

# **Solving Hybrid Decision-Control Problems Through Conflict-Directed Branch & Bound**

by  
Raj Krishnan

Submitted to the Department of Electrical Engineering and Computer Science  
in Partial Fulfillment of the Requirements for the Degree of  
Master of Engineering in Electrical Engineering and Computer Science  
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2, 2004

The author hereby grants to M.I.T. permission to reproduce and  
distribute publicly paper and electronic copies of this thesis  
and to grant others the right to do so.

Author \_\_\_\_\_  
Department of Electrical Engineering and Computer Science  
February 2, 2004

Certified by \_\_\_\_\_  
Brian C. Williams  
Thesis Supervisor

Accepted by \_\_\_\_\_  
Arthur C. Smith

# 1 Introduction

The creation of autonomous vehicles has been an area of extreme interest for years. Applications like self-controlled cars, intelligent robots, and unmanned military units have captured human imagination, while recent trends in space exploration, military tactics, and civilian urban development, have increasingly shown a need for autonomous vehicles outside of novels.

As we explore farther into deep space and the lag in radio communication increases, vehicles must possess greater autonomy. On the military front, the recent trend towards small-scale warfare reduces the need for division-level strategic planning, which is difficult to perform without humans; the focus is more on distributed vehicles operating cooperatively for reconnaissance and combat, a narrower problem that artificial intelligence is already working very hard to solve. Finally, social demands have compelled us to build taller and taller structures in urban areas; in emergencies they make ground-based search and rescue difficult, and pose significant hazards to human teams. As urban density increases and buildings get taller, these difficulties will make automated search-and-rescue more important.

Frequently autonomous systems implement *logical decision making* in order to evaluate differing choices. Examples include chess algorithms that implement constraint propagation to rule out possible piece moves, or any activity-level planner that establishes orderings on tasks. In contrast, other autonomous systems address problems that can be best represented as *algebraic* constraints over a continuous domain. An example of this type of system is a factory allocation program that distributes orders between locations, in order to minimize production costs while meeting minimum output constraints.

However, there is a significant set of problems that require a hybrid coupling of logical decision techniques with mathematical optimization. Deep space explorers must choose between various task selections and orderings, while also addressing the need for fuel usage optimization. Autonomous military vehicles will need to make decisions about which targets to strike, while minimizing mission costs in terms of time, fuel, and equipment damage. These decisions must also deal with probabilistic estimates of

vehicle loss in order to handle uncertainty. Urban search and rescue units must construct and compare different complex trajectories around a dangerous area (i.e., a fire) on the approach to a trapped individual.

All of these applications can be solved by systems that are composed of multiple cooperative autonomous vehicles. These vehicles must integrate elements of decision making, such as task ordering, obstacle avoidance, and goal selection, with some form of control-level trajectory planning that supports agile maneuvering. This thesis presents an algorithm for solving optimization problems that combine descriptions of hybrid decision-making logic, based on propositional clauses, with trajectory planning, based on linear programming.

## 1.1 Problem Statement

This thesis addresses the problem of efficiently creating a set of fuel optimal trajectories for a set of vehicles such that they achieve the goal locations specified in a temporally flexible plan, given the vehicle dynamics and an obstacle map. This objective is comprised of subsumes three separate subproblems. The first problem is to develop a formal framework for general trajectory problems that are a hybrid between logical decision making and mathematical control (referred to as *hybrid decision-control problems* or HDCPs). The second subproblem is to develop a fast solution method for these hybrid problems based on one-shot conflict learning. Finally, to demonstrate and evaluate these two elements using the cooperative path planning problem domain, which is representative of other domains that involve hybrid logic and mathematical elements.

## 1.2 Technical Challenges

The problem statement outlined in Section 1.1 poses a number of technical challenges for the formulation, solution, and demonstration parts of the thesis. First, the problem encoding must accurately reflect all the elements of the abstract HDCP. Modeling the decision-making component requires some way to represent choice between actions (a *disjunctive* element), while the standard model for optimally solving the control components is to construct a linear program (LP) and to use an LP solution technique such as the Simplex algorithm. Second, the encoding must also have a way to

incorporate all of the elements of a cooperative path planning problems. These elements might include the vehicles' dynamics, the mission goals, uneven terrain that might have real-valued incline gradients, and obstacles that must be avoided.

Third, the solution method must be efficient in space and time. Disjunctive elements in a control problem are far more costly to solve than a standard LP. These are traditionally solved using binary integer programs (BIPs), described in Chapter 2, where a decision between  $n$  inequalities is split into  $n$  constraints and  $n$  binary integer variables are used to impose the requirement that at least one of the constraints must be true. Traditional techniques for solving BIPs, such as the Simplex algorithm augmented with Branch and Bound (BB) techniques, are not sufficiently efficient for the CPP domain; therefore, our encoding must be coupled with a solution method that shows a relative improvement in speed. This also suggests the fourth and final challenge of this thesis: comparing the new encoding for HCDPs solved using the method introduced by this thesis, with a traditional BIP encoding solved using a Simplex-BB method. This comparison requires definitive metrics of comparison, and a characterization of the types of problems that can be solved efficiently.

### 1.3 Technical Approach Overview

To resolve the challenges outlined in Section 1.2, we introduce the Conflict-Directed Branch and Bound algorithm for Clausal LPs (CDCL-B&B). CDCL-B&B is based on three major insights. First, *Clausal Linear Programs* (clausal LPs, or CLPs) are introduced to develop a general formulation for combining choice and LP elements. The CLP framework also addresses the issue of incorporating all of the key parts of a CPP problem. Second, the efficiency problem is addressed by creating an algorithm that integrates Conflict-Directed A\* and Branch and Bound, modified for use on CLPs. This relies on one key idea: guiding search using minimal sets of constraints that result in an infeasibility (*conflicts*). Finally, this algorithm is demonstrated on coordinating path planning for multiple vehicles, a problem domain that has elements of both logical decision making and mathematical programming.

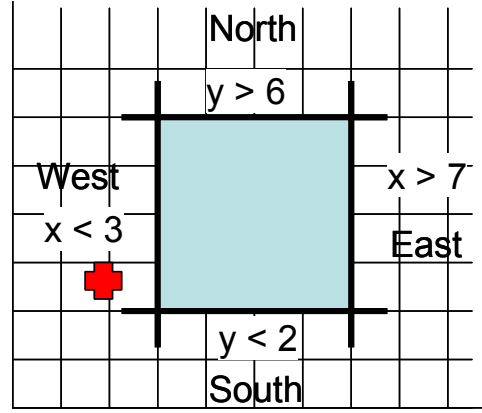
### 1.3.1 Clausal LPs

In general, solving an HDCP involves decision-making over a set of choices. The logical decisions that are made then result in a set of constraints that constitute a linear program. Encoding an HDCP requires some way of representing all the disjunctions of inequalities. Clausal LPs are the most natural solution to this need: each *clause* represents a decision to be made and each *disjunct* in the clause represents a choice made for that decision. Each disjunct is an inequality that must be satisfied.

As an example, consider the task of avoiding a square obstacle in a two-dimensional cooperative path planning problem. To avoid the obstacle, a vehicle must be one of  $\{North, South, East, West\}$  of the obstacle at every time instant. Figure 1(A) shows the boundaries of the obstacle for a particular time instant, and the constraint associated with each obstacle. Figure 1(B) shows the actual clausal LP representation of this obstacle: comprised of a disjunction of inequalities. Suppose some decision-making protocol is applied and resolves this decision

by choosing for the vehicle to be *West* of the obstacle, as indicated by the red cross. The inequality corresponding to the *West* disjunct would then be incorporated in the total LP problem. Figure 1(C) shows this inequality, “ $x < 3$ ”, included in the LP. In the general case, clausal LPs can be used to represent any arbitrary non-convex n-dimensional polytope (see Section 5.2).

Clausal LPs can represent simple linear inequalities in an HDCP. Constraints that do not involve choice can be included as unit clauses. Constraints with choice can be modeled as clauses.



(A)

$$x < 3 \vee x > 7 \vee y < 2 \vee y > 6$$

(B)

$$\begin{aligned} \min c^T x \\ \text{s.t. } Ax = b \\ x < 3 \end{aligned}$$

(C)

**Figure 1:** Modeling an obstacle in clausal LP form, and resolution of the clause.

### 1.3.2 Conflict-Directed Branch & Bound

Once Clausal LPs are defined as a way of encoding HDCPs, the next problem to address is solving the CLP optimally and efficiently. This requires finding the best feasible solution out of all possible selections of disjuncts across all clauses. The search space defined from the selection of a disjunct from each clause can be viewed as an enumerated tree, each node of which specifies a partial resolution of clauses (in other words, a selection of a single disjunct from each clause in a subset of the total set of clauses). For each such node, the selected inequalities combined with the non-disjunctive constraints result in a *relaxed* (partial) LP problem that must be solved. Each leaf node in this tree falls into one of two categories: 1) nodes that contain a complete resolution of clauses, and at which the solved LP is a feasible solution to the HDCP; and 2) nodes in which the selected inequalities, when combined with the non-disjunctive constraints, result in a relaxed LP that is infeasible.

When searching through the search tree, significant time can be wasted in expanding subtrees that all terminate in infeasible nodes even when using best-first search. This is particularly true because solving the LP problem at a node is the most costly submethod in the search. To focus away from infeasible subspaces of solutions, we use the technique of *Conflict-Directed A\**, a one-shot learning method that generalizes the reason for an inconsistent solution into an identified infeasible subspace. This is implemented through a process of identifying and resolving *conflicts*, sets of constraints that are together infeasible. For each conflict, a constituent kernel is generated. A *constituent kernel* is a minimal description of all states that resolve a particular conflict; that is, those states in which the conflict is guaranteed to not occur. Each new node in search tree is built from a *kernel*, which selects one constituent kernel for each conflict.

### 1.3.3 Cooperative Path Planning

The domain selected for demonstration of the CDBB algorithm is cooperative path planning for multiple autonomous vehicles. Problems in this domain achieve a number of required goals. First, they contain clear logical decision elements such as obstacle avoidance, described in Section 1.3.1. These are naturally modeled in an HDCP.

Second, they contain clear mathematical control elements, including continuous constraints and a goal of optimizing some function like fuel usage; these can also be modeled in an HDCP. Third, CPP problems can also be formulated into a BIP and solved by the Simplex-BB method. This enables comparison with the method traditionally used. Fourth, current methods for solving these problems are not sufficiently fast for applied use, such as for on-line vehicles control.

## **1.4 Thesis Claims**

1. Hybrid Decision-Control Problems can be effectively modeled as Clausal LPs
2. Clausal LPs can be solved using a form of one-shot learning, called Conflict-Directed Branch and Bound
3. The application of CDBB to solving HDCPs can be demonstrated on cooperative path planning problems, on which a factor of X improvement is demonstrated over the benchmark (MILP solved using Simplex-BB)

## **1.5 Chapter Summary**

Chapter 2 introduces the concept of Clausal LPs as a representation for hybrid decision-control problems, and examines the application of the clausal framework to cooperative path planning problems. Chapter 3 reviews Branch and Bound for binary integer programming problems and the Constraint-Based A\* algorithm. It then uses these as the basis for a Branch and Bound algorithm for Clausal LP problems (CL-B&B). Chapter 4 introduces conflict-directed search for clausal LPs. It reviews Conflict-directed A\*, defines conflicts and conflict resolution in the Clausal LP framework, and concludes with the description of a Conflict-directed Branch and Bound algorithm for Clausal LPs (CDCL-B&B). Chapter 6 assesses the efficiency improvement of CDCL-B&B over BIP-B&B in the CPP domain, and compares the efficiency of different methods of conflict extraction. Chapter 7 looks at future development with the CDCL algorithm, including integration into the Kirk vehicle control system.

## 2 Encoding Hybrid Decision-Control Problems as Clausal LPs

This chapter discusses the formulation of Hybrid Decision/Control Problems (HDCPs) in a way that enables fast discovery of an optimal solution. It introduces a representational formalism, called Clausal LPs, which effectively captures both the mathematical dynamics and the control decisions that are present in these types of problems. Specifically, Clausal LPs allow for the logical decision elements of the problem to be specified as disjunctions of inequalities instead of being translated into mixed integer arithmetic constraints for use in a linear program.

As background to this formulation, this chapter begins with a description of previous methods for modeling hybrid decision-control problems. The most common model for representing mathematical control elements for optimal search is to construct a linear program (LP) and to then use LP solution techniques like the Simplex algorithm [*Simplex algorithm reference*]. Proper representation of HDCPs also requires a way to encode decision-making over a set of choices. Currently in the mathematical programming field, the standard technique for integrating these two elements is mixed integer-linear programs (MIPs).

The focus of Section 2.1 is to provide a review of a particular class of MIPs where the integer values are constrained to be either zero or one. These problems are known as binary integer programs (BIPs). Section 2.2 introduces Clausal LPs and compares them to BIPs in terms of expressiveness and in terms of enabling search efficiency. It shows that any Clausal LP can be modeled as a BIP, and that any BIP where at most a single constraint is added by every decision can be modeled as a Clausal LP. It also concludes that the Clausal LP framework is more efficient than BIPs in terms of the number of nodes that must be searched.

### 2.1 Binary Integer Programs

Any representation of a hybrid decision-control problem must encode the HDCP's control elements, such as vehicle dynamics to completely solve the problem. These control elements have traditionally been represented as a set of linear constraints on a set



of real-valued variables, making them solvable using an LP solver. These linear constraints, combined with the objective function of a hybrid problem, such as a fuel-use minimization function, make up a linear program. The general format of a linear program is

$$\begin{array}{ll} \text{minimize} & \mathbf{c}^T \mathbf{x}, \\ \text{subject to} & \mathbf{A} \mathbf{x} \geq \mathbf{b} \quad \{2.1-1\} \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}, \quad \{2.1-2\} \end{array}$$

where  $\mathbf{A}$  is an  $m \times n$  matrix,  $\mathbf{c}$ ,  $\mathbf{x}$ ,  $\mathbf{l}$ , and  $\mathbf{u}$  are  $n$ -size vectors, and  $\mathbf{b}$  is an  $m$ -size vector [REFERENCE: *Beasley text*]. Here  $\mathbf{c}^T \mathbf{x}$  is the objective function and {2.1-1} and {2.1-2} together comprise the set of LP constraints. Generally, LPs can require either the minimization or the maximization of the objective function.

A MIP may augment the standard LP representation by adding the restriction that some subset of the variables  $x_j$  must be integer. A BIP further restricts all integer variables to being binary; that is, any integer variable can only have a value of zero or one. This restriction does not result in a loss of expressiveness; any integer variable with  $n$  possible values can be replaced with  $\log_2(n)$  binary variables that encode the same number of possible choices, without even a significant change in problem complexity.

