B for DLPs. The creation time of each node is shown on the left of the node. Two conflicts are discovered at the bottom. (b) The search tree for minimal set covering to generate kernels from constituent kernels.

Finally, consider the use of timestamps. Recall that 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, because the node contains at least one

Alg. 12 Unresolved-Conflict?($node, node$.unresolved)

```
1: for each E in node.unresolved do
2:    if E ∩node = ∅ then
3:       return True {If node contains no constituent kernel of this conflict, then it
         does not resolve the conflict.}
4:    end if
5: end for
6: return False
```

Alg. 13 Expand-Conflict($node, node$.unresolved)

```
1: for each E in node.unresolved do
2:    if E ∩node ≠ ∅ then
3:       remove E from node.unresolved {removes the constituent kernel set of the
         resolved conflict}
4:    end if
5: end for
6: X ← the smallest set in node.unresolved
7: for each F in X do
8:    child ← node ∪ {F}
9:    add child in childList {Each child is created by selecting one of the
      constituent kernels of an unresolved conflict}
10: end for
11: return childList
```

Alg. 14 Generate-And-Test-DLP-Candidate($kernelSet, DLP$)

```
1: S ← DLP.unitClauses
2: for each kernel in kernelSet do
3:    if Consistent?(S ∪ kernel) then
4:       DLP.unitClauses ← S ∪ 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
```

Alg. 15 Consistent?(Set)

```
1: for each e in Set do
2:    if Set ∩ {¬e} ≠ ∅ then
3:       return False
4:    end if
5: end for
6: return True
```

of the constituent kernels of the conflict, and the node's descendants all retain that constituent kernel. For example, in Fig. 4-4, 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 {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 (Alg. 8). 3. If {conflict time $<$ node time}, the conflict is guaranteed to be resolved by an ancestor node of the current node, and therefore, the conflict does not need to be resolved again.

This section introduced the search method in GCD-BB, *forward conflict-directed search*, which guides B&B away from regions of state space that the known conflicts indicate as infeasible or sub-optimal, through generating constituent kernels, kernels and DLP candidates from conflicts.

4.4 Induced Unit Clause Relaxation

Recall from Section 3.1.1 that relaxation is an essential tool for quickly characterizing a problem when the original problem is hard to solve directly. Relaxation provides bounds on feasibility and the optimal value of a problem, which are commonly used by B&B to prune the search space. Previous research [15] typically solves DLPs by reformulating them as BIPs, where a relaxed LP is formed by relaxing the binary constraints ($x \in \{0, 1\}$) to a corresponding set of 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 (a unit clause is one that

contains a single constraint). The rationale in [15] for reformulating a DLP as a BIP relaxation, is that it maintains some of the constraints of the non-unit clauses through the continuous relaxation from binary to real-valued variables; this is opposed 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. For example, consider the DLP in Eq. 4.4 and its equivalent BIP in Eq. 4.5. The relaxed LP of the DLP is shown in Eq. 4.6, and is formed by ignoring all non-unit clauses. The relaxed LP of the BIP is in Eq. 4.7. Eq. 4.7 is a more constrained LP and therefore, a better relaxation than Eq. 4.6. However, the size of the LP in Eq. 4.6 is significantly smaller than that of Eq. 4.7, in terms of the number of variables and the number of constraints.

$$\begin{aligned}
Minimize &\ -x-3y \\
Subject\ to &\ x \leq 200 \\
&\ y \leq 200 \\
&\ x \leq 100 \\
&\ x \geq 200 \vee y \leq 100 \\
&\ y \leq 100 \vee x \leq 50 \vee x \geq 300
\end{aligned} \tag{4.4}$$

$$
\begin{aligned}
Minimize\ & -x-3y \\
Subject\ to\ & x \le 200 \\
& y \le 200 \\
& x \le 100 \\
& x - 200 \ge M(b_1 - 1) \\
& y - 100 \le M(1 - b_2) \\
& x - 50 \le M(1 - b_3) \\
& x - 300 \ge M(b_4 - 1) \\
& b_1 + b_2 \ge 1 \\
& b_2 + b_3 + b_4 \ge 1 \\
& b_1, \ldots, b_4 \in \{0, 1\}
\end{aligned} \tag{4.5}
$$

$$
\begin{aligned}
Minimize\ & -x-3y \\
Subject\ to\ & x \le 200 \\
& y \le 200 \\
& x \le 100
\end{aligned} \tag{4.6}
$$

$$
\begin{aligned}
Minimize \;\; & -x-3y \\
Subject\ to \;\; & x \leq 200 \\
& y \leq 200 \\
& x \leq 100 \\
& x-200 \geq M(b_1-1) \\
& y-100 \leq M(1-b_2) \\
& x-50 \leq M(1-b_3) \\
& x-300 \geq M(b_4-1) \\
& b_1+b_2 \geq 1 \\
& b_2+b_3+b_4 \geq 1 \\
& 0 \leq b_1,\ldots,b_4 \leq 1
\end{aligned} \tag{4.7}
$$