There are several reasons why BIPs are a good selection as a benchmark framework against which to compare Clausal LPs. First, binary values are frequently used to represent the logical values of “true” and “false”. A decision between multiple choices can be viewed as the assignment of “true” to a single choice and “false” to the others, so BIPs are traditionally used to model problems involving logical decisions. Secondly, the binary restriction makes BIPs easier to analyze than MIPs because all integer variables have identical domain. For instance, in a BIP the total number of possible assignments is exactly  $2^{N_b}$ , where  $N_b$  is the number of binary variables. In contrast, in a MIP the total number is the product of the domain size of all integer variables. Therefore complexity analysis, like that done in Section 2.2.3 below, is much easier on BIP problems.

BIPs encode the objective function and control elements of an HDCP in the general LP format shown in {2.1-1} and {2.1-2}. The inclusion of binary variables enables a BIP to model the decision elements of the HDCP as well, which cannot be encoded in a simple LP. If the selection of a choice  $j$  in a decision  $i$  in the HDCP results

in the inclusion of a constraint  $A_{i,j}x \geq c_{i,j}$  as one of the problem's control elements, then choice  $j$  is encoded in a BIP as

$$A_{i,j}x - c_{i,j} \geq R(1 - b_{i,j}) \quad \{2.1-3\}$$

where  $R$  is a negative number with an extremely large magnitude relative to the other constants in the constraint, and  $b_{i,j}$  is a binary variable *[cite John How's formulation, and standard encodings]*.

The purpose of  $b_{i,j}$  in this constraint is to govern whether choice  $j$  is selected or not. If  $j$  is not selected, then this constraint should have no impact on the LP solution. If  $j$  is selected, then this constraint should be included in its original form in the LP. To do this, let the assignment of  $b_{i,j} = 0$  mean that  $j$  is not selected, and the assignment of  $b_{i,j} = 1$  mean that  $j$  is selected. If  $b_{i,j} = 0$  then  $\{2.1-3\}$  becomes  $A_{i,j}x - c_{i,j} \geq R$ . Since  $R$  is a very negative number, this constraint is trivially satisfied regardless of the values assigned to  $x$ , and so the constraint has no impact on the LP solution. If  $b_{i,j} = 1$  then  $\{2.1-3\}$  becomes  $A_{i,j}x - c_{i,j} \geq 0$ , which is equivalent to the original constraint  $A_{i,j}x \geq c_{i,j}$ .

The last element to complete the representation of a decision problem as a BIP is to add one constraint for every decision  $i$  that ensures that at least *one* choice in  $i$  is selected. Therefore for a decision  $i$

$$\sum_j b_{i,j} \geq 1 \quad \{2.1-4\}$$

is added to the LP. In order for the sum of the  $b_{i,j}$ 's across all  $j$ 's of decision  $i$  to be more than 1, at least one of the  $b_{i,j}$ 's must equal 1. This is equivalent to saying that at least one of the choices  $j$  must be selected from decision  $i$ .

As an example of the BIP representation, consider the following resource allocation problem, which incorporates the same elements as a hybrid decision-control problem. A manufacturing company is trying to set production levels for the coming year, and also decide which one of its three plants will remain open. The company produces two types of products,  $x$  and  $y$ . Production costs are \$30 per unit of  $x$  and \$40 per unit of  $y$ . It is known that demand for  $x$  will be at least 400 units, demand for  $y$  will be at least 180 units, and total demand for either product will be at least 600. Company policies require that all demand must be met in the coming year. The three factories each have different constraints: factory one can produce unlimited  $y$  but a maximum of 410  $x$ , factory two can produce unlimited  $x$  but a maximum of 190  $y$ , and factory three can

produce a maximum total of 590 products. One of these three factories can be chosen as a production facility. If the goal is to minimize total cost, what should the company do?

The first step in modeling the problem as a BIP is to identify the objective function. The goal is cost minimization, so the function would be

$$\text{minimize} \quad 30x + 40y \quad \{2.1-5\}$$

Next, consider the constraints that do not involve any decisions, the demand constraints. These are modeled as standard LP constraints:

$$\text{subject to} \quad x \geq 400 \quad \{2.1-6\}$$

$$y \geq 180 \quad \{2.1-7\}$$

$$x + y \geq 600 \quad \{2.1-8\}$$

Each of the remaining constraints, the production limits for each factory, is only relevant to the LP if the corresponding factory remains open. The constraints for each factory, in order, would be  $x \leq 410$ ,  $y \leq 190$ , and  $x + y \leq 590$ . To establish a consistent form for the constraints, these can be converted to greater-than inequalities:  $-x \geq -410$ ,  $-y \geq -190$ , and  $-x - y \geq -590$ .

The factory production constraints are composed into a single decision with three different choices: keeping open factory one, two, or three. Therefore three binary variables  $b_{1,1}$ ,  $b_{1,2}$ , and  $b_{1,3}$  are created and the constraints in greater-than form are modeled as in {2.1-3}:

$$-x + 410 \geq R(1 - b_{1,1}) \quad \{2.1-9\}$$

$$-y + 190 \geq R(1 - b_{1,2}) \quad \{2.1-10\}$$

$$-x - y + 590 \geq R(1 - b_{1,3}) \quad \{2.1-11\}$$

where  $R$  is some very negative number, like -10,000,000. If factory one is kept open, then  $b_{1,1} = 1$  and {2.1-9} becomes  $-x + 410 \geq 0$ , which is equivalent to including the original factory one constraint in the LP. Otherwise,  $b_{1,1} = 0$ , which makes {2.1-9} become  $-x + 410 \geq -10,000,000$ . This is trivially satisfied for any reasonable value of  $x$  and  $y$ . {2.1-10} and {2.1-11} operate in the same way.

Finally, a constraint must be added that ensures that at least one of the factories is chosen to remain open. Modeled after {2.1-4}, the constraint

$$b_{1,1} + b_{1,2} + b_{1,3} \geq 1 \quad \{2.1-12\}$$

completes the BIP formulation.

## 2.2 Clausal LPs

This section introduces Clausal LPs as an alternative way of modeling hybrid decision-control problems. The primary difference between the CLP representation and the BIP representation is the way in which logical decisions are handled. While BIPs use binary variables to enumerate logical decisions, CLPs instead represent a decision as a disjunctive clause made up of terms that are LP constraints. For example, a clause in a CLP might be  $x \geq 40 \vee x \leq 20$ .

Each subsection details a key advantage of CLPs over the BIP formulation of decision problems. Section 2.2.1 describes the creation of a CLP from a hybrid problem, and shows that CLPs are a *direct* encoding of inclusive choice. Section 2.2.2 performs a comparison of the formulation in terms of expressiveness and identifies the subclass of BIPs that can be modeled as CLPs. Finally, Section 2.2.3 performs a comparison in terms of enabling search efficiency, and concludes that Clausal LPs enable faster search.

### 2.2.1 Representing Choice in a Clausal LP

The Clausal LP representation uses disjunctive clauses as a way of encoding the decision component of a hybrid decision-control problem. In the most general case, any decision problem that involves selection of a certain set of constraints versus another set can be modeled as a Clausal LP. More specifically, the decision problems that can be encoded consist of  $N_d$  decisions, with the  $i^{\text{th}}$  decision consists of  $N_i$  choices. This thesis only considers decision problems where selecting choice  $j$  of decision  $i$  means the inclusion of a single linear inequality in the resulting control problem. In principle, however, the definition of Clausal LPs can be expanded to include the encoding of decision problems where an arbitrary number of additional linear inequalities are added when each  $i_j$  is chosen. Chapter 7 briefly examines what future modifications could be made to incorporate arbitrary-size constraint sets.

The Clausal LP represents each decision  $i$  in the HDCP as a disjunctive *clause*

$$c_{i,1} \vee c_{i,2} \vee \dots c_{i,N_i} \tag{2.2-1}$$

where  $c_{i,j}$  is the inequality constraint added when choice  $i_j$  is made. Each term in a clause is known as a *disjunct*; the clause in {2.2-1}, for example, has  $N_i$  disjuncts. For the

Clausal LP to be solved, at least one disjunct from each clause must be satisfied by the LP solution; this is known as *resolving* the clause.

In general an HDCP will also contain additional inequalities that do not involve any choice; these constraints are part of the control problem regardless of what choices are made. In a Clausal LP these constraints are modeled as unit clauses: disjunctive clauses with a single term. In the Clausal LP framework there is no need to distinguish between these two types of constraints. Similar to LPs, clausal LPs also contain an objective function that establishes a cost or value to any solution point. This objective function holds over all choices of constraints.

For example, consider again the simple manufacturing cost-minimization problem used to illustrate BIP formulation in Section 2.1. In the Clausal LP framework, the same problem would be encoded as

$$\begin{array}{llll}
 \text{minimize} & 30x + 40y, & & \{2.2-2\} \\
 \text{subject to} & x \geq 400 & & \{2.2-3\} \\
 & y \geq 180 & & \{2.2-4\} \\
 & x + y \geq 600 & & \{2.2-5\} \\
 & -x \geq -410 \vee -y \geq -190 \vee -x - y \geq -590 & & \{2.2-6\}
 \end{array}$$

In this encoding {2.2-2} is the objective function, {2.2-3}, {2.2-4} and {2.2-5} are unit clauses, and {2.2-6} is a clause with three disjuncts.

An important characteristic of Clausal LPs is that they encode the logical choice in a direct form: clauses are an actual representation of the disjunctive decision in the HDCP. This is in contrast to the BIP encoding, where logical choice is encoded as a series of conjunctive constraints like {2.1-3} and a single summation of the form of {2.1-4}. Hence a CLP offers a more natural representation of decision-making problems than a BIP does.

The direct representation of disjunctive choice in the CLP formulation is similar to the LCNF formalism developed in [Weld 97]. The LCNF formalism encodes disjunctions that represent decisions, and adds constraints to a linear problem as specific choices are made. In this regard, LCNF is similar to the Clausal LP framework. The key difference, however, is that LCNF does not encode any objective function, and is used instead only to find a *feasible* solution. CLPs extend the LCNF formalism by ranking solutions based on the value of an objective function, evaluated at that solution.

Therefore, CLPs enable search for an *optimal* solution over a set of possible candidates. Additionally, the semantics of LCNF are slightly different: clauses do not directly contain constraints, but instead store a disjunction of propositional variables. Assigning “true” to any of these variables entails a particular constraint, as encoded by a series of implications.

## 2.2.2 Expressiveness of Clausal LPs and BIPs

Sections 2.1 and 2.2 showed how certain limited-scope HDCPs can be represented, respectively, as binary integer programming problems and Clausal LPs. Therefore there exists a domain of problems for which the two representations are mutually transformable. In particular, problems where every disjunct of every clause consists of a single constraint, not a conjunction of constraints, can all be modeled as either BIPs or CLPs. Hence the two representations are equivalent for this class of HDCPs. 2.2.2.1 gives the general transformation of a CLP to a BIP, and 2.2.2.2 gives the reverse transformation from a BIP to a CLP. Chapter 7 discusses a potential modification to the Clausal LP framework that could enable them to be as expressive as BIPs in general as well.

### 2.2.2.1. Clausal LP to BIP Transformation

First consider the mapping from a CLP to an equivalent BIP. Suppose **CLP** is an arbitrary Clausal LP with  $N_d$  clauses, and suppose the  $i^{\text{th}}$  clause of **CLP** consists of  $N_i$  disjuncts, with each  $N_i \geq 2$ . Suppose further that there is an additional  $N_d'$  clauses which each consist of a single disjunct. In this case **CLP** can be transformed into a BIP that consists of

$$N_b = \sum_{i=1 \text{ to } N_d} N_i \quad \{2.2-7\}$$

additional binary variables and  $N_b + N_d' + N_d$  constraints.

To perform this transformation, create a new binary variable  $b_{i,j}$  for each constraint  $A_{i,j}x \geq c_{i,j}$ : the  $j^{\text{th}}$  disjunct of the  $i^{\text{th}}$  clause of **CLP**. Replace each clause  $i$  with the following  $N_i + 1$  constraints:

$$A_{i,1}x - c_{i,1} \geq R(1 - b_{i,1}) \quad \{2.2-8\}$$

$$A_{i,2}x - c_{i,2} \geq R(1 - b_{i,2}) \quad \{2.2-9\}$$

. . .

$$A_{i,N_i}x - c_{i,N_i} \geq R(1 - b_{i,N_i}) \quad \{2.2-10\}$$

$$\sum_{j=1 \text{ to } N_i} b_{i,j} \geq 1 \quad \{2.2-11\}$$

where  $R$  is a negative number with an extremely large magnitude relative to the other constants in the constraint. In this encoding, each of the constraints {2.2-8} through {2.2-10} represents a single choice in the clause. Selecting the  $j^{\text{th}}$  disjunct of the  $i^{\text{th}}$  clause in the Clausal LP framework is equivalent to assigning  $b_{i,j} = 1$  in the BIP framework; all other binary variables for disjuncts in clause  $i$  can be assigned to be zero. If the binary variable corresponding to a constraint is set to zero, the constraint becomes  $A_{i,j}x - c_{i,j} \geq R$  in the BIP. Because  $R$  is a very negative number, this constraint is trivially satisfied regardless of the values of  $x$  and so has no impact on the solution. This is equivalent to not selecting the corresponding disjunct in the Clausal LP. Constraint {2.2-11} ensures that at least one of the binary variables is assigned to be 1; this is equivalent to requiring that at least one disjunct in a clause be selected.  $N_d$  clauses contributing  $N_i + 1$  constraints each add a total of  $(\sum_{i=1 \text{ to } N_d} N_i) + N_d$  constraints to the BIP problem. From {2.2-7}, this is equal to  $N_b + N_d$  constraints.

The remaining  $N_d'$  constraints in the BIP derive from the  $N_d'$  unit clauses in the Clausal LP. For each unit clause, add the clause's single disjunct as a constraint in the BIP. Since each disjunct of a unit clause must be met in the Clausal LP in order to resolve the clause, these disjuncts are also required constraints in the BIP. The final step in the Clausal LP-to-BIP transformation is to use the objective function of the Clausal LP as the BIP objective function as well.

As an example of this transformation, again consider the simple manufacturing problem used previously. The Clausal LP formulation of this problem is given in {2.2-2} to {2.2-6} in Section 2.2. In this problem  $N_d'$  is 3,  $N_d$  is 1, and  $N_i$  for  $i=1$  is 3. Therefore, from {2.2-7}, the number of new binary variables to be created is

$$N_b = \sum_{i=1 \text{ to } N_d} N_i = \sum_{i=1 \text{ to } 1} N_i = N_1 = 3 \quad \{2.2-12\}$$

These binary variables will be termed  $b_{1,1}$ ,  $b_{1,2}$ , and  $b_{1,3}$ , since there is a single clause with three disjuncts. Following the format of {2.2-8} to {2.2-11}, clause {2.2-6} becomes

$$-x + 410 \geq R(1 - b_{1,1}) \quad \{2.2-13\}$$

$$-y + 190 \geq R(1 - b_{1,2}) \quad \{2.2-14\}$$

$$-x - y + 590 \geq R(1 - b_{1,3}) \quad \{2.2-15\}$$

$$b_{1,1} + b_{1,2} + b_{1,3} \geq 1 \quad \{2.2-16\}$$

The constraints {2.2-3} to {2.2-5} are each disjuncts in separate unit clauses, and so are added into the BIP without modification. Finally, the Clausal LP objective function in {2.2-2} is also used as the objective function in the BIP. Note that the BIP that consists of {2.2-2} to {2.2-5} and {2.2-13} to {2.2-16} is identical to the initial BIP formulation of the manufacturing problem in Section 2.1. There are a total of seven constraints, and  $N_b + N_d' + N_d$  is  $3 + 3 + 1 = 7$ .

#### 2.2.2.2. BIP to Clausal LP Transformation

To complete the proof of equivalence, now consider the mapping from a BIP to a CLP. Performing the reverse transformation is complicated by the fact that in general a BIP is not guaranteed to have entire constraints multiplied by a single binary variable. That is, not all BIP constraints will necessarily be of the form  $A_{i,j}x \cdot b_{i,j} \geq c_{i,j} \cdot b_{i,j}$  (which is equivalent to the disjunct of a Clausal LP clause) or of the form  $A_{i,j}x \geq c_{i,j}$  (which is equivalent to the disjunct of a Clausal LP unit clause). Even in the restricted case where the BIP only contains linear constraints, the general constraint form might be

$$\sum_p a_p b_p + \sum_q a_q x_q \geq c \quad \{2.2-19\}$$

For example,  $3 \cdot b_1 + 3 \cdot x \geq 5$  is one such constraint.

In the most general case, a BIP can be converted into a Clausal LP by enumerating all possible assignments to the binary variables and explicitly making each of those assignments a disjunct in one large clause. This would result in a total of  $2^{N_b}$  disjuncts where  $N_b$  is the number of binary variables. Constraints that do not contain binary variables would become disjuncts in unit clauses. For example, {2.2-18} would present four possible assignment possibilities for the two variables  $b_1$  and  $b_2$ . The following table shows the resultant constraint for each of the combination:

$(b_1, b_2)$	Resulting Constraint
(0, 0)	$z \geq 5$
(0, 1)	$y + z \geq 5$
(1, 0)	$x + z \geq 8$
(1, 1)	$x + y + z \geq 8$

Thus the clause that would result from this constraint would be:



$$z \geq 5 \vee y + z \geq 5 \vee x + z \geq 8 \vee x + y + z \geq 8 \quad \{2.2-20\}$$

The fact that this transformation can take place proves equivalence, but does necessarily provide a compact encoding for CLPs. A total of  $2^{N_b}$  states exist by using the transformation given above. Chapter 7 examines possible methods for reducing the total number of states.

### 2.2.3 Increased Search Efficiency through Clausal LPs

Clausal LPs offer a direct logical encoding of choice between alternative constraints. For HDCPs in which the decision element is primarily this type of choice, CLPs offer two major search efficiency improvements over BIPs. First, the size of the B&B search tree for CLPs is smaller than the tree that is searched when using a BIP encoding. Second, frequently the structure of a CLP enables it to be naturally relaxed to a propositional logic problem, which abstracts away the LP details of the constraints. This relaxation enables optimal satisfiability algorithms to be employed in order to perform pruning and guide search.

The search efficiency of a Clausal LP representation and its BIP transformation can be compared by analyzing the worst-case number of search tree nodes in each representation. In the Clausal LP representation, each non-leaf node is branched into  $N_i$  children because of the resolution of clause  $i$  by selecting one its  $N_i$  choices. Given a set number of choices, the worst case number of nodes occurs when the product of all the  $N_i$ 's are largest. This occurs when all clauses have an equal number of disjuncts; let this value be  $N_0$ . In this case, each node branches into  $N_0$  children. There is 1 node (the root) at the 0<sup>th</sup> level,  $N_0$  nodes at the 1<sup>st</sup> level,  $N_0^2$  nodes at the 2<sup>nd</sup> level, and  $N_0^i$  nodes at the  $i^{\text{th}}$  level. Hence the worst-case number of nodes in the Clausal LP formulation is

$$\sum_{i=0 \text{ to } N_d} N_0^i = (N_0^{N_d+1} - 1) / (N_0 - 1) \quad \{2.2-21\}$$

if there are  $N_d$  total decisions to be made.

The search tree of a BIP encoding looks identical to the Clausal LP case, but with every node expanding to exactly two children. Hence the equation for determining the number of nodes looks like {2.2-21}, but  $N_0 = 2$ . There are also  $N_b$  binary variables to resolve, instead of  $N_d$  clauses to resolve, and so  $N_d$  in {2.2-21} is replaced by  $N_b$ . These values are the same regardless of the distribution of choices among the decisions, so there

is no need to identify a worst-case situation. Hence the number of nodes to explore in a BIP is

$$\sum_{i=0 \text{ to } N_b} 2^i = 2^{N_b+1} - 1 \quad \{2.2-22\}$$

Note that most search algorithms will prune subtrees of the search tree, and so the average case complexity in practice is a function of algorithm efficiency as much as encoding efficiency.

$N_0$ :		2		4		8		12	
		BIP	CLP	BIP	CLP	BIP	CLP	BIP	CLP
$N_d$ :	1	7	3	31	5	511	9	8191	13
	2	31	7	511	21	131071	73	3E+07	157
	4	511	31	131071	341	9E+09	4681	6E+14	22621
	8	131071	511	9E+09	87381	4E+19	2E+07	2E+29	5E+08
	12	3E+07	8191	6E+14	2E+07	2E+29	8E+10	4E+43	1E+13

**Figure 2:** Comparison of the total number of nodes in a binary integer programming formulation versus the worst-case Clausal LP formulation. These values are based on the number of decisions ( $N_d$ ) in the source HDCP, and the number of choices per decision (fixed at  $N_0$ ).

There exist two alternate methods of determining the comparative size of {2.2-21} and {2.2-22}. Consider first the encoding given in 2.2.2.1 for converting CLPs to BIPs. In this transformation, the number of binary variables needed is defined by {2.2-7} as  $\sum_{i=1 \text{ to } N_d} N_i$ , which is equal to the number of disjuncts. Recall that the worst-case Clausal LP formulation occurs when the disjuncts are equally distributed, while the number of binary variables in the BIP is unaffected by the distribution. If each clause has  $N_0$  disjuncts, then  $N_b = N_0 \cdot N_d$ , and the number of BIP nodes is  $2^{N_0 \cdot N_d + 1} - 1$ . {2.2-21} gives the number of CLP nodes also as a function of  $N_0$  and  $N_d$ . Figure 2 compares the number of nodes given varying values of  $N_0$  and  $N_d$ , and shows that the CLP formulation consistently has fewer search nodes.

This determination of the relationship between  $N_0$ ,  $N_d$  and  $N_b$  is specific to the particular encoding given in 2.2.2.1. In the general case, consider the encoding of any HDCP with  $N_d$  decisions and the  $i^{\text{th}}$  decision having  $N_i$  choices. As a Clausal LP in the worst case (even distribution of disjuncts), the number of nodes is still governed by {2.2-21}. As a BIP, recall that the number of created binary variables equals the total number

of choices across all decisions. This number is  $N_b = \sum_{i=1 \text{ to } N_d} N_i = N_0 \cdot N_d$ . Therefore the total number of nodes is again given by {2.2-22}. Therefore the results in Figure 2 still hold.

The worst-case efficiency improvement of Clausal LPs over BIPs discussed so far is based only on the total size of the B&B search tree. It is also important to consider how the Clausal LP representation enables algorithms to explore the search tree in less than worst-case time. Search efficiency in exploring a tree is increased dramatically through *pruning*. A node  $n$  can be pruned when all leaf nodes that are below  $n$  are guaranteed to not provide an optimal solution to the problem. This reduces the total number of nodes to explore by the size of the subtree; if  $n$  is high in the tree, a significant part of the tree may be ignored.

In B&B-based techniques that solve BIP problems, pruning is performed by constructing relaxed LPs. These are problems where only a subset of all binary variables are assigned binary values; the rest are unconstrained and may be assigned real values. These relaxed problems are solved at intermediary nodes between the root of the tree and the leaves, and can be used to identify subtrees that can be pruned. Chapter 3 explores relaxed problems in more detail and shows how similar LP relaxations can be constructed for CLPs.

However, Clausal LPs have the added benefit that they can be represented as propositional sat problems. In this representation, every clause is viewed as a simple selection between multiple choices, much like a direct formalization of only the decision elements in an HDCP. A given disjunct in a clause is only a symbolic term, and the specific LP details of the constraint that the disjunct maps to is abstracted away. This is also equivalent to viewing clauses as variables needing assignments, with each variable's domain equal to the set of disjuncts for the clause. With this view, subtrees with relaxed clauses that are unsatisfiable on the propositional level can be identified and pruned. Chapter 4 outlines a method for identifying these prunable nodes and introduces the CDCL-B&B algorithm, which combines pruning from B&B methods with pruning from SAT.

### 3 Solving Clausal LPs through Branch and Bound

Branch & Bound is one of the most effective pruning methods for efficiently solving BIPs. The key idea in B&B is to compute an optimistic bound on the solution costs for nodes within a subtree of the search space, and to prune that subtree whenever the bound is proven to be worse than a feasible solution already found. This chapter introduces a method for solving Clausal LP representations of hybrid decision-control problems (HDCPs) that is analogous to the combined Simplex-B&B method for solving binary integer programs (BIPs). The key difference between the algorithms is in the way they create branches in the search tree. The Clausal LP B&B algorithm introduced here branches by selecting a disjunct from a clause rather than an integer assignment from the domain of possible assignments. As we showed in Chapter 2, the advantage of this approach is that the growth in the size of this search tree tends to be substantially less than that for the equivalent encoding of the CLP as a BIP. In addition CLP provides a building block for Conflict-directed Branch & Bound, introduced in Chapter 4.

Section 3.1 first reviews the traditional use of Simplex-BB techniques to solve BIP problems. This BIP-BB provides the basis for the modified Branch and Bound algorithm for Clausal LPs, introduced in Section 3.3. CL-BB then provides a building point for the complete Conflict-directed Branch and Bound algorithm, introduced in Chapter 4, which completes the unification of the search efficiency of B&B with conflict-direction and the powerful Clausal LP representation introduced in Chapter 2.

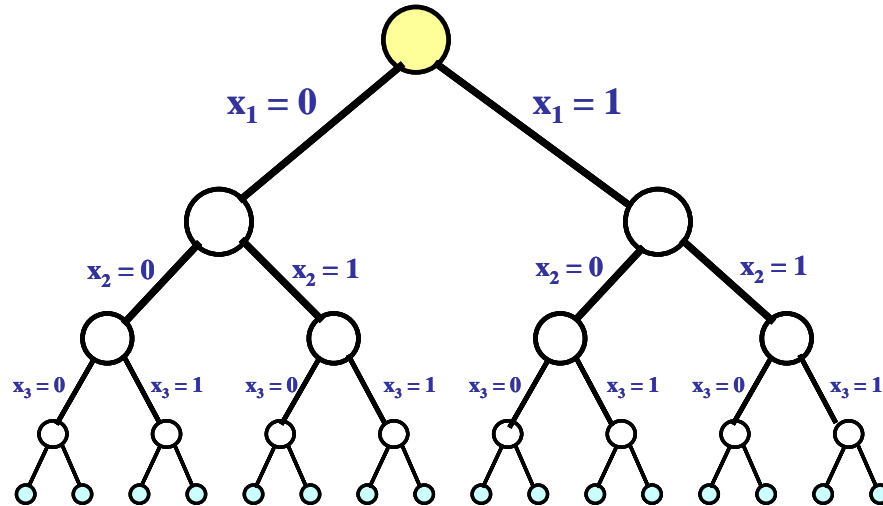
#### 3.1 Review of Branch and Bound for BIPs

The Branch and Bound technique has traditionally been applied to MIP problems, and many of the key concepts of the algorithm carry over naturally to solving Clausal LPs. The parallels between a BB solver for MIPs and a BB Solver for Clausal LPs are particularly strong when one considers the restriction of MIP problems to the binary case (BIPs). This section describes the B&B algorithm for BIPs as a foundation for the clausal LP algorithms of this thesis. The motivation for the algorithm and the basic search process are introduced in Section 3.1.1. Sections 3.1.2 to 3.1.4 review traditional methods for improving search efficiency by identifying subsets of the search tree that can

be ignored without a loss of solution optimality. Each of these methods relies on the discovery of a special type of node midway down the tree. These special types are, respectively, 1) nodes more costly than the current best solution (the *incumbent*), 2) infeasible nodes, and 3) new incumbent nodes. Section 3.1.5 outlines an additional modification that improves search efficiency by reducing the depth of the search tree.

### 3.1.1 Exhaustive Search Using Incumbents for a Global Optimal Solution

The solution to a BIP problem must assign values to both the real-valued variables and the binary variables of the problem. A straightforward method for deriving such a solution is to consider all possible assignments to the set of binary variables. Given a particular binary variable assignment, the best solution that is consistent with this assignment can be found by solving a single LP over the remaining unassigned, real-valued variables. The optimal solution to the BIP is the best solution over all possible individual assignments.