Our approach leverages the reduced state space, by starting 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. For example, our relaxation of the DLP in Eq. 4.4 is Eq. 4.8, which provides a better relaxation than Eq. 4.5 and has a smaller problem size than Eq. 4.7.

$$
\begin{aligned}
Minimize \;\; & -x-3y \\
Subject\ to \;\; & x \leq 200 \\
& y \leq 200 \\
& x \leq 100 \\
& y \leq 100
\end{aligned} \tag{4.8}
$$

This relaxation is defined by Induce-Unit-Clause (Alg. 16) and demonstrated in Fig. 4-5. In the experiment section we compare our induced unit clause relaxation with BIP relaxation, and show a profound improvement in runtime on a range of cooperative vehicle plan execution problems (Section 5.1.2).

Recall that in BB-DLP (Alg. 2) the method to obtain *node*.relaxedLP is not men-

tioned. To incorporate induced unit clause relaxation, we add function Induce-Unit-Clause after line 5 in Alg. 2. Induce-Unit-Clause (Alg. 16) performs unit propagation among the unit and non-unit clauses to induce more unit clauses. Next we elaborate

---

Alg. 16 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

---

on Induce-Unit-Clause (Alg. 16) by walking through the example DLP in Eq. 4.4, using Fig. 4-5. Recall from Section 4.1 that, in the original DLP, each linear inequality

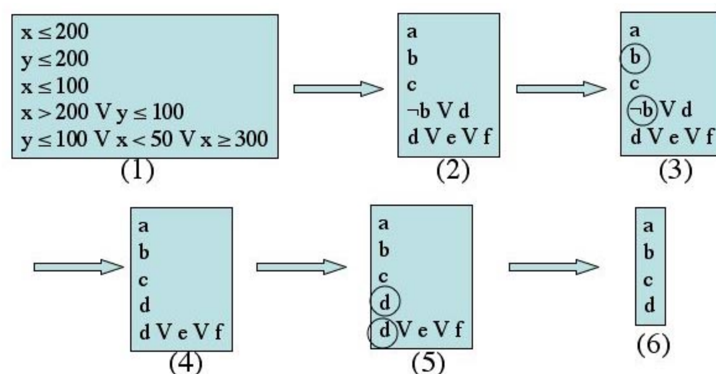

Figure 4-5: An example of induced unit clause relaxation: from Eq. 4.4 to Eq. 4.8

is associated with a propositional symbol. Repeated linear inequalities are given the same propositional symbol and negated linear inequalities differ from each other by ¬, as shown in Steps (1)-(2) of Fig. 4-5.

Induce-Unit-Clause simplifies a DLP by performing unit propagation on the propositional clause set of the DLP, and then reflecting the consequences of the propagation back on the original DLP. More specifically, in line 1 of Alg. 16, Induce-Unit-Clause calls function Unit-Propagation to simplify the unit and non-unit clause sets of a DLP (Steps (2)-(5) in Fig. 4-5). A relaxed LP is also formed by combining the objective function and the unit clause set (line 2).

This section introduced induced unit clause relaxation as our approach to form

relaxed LPs from DLPs using unit propagation, and demonstrated the advantages over other relaxation methods.

4.5 Search Order: Best-first versus Depth-first

Given a fixed set of heuristic information, [9] 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 is significantly reduced. Our experimental results show that BFS can take memory space similar to DFS, while taking significantly less time to find the optimum (Section 5.1.3).

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 sub-optimality (Section 5.1.2).

To summarize, this chapter introduced a novel algorithm, *Generalized Conflict-Directed Branch and Bound*, for solving DLPs. It extends traditional Branch and Bound, by first constructing a conflict from each search node that is discovered 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. This is accomplished through three main features: *generalized conflict learning*, *forward conflict-directed search* and *induced unit clause relaxation.*

Chapter 5

Evaluation and Discussion

This chapter provides experimental results of the GCD-BB algorithm on a range of test problems, for coordinated air vehicle control [20]. GCD-BB is compared with the benchmark B&B algorithm applied both to DLPs and their equivalent BIP encoding. We also compare the effect of several algorithmic variants, in particular, infeasibility conflict learning versus sub-optimality conflict learning, forward conflict-directed search versus backtrack conflict-directed search, and BFS versus DFS. While each algorithmic variant terminates with the same optimal solution, GCD-BB achieves an order of magnitude speed-up over BIP-BB. Our experiments show that the elements of the algorithm that are most important with respect to achieving this performance are: generalized conflict learning, forward conflict-directed search and induced unit clause relaxation. The evaluation is followed by a discussion about future work.