**Figure 3:** Search tree enumeration for Branch and Bound applied to Binary MIPs.

To keep track of the best solution at every step in an exhaustive search of all possible assignments, the algorithm uses the concept of an *incumbent*. An incumbent solution 1) assigns integer values to all binary variables; and 2) is more optimal (less costly) than any previously discovered feasible solution. The incumbent solution is continuously updated as better feasible solutions are found during the search. Therefore, for minimization problems, the cost of the current incumbent sets an upper bound on the

possible cost of the best solution. Any LP problem that results in a solution that is more costly than the incumbent will be discarded.

```

BIP_BB(root)
1      U =  $+\infty$ 
2      incumbent = null
3      q = empty queue
4      q.insertFirst(root)
5      while (q not empty) {
6          current = q.removeFirst( )
7          cost = solution to relaxed LP at current
8          if (current is infeasible)
9              go to line 5
10         if (cost  $\geq$  U)
11             go to line 5
12         non_integer_found = false
13         for each binary variable j, while (! non_integer_found) {
14             if (value of j is non-integer) {
15                 non_integer_found = true
16                 p1 = copy of current
17                 add constraint "j = 0" to p1
18                 q.insertFirst(p1)
19                 p2 = copy of current
20                 add constraint "j = 1" to p2
21                 q.insertFirst(p2)
22             }
23         }
24         if (! non_integer_found) {
25             U = cost
26             incumbent = current
27         }
28     }
29     return incumbent's variable assignment

```

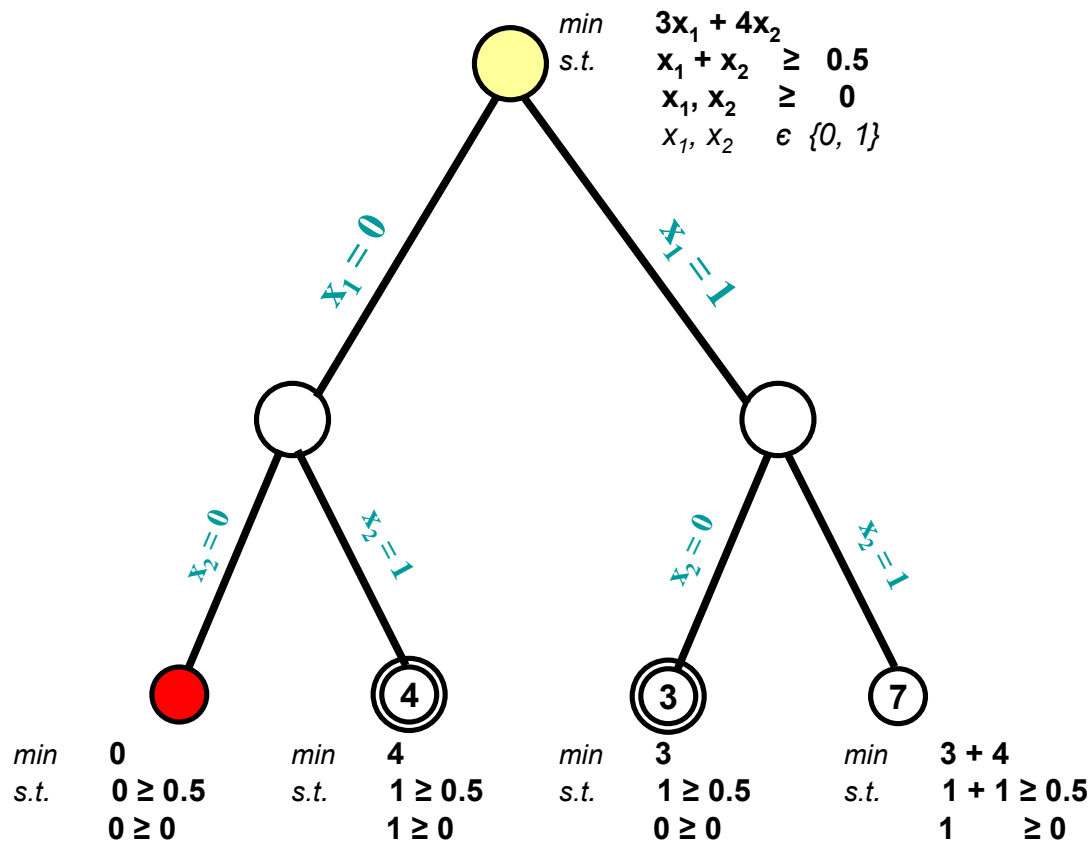
An enumeration tree is used to establish and keep track of the order in which assignments to the binary variables are considered. Figure 3 shows this tree structure. At every level an unassigned binary variable is selected; each tree branch is labeled with one of the assignments to that variable. This is called *resolving* the variable at that level. All nodes from that branch downward include the assignment of that variable to that value. Each leaf node at the bottom of the tree contains a complete assignment for all binary variables. Solving the LP that includes the assignments at any leaf may also provide a feasible, possibly non-optimal solution to the original binary BIP problem. The solution at any of these nodes would serve as an incumbent if the value at that node was more optimal than any previously discovered incumbent.

Figure 4 provides the pseudo code for the complete BB algorithm for BIPs. The variable **incumbent** stores the current incumbent node, which includes the particular assignment to the binary variables and the cost of the LP solution with those assignments. **U** stores the cost of the incumbent (or  $\infty$  when no incumbent exists). A queue **q** keeps track of the order in which the nodes in the search tree are explored. Note that nodes are added to and removed from **q** from the front. This LIFO policy results in a depth-first search exploration pattern. During the iteration of the main loop, **current** references the current node being examined that may be expanded into children nodes. Finally, the boolean flag **non\_integer\_found** is true during an iteration if and only if one of the binary variables is assigned a non-integer value in the solution to the LP. Certain parts of the pseudo code, such as the concept of relaxed problems and the pruning in line 10, will be discussed later in this section.

For each iteration, a node in the tree is selected (6) and the LP at that point is solved (7). The problem is framed as an LP instead of as a BIP by allowing any unassigned binary value to take on a *real* value between 0 and 1. The purpose of this relaxation is to identify prunable subspaces quickly, a process that is explained in Section 3.1.2. If the solution at the current node has all integer values for the binary variables and if the cost is not greater than or equal to the incumbent cost already found (checked on line 10), then the new solution provides an incumbent with lower cost (25-26). Otherwise, the algorithm creates a new level in the tree by branching into two descendant nodes (15-21). These new nodes are modified from the parent through the addition of constraints (17, 20). These constraints impose distinct integer values onto a binary variable (selected in line 14) that is assigned a non-integral value in the solution of the parent's LP. Only variables with non-integral values are considered for this assignment in order to ensure that no tree level adds dominated constraints that do not affect the solution; the rationale behind this selection criterion is explained more in Section 3.1.5. The two subproblems that result from adding the new constraints are inserted back into the queue (18, 21).

Figure 5 illustrates how the search process uses incumbent solutions at the leaf nodes of the tree to track the best global solution found thus far. The initial BIP problem with no assigned binary variables is shown to the right of the root node at the top (0<sup>th</sup>

level) of the tree. As the search progresses, each branch of the tree adds the assignment of 1 and 0 to a particular binary variable. Using a depth-first search policy, first the root node is expanded and the nodes on the 1<sup>st</sup> tree level are created by assigning values to  $x_1$ . The left node on the first level is then expanded to the 2<sup>nd</sup> level by assigning values to  $x_2$ .



**Figure 5:** Illustration of the search process using BIP B&B. Numbers in the circles are the cost of the LP solution at a particular node. Integer value assignments to the binary variables are shown along the tree branches. Double circles indicate an incumbent node and filled-in circles indicate infeasibilities.

The first node created at the second level of the tree (displayed as a filled in circle with no numeric value) contains the assignments  $x_1 = 0$  and  $x_2 = 0$ . These two assignments together with the original LP constraint  $x_1 + x_2 \geq 0.5$  result in an infeasible problem. Its sibling node, containing  $x_1 = 0$  and  $x_1 = 1$ , results in a solution cost of 4. At this point, no incumbent nodes have been found. Therefore, this node meets the first and second characteristic of incumbents: it assigns integer values to all binary variables, and it is more optimal than any previously discovered incumbents) characteristics of incumbent. Hence the node is stored in incumbent.



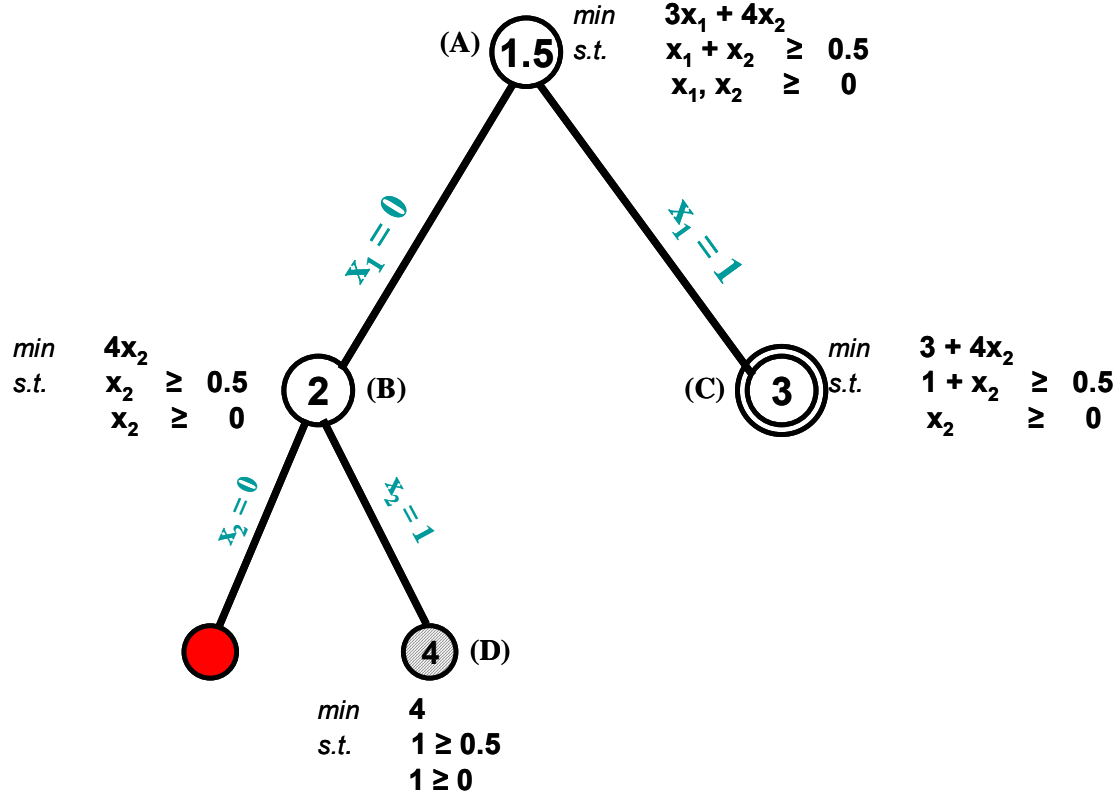
Since none of the nodes on the second level can be expanded, the algorithm expands the right branch of the 1<sup>st</sup> tree level next. The node with cost 3 (which assigns  $x_1 = 1$  and  $x_2 = 0$ ) becomes the new incumbent because it assigns integer values to the binary variables and is less costly than the last incumbent (which had an objective function value of 4). The final, rightmost node at the 2<sup>nd</sup> level assigns  $x_1 = 1$  and  $x_2 = 1$ . It therefore would not qualify as an incumbent even though it assigns integer values to the binary variables, because an incumbent with lower cost (its sibling with cost 3) has already been found. The globally optimal solution for this problem is the lowest-cost incumbent, the node with cost 3.

Note that the entire search tree contains approximately  $O(a^n)$  leaf nodes, where  $a$  is the average number of values that each integer variable can be assigned, and  $n$  is the number of such integer variables. Hence an exhaustive search over binary assignments is exponential in time. Branch & Bound speeds up this search process by pruning the search tree under several conditions, described in the next three sections.

### 3.1.2 Subspace Pruning through Bounding

The process of searching through the sets of binary variable assignments can be significantly sped up using the concept of *bounding*. As discussed in Section 3.1.1, the cost of an incumbent sets an upper bound on the cost of the overall solution to the BIP. If the algorithm can determine that all of the nodes in a particular search subtree will have higher cost than the current incumbent, the entire subtree can be left unexplored. This is called *pruning* the subtree, and is the “bound” part of the Branch and Bound algorithm. This pruning can only occur when an incumbent has already been discovered earlier in the search process. Pruning also requires a way to identify whether or not a subtree contains solutions with cost worse than the incumbent, and must perform this identification quickly.

The algorithm identifies subtrees that consist entirely of nodes worse than the incumbent by solving a *relaxed LP problem* at the root of each subtree. The solution of a relaxed LP at a node is an optimistic estimate of the cost of all nodes in the subtree below



**Figure 6:** Illustration of relaxed LPs and the use of bounds for pruning the search tree in B&B. Numbers in circles are the cost of the solution to the relaxed LPs, shown next to each node. Double circles indicate an incumbent node, filled-in circles indicate infeasibilities, and diagonal lines through a node indicate that the subtree rooted at that node is pruned.

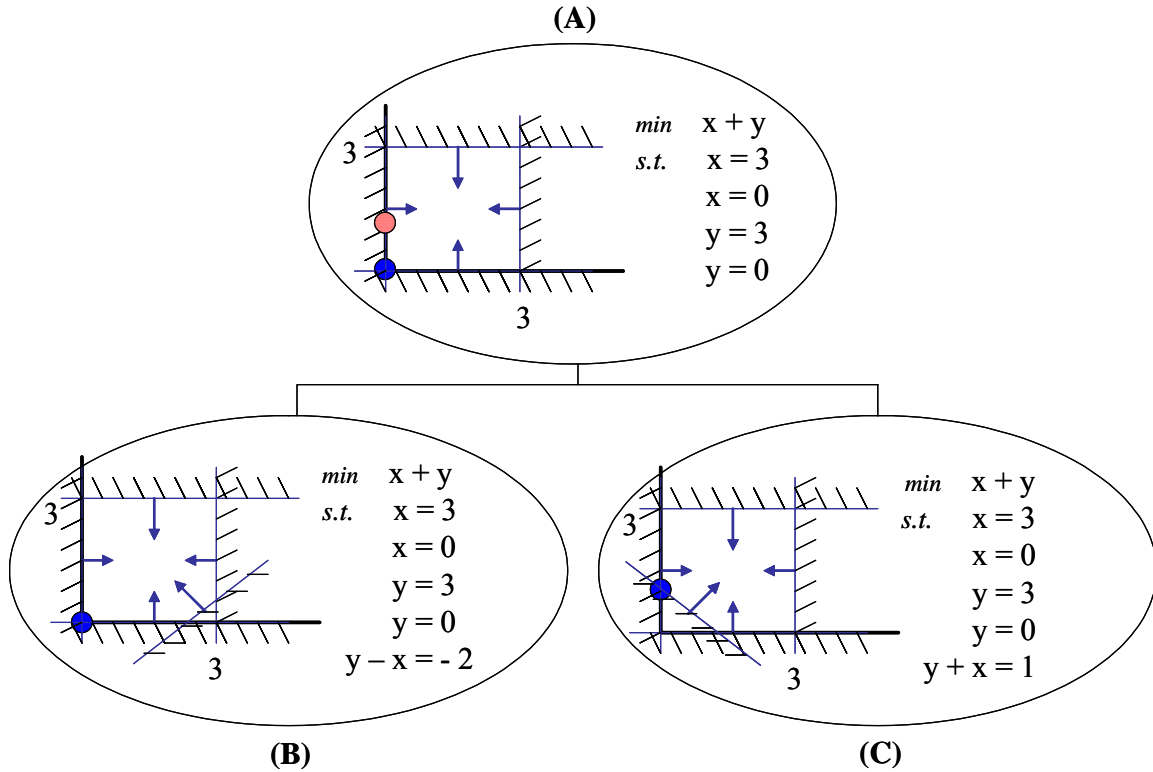
that node. If this optimistic bound is greater than the cost of the incumbent, then the subtree need not be expanded. Given a current node being visited during a search, a relaxed LP for the node is derived from the original BIP problem by *relaxing* all of the unassigned binary variables to be real-valued variables. More specifically, unassigned variables are allowed to have any real value between zero and one. As a result, only real-valued variables exist in the problem, hence the problem is an LP and can be solved using Simplex. In addition, the solution of this LP is guaranteed to be at least as good as the solution of any leaf node of that subtree.