5.1 Empirical Evaluation

As the bulk of the computational effort expended by HDLOP B&B 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 LPs 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 variants in Java. All algorithms used the commercial software CPLEX as the LP solver. Test problems were generated using the model-based temporal planner [20] discussed in Section 2.5, on the performance of multi-vehicle search and rescue missions. Recall that 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.

5.1.1 The Average LP Size

Each algorithm solves a number of relaxed LPs before reaching the final optimal solution. Table 5.1 compares the average size of these relaxed LPs, in terms of the number of constraints in each relaxed LP, solved by each algorithm. Each column represents a set of test problems with similar dimensionality. For example, the first column 80/36 represents a set of test problems that all have 80 clauses and 36 variables. Each row specifies performance for a particular algorithm, in terms of the average size of relaxed LPs solved for each test problem set. Note that the three tables in this chapter use the same format.

From the table we make three observations. First, the average size of LPs solved in BIP-BB is larger than that of the LPs solved by any algorithm for DLPs. Second, the difference increases from about 20% to about 40% as the test problem size increases. Finally, the average size of LPs solved by each DLP algorithm variant is similar to one another. These observations agree with the theoretical comparison of the BIP and DLP formulations, in Section 2.1 and 2.2, in terms of state space size.

Table 5.1: 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

5.1.2 The Total Number of LPs

Table 5.2 records the number of relaxed LPs solved by each algorithm. In the following subsections, we use this table, first, to show the reason for using conflict learning (Section 4.2). Second, to show the reason for using forward conflict-directed search instead of conflict-directed backtrack search (Section 4.3). Third, to compare BIP and DLP encodings (Section 4.4), and finally, to address the tradeoffs of BFS and DFS (Section 4.5).

Table 5.2: 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

Generalized Conflict Learning

In order to show the reason for using conflict learning, we compare row "DLP BFS without Conflict Learning" with row "DLP BFS Infeasibility Conflict", and row "DLP DFS without Conflict Learning" with row "DLP DFS Infeasibility+Suboptimality Conflict". In addition, in order to compare the effect of infeasibility conflicts with that of sub-optimality conflicts, we compare row "DLP DFS Infeasibility Conflict" with row "DLP DFS Suboptimality Conflict".

In both the BFS and the DFS cases, the algorithm with conflict learning performs significantly better than the one without conflict learning. In addition, the difference increases with test problem size, from about 20% to about 95% in the BFS case and from about 20% to about 98% in the DFS case. Finally, for DFS using "Infeasibility Conflict" performs better than "Suboptimality Conflict", and the difference increases from 12% to 48% as the test problem enlarges.

Forward versus Backward Conflict-Directed Search

In order to show the reason for using forward conflict-directed search instead of conflict-directed backtrack search, we compare row "DLP BFS Infeasibility Conflict" with row "DLP BFS Conflict-directed Backtrack", and row "DLP DFS Infeasibility Conflict" with row "DLP DFS Conflict-directed Backtrack".

The backtrack algorithm, based on dependency-directed backtracking [28], 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. This difference increases as the test problem enlarges, from about 17% to about 90% in the BFS case and from about 12% to about 81% in the DFS case. In summary, the key finding is that forward conflict-directed search is better than conflict-directed backtrack in time efficiency.

Induced Unit Clause versus BIP Relaxation

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".

DLP performs significantly better than BIP, and the difference increases with test problem size, from about 10% to about 43%. It shows that our induced unit clause relaxation on DLPs works better than reformulating DLPs to use the BIP relaxation.

Best-first versus Depth-first

In order to address the tradeoffs of BFS and DFS, we compare row "DLP BFS without Conflict Learning" with row "DLP DFS without Conflict Learning", and compare row "DLP BFS Infeasibility Conflict" with row "DLP DFS Infeasibility Conflict" and row "DLP DFS Infeasibility+Suboptimality Conflict".

In the "without Conflict Learning" case, BFS performs better than DFS, and the difference increases with test problem size, from 14% to 43%. In the "Infeasibility Conflict" case, BFS also performs better than DFS, and the difference increases with test problem size, from 14% to 65%. Finally, BFS Infeasibility Conflict performs similar to DFS Infeasibility+Suboptimality Conflict; however, for large test problems, DFS performs better than BFS by up to 21%.

5.1.3 Maximum Queue Size

Maximum queue size of the search tree of each algorithm is recorded in Table 5.3. 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.

In summary, our key results corresponding to the three key features of GCD-BB are the following. First, in both the BFS and the DFS cases, the algorithm with

Table 5.3: 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

conflict learning performs significantly better than the one without conflict learning, and the difference goes up to 98% for large test problems. Second, in both the BFS and the DFS cases, the forward algorithm performs significantly better than the backward algorithm, and the difference goes up to 90% for large test problems. Third, our relaxation method performs significantly better than the BIP relaxation, and the difference goes up to 43% for large test problems.