Figure 6 illustrates the use of relaxed problems in the B&B search process. The most relaxed LP results from making all integer variables real valued (removing all

integrality constraints), and serves as the relaxed problem for the root node of the search tree. The node at the 0<sup>th</sup> level of the tree (A) contains the most relaxed LP that results from relaxing the integrality constraint for both  $x_1$  and  $x_2$ . Each deeper level of the tree is constructed by selecting a relaxed binary variable to resolve (by assigning it to be either 0 or 1) and leaving relaxed the remaining binary variables that are not already resolved at the parent node. For example, the first level of the tree in Figure 6 resolves  $x_1$  and leaves  $x_2$  relaxed. Next to each node in the 1<sup>st</sup> tree level is shown the relaxed LP contained in that node. In the next iteration, another fringe node is selected and one of *its* relaxed binary variables is resolved. For example, the subtree that extends node (B) resolves  $x_2$ , leaving no variables relaxed. In this way, all variables at any node are either relaxed or resolved. As mentioned, since only real-valued variables exist in every problem, an LP solver can be used to solve the problem at every node.

As pointed out earlier, using relaxed LPs enables the algorithm to identify subtrees that can be pruned because the solution to a relaxed LP at any node sets a lower bound on the cost of the LP solutions at all descendant nodes. At each level of the search tree, the only difference between a child node and its parent is the *addition* of a constraint that assigns a value to a previously real-valued binary variable. For example, the only difference between the nodes at Figure 6 (A) and (B) is the *addition* of the constraint  $x_1 = 0$ .

The list of constraints in the LP grows monotonically with the depth of the tree; that is, no constraint that restricts the feasible region is ever removed, and every descendant of a node has a feasible region that is at most as large as the ancestor's feasible region (the feasible region weakly decreases in size with tree depth). The objective function for all nodes is the same, and the optimal solution for each node is simply the point in the feasible region that has the lowest cost, based on that function. As a consequence, a child node will either have an optimal solution at the same point as the parent, or will have a more costly optimal solution at another point in the restricted feasible region.

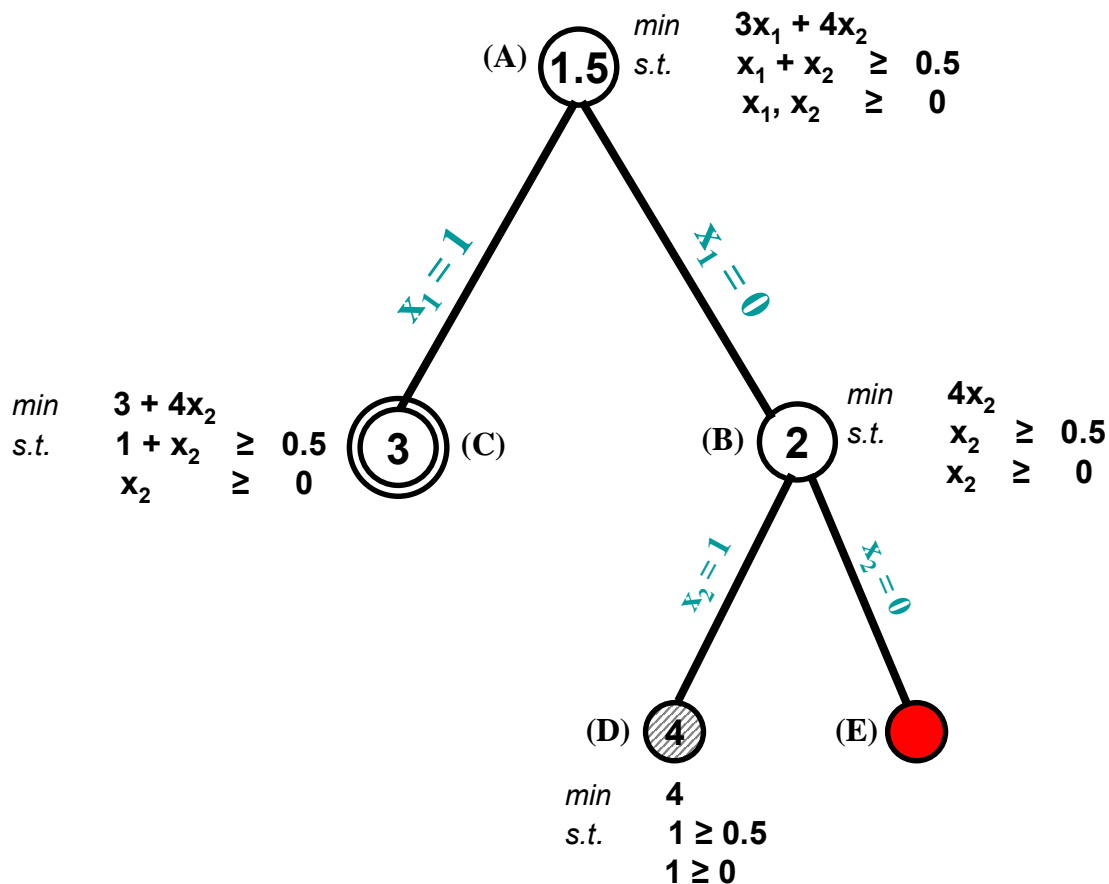


**Figure 7:** Illustration of how the LP feasible region either shrinks or remains the same in descendant nodes. The filled dot indicates the optimal solution for each node. Note that the optimal solution in (C) is also within the feasible region in the parent (A), indicated by the lightly shaded dot. Since it is in general true that no new points will become feasible in descendants, no descendant can ever have a more optimal solution than its ancestor.

For example consider Figure 7, which illustrates this idea using a minimization problem. The parent node (A) contains four constraints that bound a feasible region (arrows indicate the direction of the constraint, and hashed lines on one side of a line indicate the infeasible region). Within this region the optimal point is at  $(0, 0)$ , indicated by the solid dot. Note that the lightly shaded point  $(0, 1)$  is within the feasible region.

The first descendant node (B) adds the constraint  $y - x \geq -2$  to the constraint list, which shrinks the infeasible region. This does not affect the solution value, however, because  $(0, 0)$  is still feasible and no new points have been introduced. Descendant node (C) adds constraint  $y + x \geq 1$ , which makes  $(0, 0)$  infeasible; the new solution at  $(0, 1)$  is of greater cost. (C) could not have had a better cost solution than (A) since (C)'s feasible region is entirely contained within (A)'s. More generally, (A) sets a lower bound on the cost of the LP solutions at all nodes in its subtree.

As mentioned earlier, the B&B algorithm uses the upper bound set by an incumbent (from Section 3.1.1) combined with the lower bound set by the root node of any subtree to identify subtrees that can be pruned. If a node is discovered with cost greater than that of the current incumbent, this node is guaranteed to not be an optimal solution to the overall BIP, since its cost exceeds the upper bound. Further, all descendants of this node have greater cost than it, and therefore have greater cost than the incumbent's upper bound. Therefore, all descendant nodes can be ignored: the subtree that is rooted at the higher-cost node can be pruned.



**Figure 8:** Pruning process due to lower and upper bounds on the cost of solutions.

To summarize, relaxed LPs are a good way of identifying prunable subtrees. First, they can be used to discover subtrees that consist entirely of nodes that are guaranteed to not contain the optimal solution to the overall BIP. Second, because an LP solver can be used to solve the relaxed LP, they are efficient to use.

Figure 8 illustrates the overall pruning process for B&B. In this example, the assignment of a binary variable to 1 is incorporated into the first sibling, and the assignment to 0 into the second. The relaxed LPs at nodes (B) and (C) are solved first. Node (B) assigns  $x_1 = 1$  and  $x_2 = 0$ . It therefore becomes the first incumbent. The solution at node (C) assigns  $x_1 = 0$  (it is resolved) and  $x_2 = 0.5$ . All variables are not integral, so this node is not an incumbent. Node (B) is not expanded because a subtree rooted at an incumbent is guaranteed to not contain the optimal solution to the problem (see Section 3.1.4). When Node (C) is expanded next, one of its descendants is infeasible (E). The other descendant, node (D), has a higher cost (4) than the current incumbent's cost of 3. Therefore node (D) and all its descendants would be ignored and the subtree rooted at (D) – if there was one – would be pruned. In this particular scenario the subtree rooted at (D) consists of only a single node. The remainder of the search process in this example is discussed in later sections.

The pseudo code in Figure 4 shows the implementation of the process of pruning because of bounds. Line 10 performs an initial check to see if the cost of the solution at **current** is more than **U**, the cost at **incumbent**. If so, the algorithm returns to line 5 and performs no processing on **current**. Since no children are added to **q** and **current** has been removed from **q**, the entire subtree rooted at **current** is pruned.

### 3.1.3 Subspace Pruning using Infeasibility

Another method employed by B&B to improve search efficiency is based on identifying subtrees that consist entirely of nodes that are guaranteed to be infeasible. B&B prunes infeasible subtrees in a way similar to the pruning of suboptimal subtrees discussed in the previous section.

During the search process, the relaxed LP problem at a node may be found to contain a set of constraints that results in no feasible region. This is discovered by the LP solver. When this occurs, no leaf node below the infeasible node can provide a feasible solution to the original BIP problem. This follows by using the same argument that was used in Section 3.1.2. Recall that the feasible region of all descendant nodes are either smaller or the same size as the subtree root node. If the root node is infeasible, meaning the feasible region is empty, then all descendants must also have empty feasible regions.

Thus all descendant nodes are guaranteed to not provide a feasible solution to the BIP and can be ignored. This process of identifying prunable subtrees is efficient because it only requires solving a single LP.

As an illustration of pruning using infeasibility, consider Figure 8 again. (E) is infeasible because the selected constraints  $x_1 = 0$  and  $x_2 = 0$  are not compatible with the requirement that  $x_1 + x_2 \geq 0.5$ . Thus (E) and all descendants in its subtree (in this example the subtree consists of only one node) are ignored. Lines 8 and 9 in Figure 4 implement pruning based on infeasibility. They mirror lines 10 and 11, which were used for pruning based on bounds. If **current** is infeasible then the execution point restarts at line 5 and **current** is ignored.

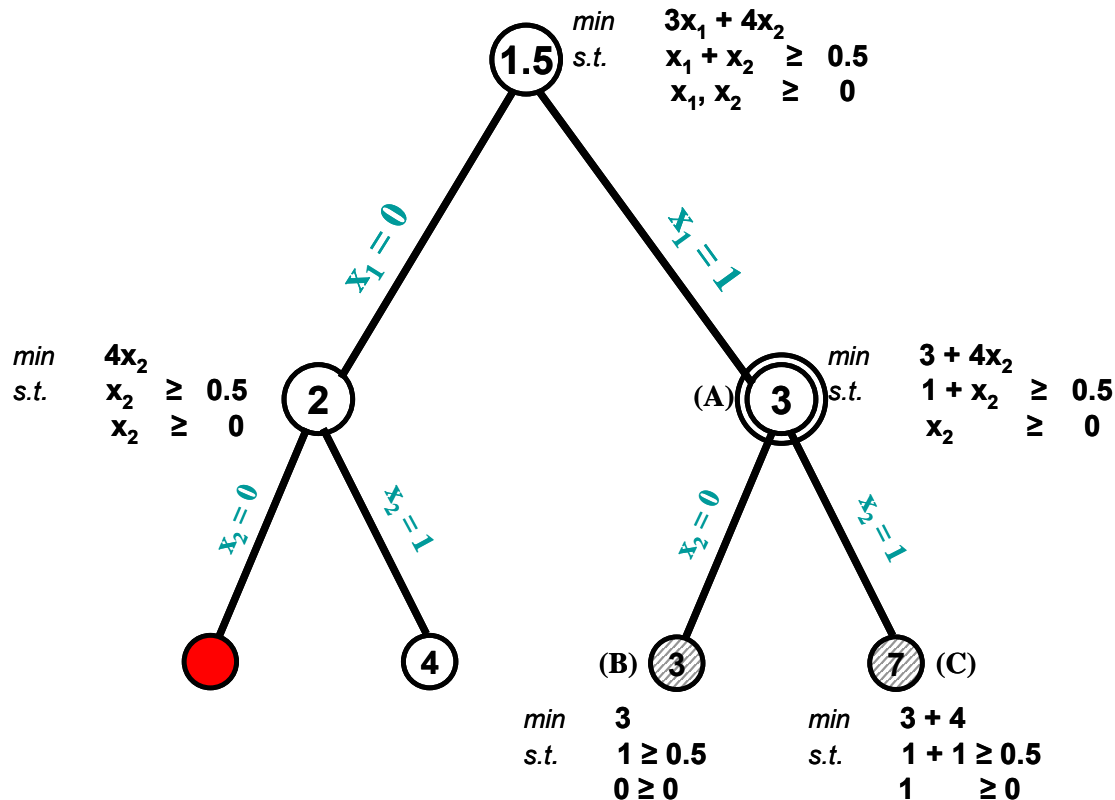
#### 3.1.4 Subspace Pruning using Exact Integer Solutions

This section reviews a third way of improving search efficiency through subspace pruning. This approach can be used when the exact optimal solution of a subtree can be identified from the relaxed solution at the subtree root. This allows the algorithm to avoid explicitly exploring and solving the LP at every node in the subtree. In particular, the relaxed LP at the root of a subtree may return as its optimal solution an assignment where all relaxed variables are given integer values. If this occurs, the relaxed solution at the root node must also be the optimal solution at all leaves of the subtree. .

If all the binary variables of a relaxed problem at a node N have been assigned integer values, then the only difference between the unrelaxed LP at N and the relaxed LP at N is that the unrelaxed LP includes constraints that require the relaxed variables to be integers. These constraints can have no impact on the solution to the relaxed LP, since this solution already assigns integer values to all variables. Therefore, the solution to the relaxed LP at N is the same as the solution to the unrelaxed problem at N.

For example, consider Figure 9, which completes the search process for the problem used in Section 3.1.2 and 3.1.3. At node (A),  $x_1$  is resolved and  $x_2$  is relaxed. Hence  $x_1$  is already assigned an integer value while  $x_2$  is not constrained. However, the optimal solution to the relaxed LP (shown beside the node) assigns  $x_2 = 0$ . Therefore the solution to the relaxed problem assigns integer values to all binary variables. The unrelaxed problem, which would require  $x_2$  to be 0 or 1, would still be solved by this

assignment. Since the *addition* of a constraint cannot improve the solution, the *optimal* solution for the unrelaxed problem is also  $(x_1 = 1, x_2 = 0)$ . The solution at (A) is an exact integer solution. None of (A)'s descendent nodes is a superior solution: (B) has the same cost and (C) is even more costly. Hence pruning the subtree rooted at (A) would not reduce the optimality of the overall discovered solution.



**Figure 9:** Illustration of B&B pruning based on the discovery of an exact integer solution. Double circles indicate an incumbent node with an exact integer solution, and diagonal lines through a node indicate that the node is pruned.

When B&B discovers an exact integer solution, it acts based on one two cases: either the solution has a cost equal to or greater than the cost of the current incumbent solution; or the solution has a lower cost than the current incumbent solution (which is identical to the case when no prior incumbent has been found). In the first case, the node does not improve on the incumbent, and so it is ignored and its subtree is pruned. In the second case, the newly discovered solution meets all the requirements of a new incumbent solution: it assigns integers to all binary variables, and has a lower cost than



any previously discovered incumbent solution. Hence it is recorded as the new incumbent and its subtree is pruned. In summary, a new exact integer solution is discovered exactly when a new incumbent is discovered.

In Figure 4, the process of pruning because of incumbent discovery is already reflected in lines 14-22. If `current` really is a new incumbent, it will be neither infeasible nor have a lower `cost` than `U`, and execution will proceed past lines 9 and 11. Additionally, no binary variable `j` will be discovered to have a non-integer value in line 14. Therefore, lines 15-21 are never executed, and so `non_integer_found` will still be the default value of `false` at the end of the loop. Lines 25 and 26 are executed, and so the discovery of the new incumbent is noted by updating variables `incumbent` and `U`. Finally, because lines 15-21 are never executed, the `insertEnd()` function will not be invoked for any children of `current`, and the subtree rooted at `current` is pruned.

### 3.1.5 Restricting Search Tree Depth using Active Variables

The final search efficiency improvement focuses not on subspace pruning, but on reducing the number of tree levels by restricting the possible choices for variable resolution. This is accomplished by not splitting on binary variables that are serendipitously assigned integer values by Simplex. More specifically, a binary variable is resolved in order to ensure that it has an integer value. If at least one relaxed variable is assigned a non-integer value, then at least one variable must be resolved in order to find the unrelaxed solution. Relaxed, unresolved variables that are assigned non-integer values at a particular node are called *active* variables at that node. The set of active variables is different for each node, based on the variable assignments. For example, node (B) in Figure 6 assigns  $x_2 = 0.5$ , and so  $x_2$  is an active variable at node (B). On the other hand, node (C) in the same figure assigns  $x_2 = 0$ , and so  $x_2$  is not an active variable at node (C).

If variable `v` is non-active at a node being expanded, then `v` does not need to be explicitly resolved for during the expansion. Since the variable is already integral, nothing is gained by resolving it and explicitly setting it to be integral. Furthermore, resolving other variables that are active may eventually result in a solution where no variables are active, without having ever explicitly expanded `v`. If this occurs, an exact

integer solution (as discussed in Section 3.1.4) has been discovered. This removes an unnecessary level of the subtree entirely, and reduces the total number of nodes that are explored. Therefore whenever a node is expanded, only the active variables at that node are considered for resolution. This is reflected in the pseudo code by the fact that block 15-21 in Figure 4, which adds a level to the tree, only occurs if the Boolean expression in line 14 returns true. Line 14 returns `true` only if the binary constraint on  $j$  is not met, which means  $j$  is active.

This concludes the review of B&B for BIPs. To summarize, B&B performs search by solving incrementally less relaxed problems at each level of the search tree. The discovery of infeasible nodes, incumbents, and nodes less-optimal than the current incumbent allow B&B to prune subtrees and thus reduce the total computational work necessary to solve the problem. To further increase efficiency, activity is used as a criterion for choosing which variable to branch on at every level. This ensures that each new node advances the algorithm towards discovery of a feasible solution with all binary variables assigned either 0 or 1.

## 3.2 Constraint-based A\* for Optimal SAT

Constraint-based A\* is an algorithm that builds from the concepts of DPLL SAT and A\* search to solve optimal satisfiability problems. There are also strong connections between C-b A\* and Branch and Bound. The primary feature of C-b A\* is the use of decision variables to explicitly select constraints to include in the LP that is solved at every node.

C-b A\* frames optimal CSPs as state space search problems by equating each search state with a (partial) assignment of values to a set of decision variables. To build a solution it solves for the unassigned variables in the problem by relaxing the corresponding constraints and solving the partially constrained problem. At each level it branches on the possible values for a particular decision variable, adding constraints to the children nodes. In this way the algorithm makes progress towards assigning values to all decision variables and arriving at an overall optimal solution.

The second key feature of C-b A\* is the use of mutual preferential independence to select an admissible heuristic. In the case where the cost function of the optimal CSP

is MPI, the cost of a particular state is minimized by minimizing the cost of each assignment at that state. Each estimated search state places a lower bound on the cost of the overall solution, since the current assignment may end up being inconsistent later in the search process. Since the estimated cost at each node is optimistic, the heuristic is admissible.

The final key feature of C-b A\* is the restriction on the number of children expanded at every tree branch. For every node that is expanded, only a single child node is created, corresponding to the child with the best (lowest-cost) assignment, out of all potential children nodes. This can be done because there must exist some leaf node of the least-cost child that has a more optimal solution than all leaf nodes of any other child. Therefore the best-cost unexplored state within the expanded node must be a leaf node of the least-cost child. Until one or more states within the least-cost child have been eliminated, C-b A\* therefore ignores all other children of the expanded node. When the least-cost leaf node has been eliminated, the next best sibling is expanded. This process significantly improves the efficiency of the search process, from  $O(nb-n)$  to  $O(n)$ , where  $b$  is the maximum variable domain size and  $n$  is the number of algorithm iterations.

### 3.3 Branch and Bound for Clausal LPs

This section details the unification of the Branch and Bound method discussed in Section 3.1 with the Constraint-based A\* of 3.2. These two methods are typically used to solve distinct types of problems: B&B as described is used for solving binary integer programming problems while constraint-based A\* is used for solving optimal decision problems with logical constraints, such as propositional clauses. Therefore the unified algorithm is perfectly tailored to solve the unified problem formulation presented in Chapter 2: clausal LPs. Clausal LPs offer an effective way to model hybrid decision/control problems, and the algorithm presented in this section, Clausal LP Branch and Bound (CL-B&B), in turn offers an effective way to solve Clausal LP problems.

The unified algorithm lifts a number of features from each of its parent algorithms. CL-B&B branches its search tree by resolving unsatisfied clauses in the same way as C-b A\* (Section 3.3.1), thus eventually arriving at a solution that solves all clauses. It solves relaxed problems iteratively (Section 3.3.2), enabling efficiency

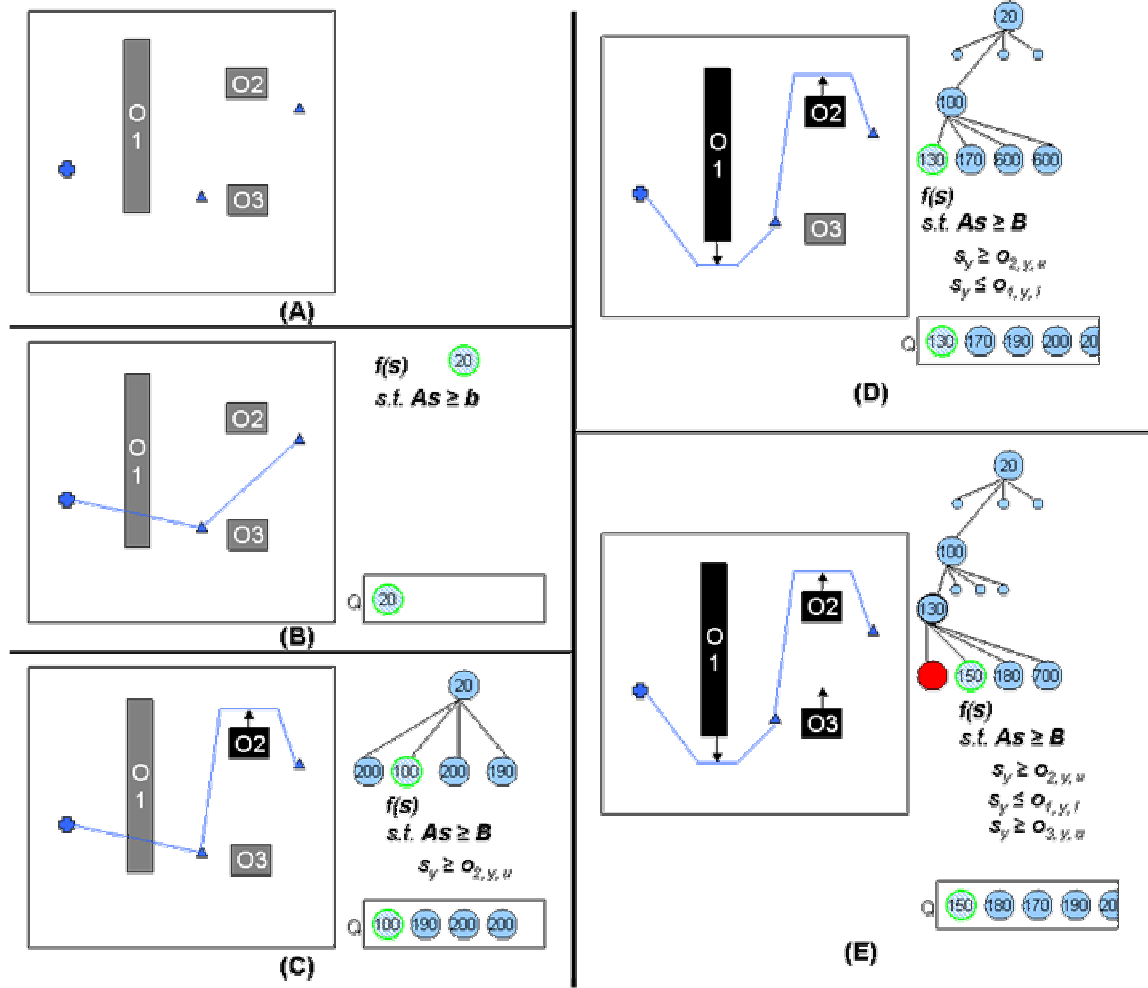
improvements through pruning. It restricts tree depth by using activity as a clause selection criterion (Section 3.3.3), like BIP-B&B. It also uses clause size as a secondary selection criterion in order to minimize the branching factor of the search tree (Section 3.3.4). Finally, it explores the search tree in a best-first order like C-b A\* (Section 3.3.5). All of these methods are implemented for Clausal LPs using modifications on concepts borrowed from the BIP-B&B algorithm introduced in Section 3.1. Pruning subtrees based on bounding and exact integer solutions is not implemented in CL-B&B because incumbents are not used (see Section 3.3.5); however, infeasibility-based pruning still occurs.

### 3.3.1 Search Tree Branching

The first key feature of the Clausal LP Branch and Bound algorithm is the construction of the search tree. A node is expanded by selecting disjuncts for resolving a previously unresolved clause. A clause is *resolved* at a particular node if one of the clause's disjuncts is explicitly included in the LP at that node. *Resolution* of a clause occurs by selecting one of the clause's disjuncts for inclusion in the problem. Thus during each iteration of the CL-B&B, a deeper level of the search tree is created by branching based on the disjuncts in the clause being resolved. Leaf nodes of the tree resolve all the clauses in the initial problem. Therefore each leaf node contains an LP that consists of the root node LP plus a distinct set of clause disjuncts. The best solution to the original Clausal LP is the best solution across all leaves.

This tree structure generalizes from the branching process utilized in Constraint-based A\*. In particular, CL-B&B adds a constraint to every node at every tree level, with each clause corresponding to a *decision* between constraints; this is similar to the assignment of values to decision variables in the branching process of C-b A\*. CL-B&B also solves optimal CSPs that start are complete relaxed of all decisions/clauses at the root of the search tree, and completely unrelaxed at the leaves. However, CL-B&B does not restrict the number of children expanded to only the optimal child; instead it uses a best-first search process (see Section 3.3.5) to restrict the node that it solves to that with the lowest cost.

The enumeration trees in the vehicle path planning problem of Figure 10 illustrate the branching process using obstacles which can be represented as clauses with four disjuncts each, as described in Chapter 5. Objective function values are inside of each node. Hashed/partially filled nodes indicate the current node, and solid nodes with no



**Figure 10:** Pictorial progression of tree expansion for the Branch and Bound for Clausal LPs. The vehicle path that results from the solution at the current node is shown on the map on the left at each step, with the cross representing the vehicle’s start location and the triangles representing waypoints. Lightly shaded obstacles without arrows are unresolved. The current enumeration tree is shown to the right of each map, with the cost of the relaxed LP solution written inside each node and the current node lightly shaded. Below the current node at each step is the relaxed LP at that node and the dynamically expanding queue, Q.

objective value indicate infeasible subproblems. The matrix equation “ $As \geq b$ ” is an abstracted view of the vehicle dynamics. At each tree level, the children represent avoidance of the obstacle by going (when reading left to right) South, North, East, and

**West.** Thus each expansion of the tree creates four branches to four children, each representing one of these selections. Moving from (B) to (C), for instance, exhibits the selection of “north” as the way to avoid obstacle 2 (or alternatively, to resolve clause 2). In the corresponding map in (C), the vehicle’s path adjusts to avoid obstacle 2 by going north of it. In (D), obstacle/clause 1 is resolved through the selection of “south”, and the map in (D) reflects this choice.