5.2 Discussion

This thesis 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.

With respect to future work, empirically we would like to run GCD-BB on a range of well-known benchmark problems, and compare its runtime against that of BIP-BB. In addition, it would be interesting to study empirically the reason why sub-

optimality conflicts do not speed up search as much as infeasibility conflicts. Finally, there are several algorithmic improvements we want to explore in the future, such as applying GCD-BB to a more general form of HDLOPs than DLPs, and extending conflict learning to non-linear programs.

Bibliography

[1] E. Balas. Disjunctive programming. *Annals of Discrete Math.*, 5, 1979.

[2] R. Bayardo and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence*, 1997.

[3] D. Bertsimas and J.N. Tsitsiklis. *Introduction to Linear Optimization.* Athena Scientific, 1997.

[4] J. Bitner and E. Reingold. Backtrack programming techniques. *Communications of the Association for Computing Machinery*, 18(11), 1975.

[5] A. Borning, K. Marriott, P. Stuckey, and Y. Xiao. Solving linear arithmetic constraints for user interface applications. In *ACM Symposium on User Interface Software and Technology*, 1997.

[6] G. Borradaile and P. Van Hentenryck. Safe and tight linear estimators for global optimization. In *Constraint Programming Workshop on Interval Analysis and Constraint Propagation for Applications*, 2003.

[7] J. de Kleer and B. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32, 1987.

[8] J. de Kleer and B. Williams. Diagnosis with behavioral modes. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1989.

[9] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *ACM*, 32, 1985.

[10] D. Detlefs, G. Nelson, and J. Saxe. SIMPLIFY: A theorem prover for program checking. *HP Technical Report, H.P. Labs*, 2003.

[11] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12, 1979.

[12] J. Gaschnig. Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisfying assignment problems. In *Proceedings of the 2nd Canadian Conference on AI*, 1978.

[13] M. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[14] A. Hofmann and B. Williams. Safe execution of temporally flexible plans for bipedal walking devices. In *Plan Execution Workshop of the International Conference on Automated Planning and Scheduling*, 2005.

[15] J.N. Hooker. Logic, optimization and constraint programming. *INFORMS J. on Computing*, 14, 2002.

[16] J.N. Hooker and M.A. Osorio. Mixed logical/linear programming. *Discrete Applied Math.*, 96-97, 1999.

[17] H. Kautz and J.P. Walser. State space planning by integer optimization. In *Proceedings of the National Conference on Artificial Intelligence*, 1999.

[18] R. Krishnan. Solving hybrid decision-control problems through conflict-directed branch and bound. Master's thesis, MIT, 2004.

[19] A. Land and A. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28, 1960.

[20] T. Léauté and B. Williams. Coordinating agile systems through the model-based execution of temporal plans. In *Proceedings of the National Conference on Artificial Intelligence*, 2005.

[21] H. Li. On the "minimum" extreme ray of unbounded LPs. *Working paper*, 2005.

[22] A. Neumaier. Complete search in continuous global optimization and constraint satisfaction. *Acta Numerica*, 2004.

[23] P. Papalambros. Monotonicity in goal and geometric programming. *Mechanical Design*, 104, 1982.

[24] P. Papalambros and D. Wilde. Global non-iterative design optimization using monotonicity analysis. *Mechanical Design*, 101(4), 1979.

[25] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3), 1993.

[26] R. Ragno. Solving optimal satisfiability problems through clause-directed A*. Master's thesis, MIT, 2002.

[27] T. Schouwenaars, B. de Moor, E. Feron, and J. How. Mixed integer programming for multi-vehicle path planning. In *European Control Conference*, 2001.

[28] R. Stallman and G.J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9, 1977.

[29] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica: A Modeling Language for Global Optimization*. The MIT Press, 1997.

[30] T. Vossen, M. Ball, A. Lotem, and D. Nau. On the use of integer programming models in AI planning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1999.

[31] B. Williams and J. Cagan. Activity analysis: The qualitative analysis of stationary points for optimal reasoning. In *Proceedings of the National Conference on Artificial Intelligence*, 1994.

[32] B. Williams, M. Ingham, S. Chung, and P. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. IEEE*: Special Issue on Modeling and Design of Embedded Software*, 9(1), 2003.

[33] B. Williams and P. Nayak. Immobile robots: Artificial intelligence in the new millenium. *AI Magazine*, 17(3), 1996.

[34] B. Williams and P. Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of the National Conference on Artificial Intelligence*, 1996.

[35] B. Williams and R. Ragno. Conflict-directed A* and its role in model-based embedded systems. *To appear in Journal of Discrete Applied Math*, 2005.

[36] S. Wolfman and D. Weld. The LPSAT engine & its application to resource planning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1999.