Figure 11 shows the pseudo code for CL-B&B. **current** and **child** once again store nodes of the enumeration tree, while **solution** and **cost** contain specific node components. Branching is handled between lines 13 and 19. Each **child** is an exact replica of the parent **current** (14) except for the inclusion of the constraint **disjunct** from clause **c** (15). **child** is then added to **q** if it is feasible (18). The process of determining *which* clause to branch on is discussed in Sections 3.3.3 and 3.3.4.

### 3.3.2 Relaxed Problems

The second key idea in CL-B&B is the use of relaxed problems midway through the enumeration tree. Like BIP-B&B, CL-B&B solves a relaxed problem at each node, and nodes at lower levels in the search tree are less relaxed than their ancestors. For CL-B&B, a *relaxed problem* is created by ignoring some subset of the original problem’s clauses that have not been resolved. This is in contrast to B&B for BIPs, which removes integrality constraints, and is similar to Constraint-based A\*, which relaxes any unresolved

clause at a particular node by removing it from the relaxed problem at that node. Recall that the leaves of the search tree consist of nodes that resolve all clauses and the root node (0<sup>th</sup> level) of the tree includes no clauses at all. Nodes in between the root node and

```

Clausal_BB(root)
1  cost = solution to relaxed LP at root
2  q = empty queue
3  q.orderedInsert(root)
4  while (q not empty) {
5      current = q.removeFirst( )
6      activeList = empty list
7      for each unresolved clause c in current
8          if (c is violated at current)
9              activeList.orderedInsert(c)
10     if (activeList empty)
11         return solution to current
12     c = activeList.removeFirst( )
13     for each disjunct in c {
14         child = current
15         add disjunct to child
16         compute relaxed LP at child
17         if (child feasible)
18             q.orderedInsert(child)
19     }
20 }

```

**Figure 11:** Branch and Bound for Clausal LPs

the leaves contain partially relaxed problems that resolve some of the disjunctive clauses and relax the rest. Every node in the tree also has an exact replica of the root node's LP (the objective function plus the non-disjunctive constraints).

As an example of relaxed problems in CL-B&B, again consider Figure 10. In the initial problem, the presence of obstacles O1, O2, and O3 impose constraints on where the vehicle (designated by the cross) can maneuver to. However, the algorithm first solves relaxed problems, higher in the search tree, while ignoring the constraints imposed by some of these obstacles (B). As the algorithm progresses, it incorporates more and more constraints and the LP problems that it solves get less relaxed (C, D). By the end of execution the algorithm is solving an LP that incorporates all of the obstacle-based constraints (E).

In Figure 11, the initial problem solved is the completely relaxed LP contained in root (1). At each newly constructed tree level, a particular **disjunct** is added to each **child** (15), which creates a slightly less relaxed problem than **current**. The relaxed LP at the new **child** is then solved (16) before insertion into the queue.

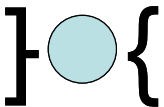
### 3.3.3 Restricting Search Tree Depth using Active Clauses

The third key feature of CL-B&B is that it applies the concept of activity to clauses in order to determine which clause to branch on during every iteration. This is in contrast to BIP B&B, where the branching takes place on a binary variable, based on variable activity. A clause is *active* at a particular tree node if and only if all of its disjunct inequalities are violated by the solution to the relaxed problem at that node. The principle behind this definition is similar to the definition of active variables. An inactive variable is one that does not need to be resolved because it is already satisfied by the current solution to the relaxed problem. Similarly, an inactive clause does not need to be resolved because it is satisfied by the current relaxed problem solution. Specifically, the clause is satisfied if at least one of its disjunct inequalities is satisfied by the current relaxed problem solution.

Figure 12 illustrates the concept of active and inactive clauses. The problem in this example is used again extensively in Chapter 4. The node being examined is a root node, with no resolved clauses. The list of clauses in the overall Clausal LP is listed to

the left of the node. The solution to the LP with no added disjuncts assigns  $x = 200$  and  $y = 200$ . At this solution point, three of the disjuncts in clause **c** are satisfied:  $x \geq 80$ ,  $x \geq 60$ , and  $x \geq 30$ . Therefore the current solution does not need to change at all to resolve clause **c**; it is possible that resolving just clauses **a** and **b** will result in a solution to the original unrelaxed clausal LP without explicitly resolving **c**. Thus **c** is non-active and is not a candidate for resolution. None of the disjuncts in clause **a** or **b** are satisfied and therefore both of these clauses are active.

Lines 6-12 in Figure 11 perform the selection of the branching clause using clause activity. **activeList** stores the list of active clauses and is initialized to be empty (6). For each clause **c** that is not *explicitly* resolved by having a disjunct included at **current** (7) CL-B&B adds it into **activeList** if it is active at **current** (8-9). If no active clauses exist (10-11), every clause is resolved by **current** and a solution has been found and is returned (Section 3.3.5 explains why a solution found here must be the globally optimal solution).

<u>Unresolved Clauses</u>	<u>Node</u>	<u>Relaxed LP</u>	<u>LP Solution</u>
<b>a:</b> $x \leq 100 \vee y \leq 50$ <b>b:</b> $x \leq 10 \vee y \leq 5 \vee y \leq 4$ <b>c:</b> $x \geq 80 \vee x \geq 60 \vee x \geq 30 \vee y \leq 0$		$\begin{array}{ll} \text{max} & x + 3y \\ \text{s.t.} & x \leq 200 \\ & y \leq 200 \end{array}$	$\begin{array}{l} x = 200 \\ y = 200 \\ f = 800 \end{array}$

**Figure 12:** Illustration of active and non-active clauses. To the left of the node is the list of clauses, with the non-active clause and its satisfied disjuncts faded and in italics. To the right of the node is the LP with no additional disjuncts, and to the far right is the LP solution at this node.

### 3.3.4 Ordering Clause Resolution based on the Fewest-Disjunct Clause Heuristic

The fourth key idea in CL-B&B is using size as a secondary criterion (after activity) for the selection of clauses to branch on. In the case where multiple clauses are active at a parent node, the algorithm selects the clause with the fewest number of disjuncts. This concept builds from the most-constrained variable heuristic used in other search algorithms [Russell & Norvig]. This heuristic selects for assignment the variable with the fewest remaining choices in its domain. It has the benefit of trying to delay assignment of variables with large numbers of choices, which would create many tree branches. The assignment of the most constrained variable sometimes restricts the



choices available for other variables and reduces the overall average branching factor of the search tree.

In the clause-selection domain, the analogous heuristic used is termed the *fewest-disjunct clause* heuristic. It has the benefit of trying to delay resolution of clauses with large numbers of choices, which would also tend to increase the tree branching factor considerably. This reduces the number of children created for the parent node; in general this tends to reduce tree width at shallower levels. As discussed above in Section 3.1, branch and bound algorithms will sometimes prune a subtree and not explore deeper levels. Therefore delaying the increase in tree width to deeper in the tree may have the effect of avoiding the width increase entirely; this will tend to improve search efficiency.

In addition to checking clause activity, lines 6-12 in Figure 11 also performs clause selection based on clause size. The insertion of clauses into `activeList` is ordered on the number of disjuncts, so clauses with fewer disjuncts are stored at an earlier position in the list. Therefore the first element of `activeList` will be the active clause with the fewest disjuncts. If an active clause exists, the first `c` of `activeList` is retrieved (12) and used for branching (13-19).

### 3.3.5 Utilizing a Best-First Search Order

The fifth and final key concept in CL-B&B, based on C-b A\*, is the order in which nodes in the search tree are explored. CL-B&B uses a best-first search order. This is in contrast to the depth-first order used by the BIP-B&B algorithm introduced earlier. A best-first order maximizes search efficiency by keeping the algorithm from deeply exploring a subtree that is known a priori to contain only high cost nodes, relative to other subtrees. However, use of BFS comes with some tradeoffs. For example, memory usage of BDS is  $O(b^d)$  where  $b$  is the maximum number of disjuncts per clause and  $d$  is the number of clauses. By comparison, depth-first search only uses  $O(b+d)$  memory. Insertion into a min-priority queue (see the implementation discussion below) also requires more time than the stack manipulation required for depth-first search.

Another result of using best-first search is that incumbents are not used. The reason is that any solution that is discovered in a BFS ordering and would qualify as an incumbent solution would also qualify as the optimal solution. This is because a solution

must assign integer values to all binary variables in order to qualify as an incumbent. But since nodes are extracted from the queue and examined in a best-first order, every examined node must also be more optimal than any node that has yet to be examined. Therefore the current incumbent node is a solution to the overall unrelaxed problem, and is the best solution possible; hence it can be returned as the optimal solution. As a result, incumbent nodes never need to be stored. Without incumbents, it is meaningless to try to prune using bounding (as discussed in Section 3.1.2) or prune using exact integer solutions (Section 3.1.4).

Certain elements of the pseudo code implementation reflect the modified search ordering. Instead of  $q$  being a simple stack that is pushed into and popped from, the algorithm uses a min-priority queue that orders nodes by cost (3, 18). Therefore lowest-cost nodes are resolved earlier. This requires that a node's relaxed problem is solved *before* insertion into the queue (16) instead of after removal. This also guarantees that as soon as a node that solves the problem is removed (i.e., as soon as any node with no active clauses is found), the algorithm has discovered an optimal solution and can terminate (11). Note that since the algorithm does not store incumbents, there is no reference to `incumbent` or `U`.

In Figure 10, each iteration of the algorithm (B), (C), (D) extracts the least-cost element from the queue,  $Q$ , for expansion. In (C), for example, the node with cost 100 is at the head of the queue, followed by the remaining nodes in increasing cost order. In (D), the node with cost 100 has been removed and expanded, while its children have been added. The ordering of the entire queue is based first on cost; therefore the new nodes with cost 130 and 170 are inserted before the older node with cost 190; the higher-cost children are inserted deeper in the queue. The algorithm halts in (E) because it has found a solution that resolves all obstacles, which is guaranteed to be the optimal solution because of the best first order.

## 4 Conflict-Directed Branch and Bound

An effective method for quickly solving logical decision problems is conflict learning, also known as conflict-directed search (CdS) [*citation*]. In CdS, when a solution is found to be infeasible, the source of the infeasibility is generalized and used to prune other infeasible portions of the state space. This generalization is represented as a partial assignment to variables, called a *conflict*. This chapter generalizes conflict-directed search to clausal LPs, building upon the Branch and Bound algorithm for Clausal LPs described at the end of Chapter 3. The resultant algorithm is called Conflict-directed Clausal LP Branch and Bound (CDCL-B&B). The integration of conflict-direction into B&B reduces the number of relaxed LP problems that must be solved, significantly improving the total time to find a solution. This is empirically demonstrated in Chapter 6. The efficiency improvement is accomplished by guiding the search progression away from areas of the search space that are known to be infeasible, based on previously discovered infeasibilities. Hence CDCL-B&B unifies the CD-A\* algorithm for optimal sat problems with BIP B&B analogous to the unification of Chapter 3.

Section 4.1 provides a high-level review of the Conflict-directed A\* algorithm for use in solving optimal logic decision problems. Section 4.2 then discusses the adaptation of CD-A\* and conflict learning to clausal LPs. Section 4.3 introduces in detail the central contribution of this thesis, CDCL-B&B.

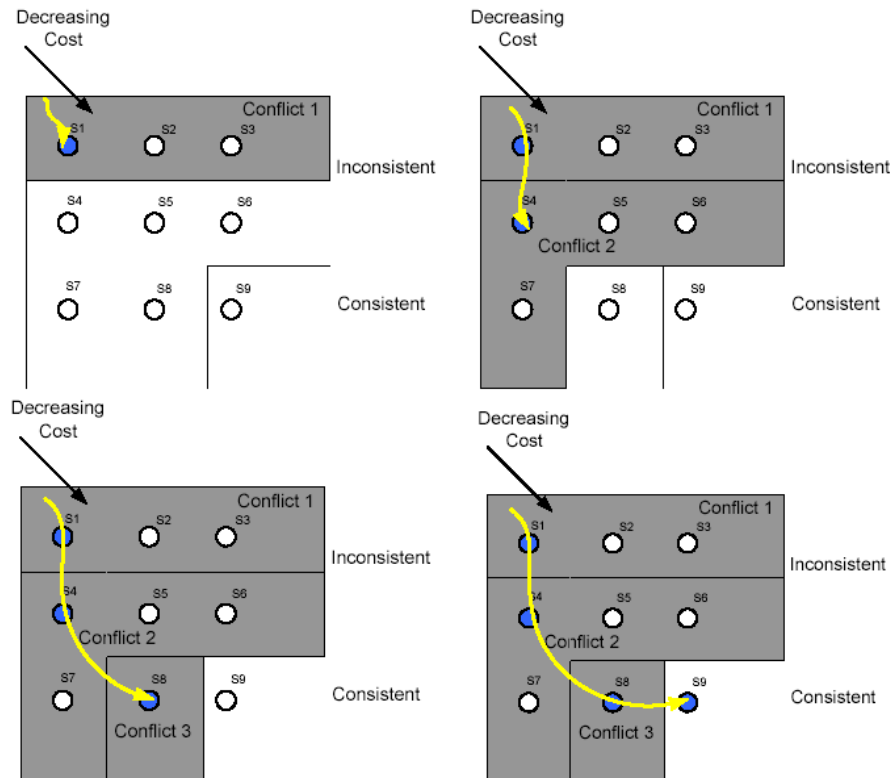
### 4.1 Review of Conflict-Directed A\*

The Conflict-Directed A\* algorithm utilizes the concept of a *conflict*, which can be informally thought of as a partial set of assignments that together result in infeasibility, to guide the search process to a solution to an Optimal CSP. Many of the ideas used in Conflict-directed Branch and Bound, such as conflicts, conflict-directed search, and constituent kernels, derive from the Conflict-Directed A\* algorithm. Therefore understanding the CDCL-B&B algorithm must be rooted in a review of CD-A\*. The material and figures in this section is based on [Williams & Ragno 2003].

Regular A\* search uses a heuristic to estimate the cost of every state in the search space. An admissible heuristic is one that provides an optimistic estimate of the cost. A\*

only searches states with an estimated cost that is less than the optimal solution. However, to guarantee optimality of the final solution, it must search every such node. In situations where some states are infeasible, the search process can be considerably sped up by generalizing individual infeasibilities into regions of the subspace that must contain only infeasible states. *All* of these states can then be avoided based on the result of examining a single infeasible state. In CD-A\*, these generalizations are partial assignments known as conflicts.

For example, consider Figure 13, which illustrates the search process used by CD-A\*. Whereas A\* 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 contained state.



**Figure 13:** Illustration of the search process used by Conflict-directed A\*. Unfilled circles are unexplored states, and filled circles are states that have been examined. The arrow indicates the order in which nodes are explored.

The main CD-A\* method is an interleaving of the generation of new *candidate solutions* with the testing of each solution. Since the search proceeds in a best-first manner (as in regular A\* search), discovering that a candidate is feasible means that the

candidate is an optimal solution. If a candidate is inconsistent, the inconsistency is generalized into a conflict, which identifies other states that are also guaranteed to be infeasible (this generalization is discussed further below). The search node expansion is organized such that the next candidate generated is the best candidate that resolves all known conflicts.

A key component of CD-A\* is the process of generalizing an inconsistent state into a subspace that is known to only include states with that same inconsistency. The first step in this process is to extract the conflict from the inconsistent state. One of the key advantages of CD-A\* is that it is flexible in regards to which algorithm is used to perform this expansion; any CSP algorithm that performs conflict extraction is suitable. The next step in the generalization is to map each conflict extracted so far to one or more constituent kernels. A *constituent kernel* is a minimal description of all states that *resolve* a particular conflict; that is, those states in which the conflict is guaranteed to not occur. In a CSP, for instance, the constituent kernel of a conflict is a set of assignments, each of which guarantees that the conflict cannot occur. More specifically, any state that contains the assignments of any one constituent kernel of a conflict is guaranteed to resolve that conflict.

For example, consider a set of variables  $V_1$ ,  $V_2$ , and  $V_3$ , each of which can be assigned two values, G or U. Suppose during the investigation process, the state that assigns  $\{V_1=G, V_2=G, V_3=U\}$  is examined and discovered to be infeasible. Suppose further that the conflict  $\{V_1=G, V_3=U\}$  is identified to cause this infeasibility through some conflict extraction process. In this case, one constituent kernel would be  $\{V_1=U\}$ , because it ensures that  $V_1$  would not be assigned the value G, and so the conflict would not arise. Another constituent kernel would be  $\{V_3=G\}$ , since it ensures that  $V_3$  would not be assigned U.

Recall that during the generate-and-test cycle of CD-A\*, generated candidates resolve *all* known conflicts. These candidates are constructed by first generating the concise descriptions, called *kernels*, of the states that resolve all known conflicts, and by extracting the best state from these kernels. In a CSP, kernels are constructed as the minimal set covering of the constituent kernels of known conflicts. Hence, each kernel contains a constituent kernel for every conflict. Finally, the next best candidate is created

from a kernel by assigning values to the variables that are still unassigned in the kernel. This is done by assigning each variable its best utility value, independent of all other assignments. Based on the property of mutual preferential independence (MPI) [reference], this method results in the next best candidate.

Continuing the above example, suppose another identified conflict is  $\{V2=U, V3=U\}$ , resulting in the constituent kernels  $\{\{V2=G\}, \{V3=G\}\}$ . The minimal set covering of these the sets of constituent kernels is  $\{\{V3=G\}\}$ , since this constituent kernel resolves all known conflicts.

The final key property of CD-A\* is the way in which kernels are constructed from the constituent kernels. In the worst case, the minimal set covering technique that is used to build the kernels is exponential in the number of decision variables. Therefore CD-A\* views the minimal set covering process as a search and uses A\* to find the most optimal state. In the search tree that is built for this search, the root node is resolves no conflicts, the leaf nodes are kernels, and the intermediary nodes are partial set coverings of the constituent kernels of the conflict. For example, in the example used above, the search process would begin with a partial assignment that resolved no conflicts,  $\{\}$ . Next, the least cost constituent kernel that resolves the first conflict would be identified,  $\{V3=G\}$ . At the next level of the search tree, the second conflict would be resolved, but the current partial set covering already resolves the second conflict. Thus the kernel identified would be  $\{\{V3=G\}\}$ .

## 4.2 Conflicts in the Clausal LP Framework

The final objective in this thesis is to generalize the process of CD-A\* to Clausal LPs. To accomplish this first requires defining the concept of conflicts as it pertains to linear programs. This section builds on the definitions of conflicts for satisfiability and CSP problems to define conflicts in the LP framework. Section 4.2.1 develops conflicts for Clausal LPs. Section 4.2.2 defines conflict resolution for Clausal LPs and describes how conflicts can help guide the Branch and Bound search process away from infeasible substates. The next section then develops the variant of CD-A\* for Clausal LPs.

#### 4.2.1 Defining Conflicts

Traditional conflicts in a CSP arise when a particular assignment of values to a subset of all variables is impossible. As described in Section 4.1, this occurs when two or more variable assignments are impossible to satisfy concurrently. Recall from Section 2.2.3 that a Clausal LP problem is analogous to CSP, where clauses are variables that are resolved by selecting a disjunct (“value”) from the list of disjuncts in the variable (“domain”). Viewed this way, a conflict in the Clausal LP framework occurs when a particular selection of disjuncts is impossible to satisfy concurrently. A set of disjuncts cannot be satisfied if the disjuncts’ inequalities together denote an infeasible region.

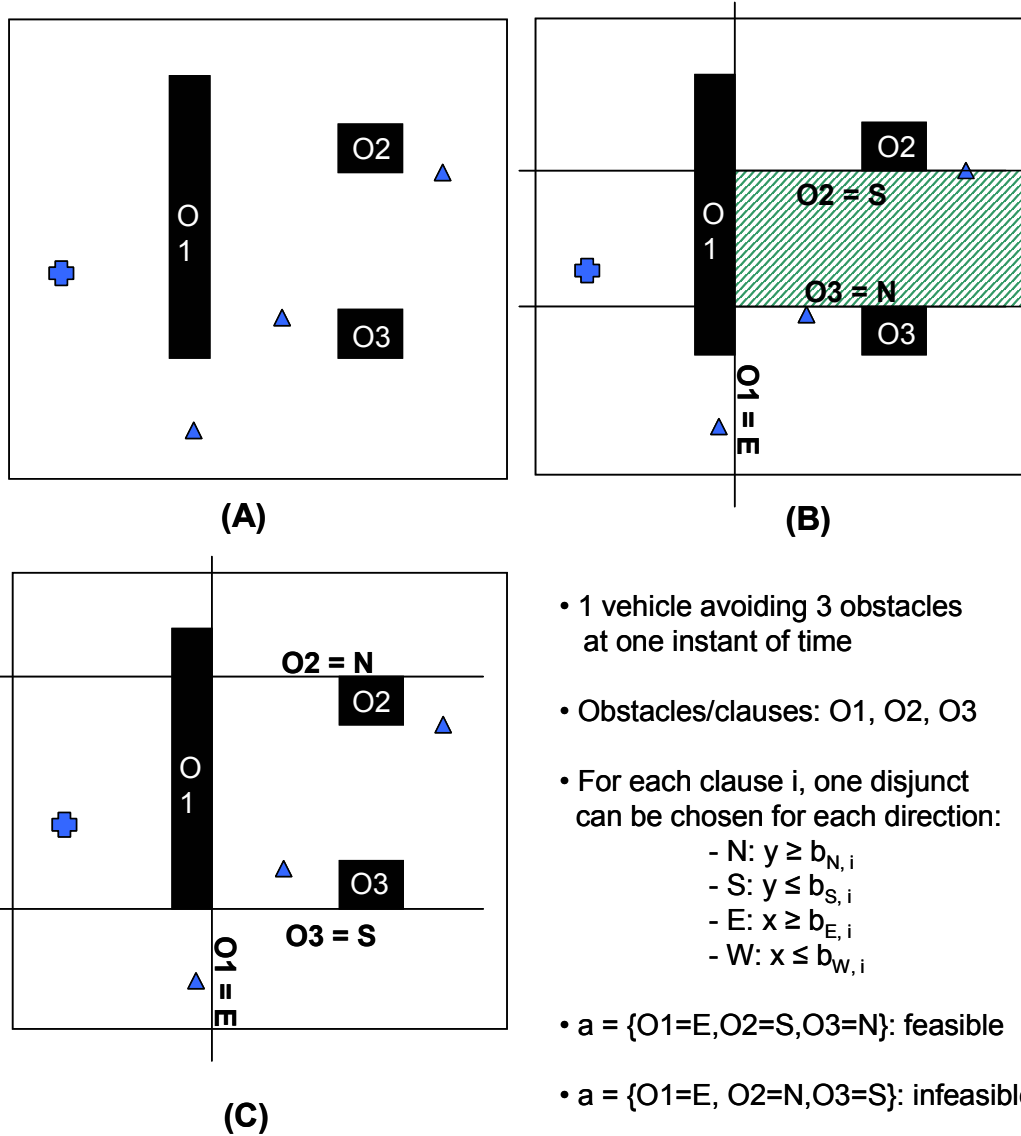
Clausal LPs consist of not just clauses of disjuncts, but also non-disjunctive constraints which must all also be met. However, these non-disjunctive constraints can be viewed simply as clauses with a single disjunct (unit clauses). Therefore the formal definition of conflicts need not distinguish between non-disjunctive constraints and clauses. Note that an infeasibility that exists purely as a result of non-disjunctive constraints renders the problem trivially unsolvable. CDCL-B&B assumes that these infeasibilities are caught during problem formulation, and that the problems presented for solving will contain only infeasibilities that involve at least one clause disjunct.

The formal definition for a conflict in the Clausal LP domain is:

*A conflict consists of a set of inequality constraints  $D$  from a Clausal LP CLP with the following characteristics:*

1. *Each inequality in  $D$  is a particular disjunct in a clause  $cl$  of CLP.*
2. *The inequalities in  $D$  result in an empty feasible region.*

Characteristic 1 says that each inequality constraint in the conflict must be a disjunct from a clause of the Clausal LP. Characteristic 2 identifies that a conflict arises due to an



**Figure 14:** Representation of obstacles (A) as clauses in a Clausal LP, and assignments that result in a feasible region (B) an infeasibility (C). The cross is a vehicle and the triangles are goal points.  $b_{x,i}$  denotes  $X$  the boundary of obstacle  $i$ .

infeasibility. For the remainder of this discussion, conflicts will be referenced in the form of a set  $\{\alpha_x, \beta_w, \dots\}$ . Each element of the set, such as  $\alpha_x$ , references a particular clause ( $\alpha$ ) and a particular disjunct within that clause ( $x$ ).

As an example of Clausal LP conflicts, consider the following scenario in the vehicle path-planning domain. Obstacles in this domain can be represented in the Clausal LP framework as clauses of disjuncts. To do this, an obstacle's outside boundaries are linearized and an inequality constraint is introduced to describe each linear segment. A



convex boundary is then described by a clause that is built from the disjunction of each linear element of the boundary (this is discussed in more detail in Chapter 5). In the simple case of a two dimensional square object, the boundaries linearize to four disjuncts. A vehicle can avoid the obstacle (and thus resolve the clause associated with the obstacle) by going to the north, south, east or west of the obstacle. Based on the exact location of the vehicle, these disjuncts are each associated with an inequality.

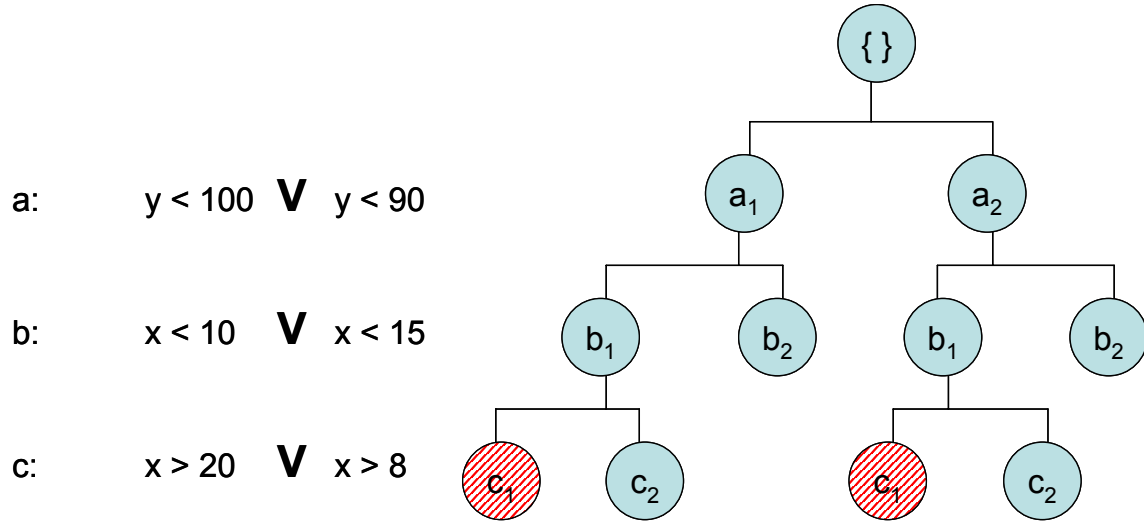
Figure 14(A) shows what would arise with three obstacles: O1, O2, and O3. As shown, there is one clause that must be resolved for each obstacle. In Figure 14(B), a particular selection of disjuncts for all the obstacles/clauses results in a feasible region, shown in diagonal lines. This means no conflict exists in this situation. Figure 14(C), on the other hand, shows a particular selection of disjuncts that results in no feasible region, because the constraints requiring the vehicle to be north of O2 and south of O3 cannot be concurrently met. Therefore, a conflict exists in this scenario, arising from disjuncts of clauses.

Two conflicts are possible in this case. The formal definition of a conflict places no requirement that the set of disjuncts  $D$  be of minimal size; it can include additional disjuncts that are not necessary for infeasibility. Therefore, the conflict in Figure 14(C) could consist of all three disjuncts  $\{O1=E, O2=N, O3=S\}$  or simply the minimal subset of these constraints that results in the infeasibility  $\{O2=N, O3=S\}$ .

#### 4.2.2 Efficient Search by Using Conflicts for Pruning

Just as conflicts are used to guide the search process in Conflict-directed A\*, the process of solving a Clausal LP can be guided using conflicts in order to identify areas of the search space that must be infeasible. Infeasibilities tend to appear repeatedly during the search process. For instance, suppose a search algorithm were exploring the tree shown in Figure 15. Suppose the left branch rooted at the  $a_1$  node is searched initially, and the infeasibility between  $c_1$  and  $b_1$  is discovered at the far left leaf node (designated by the node with diagonal slashes). Later in the search, when the right branch rooted at  $a_2$  is explored, the same infeasibility would be discovered again below  $b_1$  (also diagonally slashed). Without some form of conflict detection, the algorithm would not be able to identify  $\{b_1, c_1\}$  as a selection to avoid preemptively. Therefore the search

algorithm would waste time trying to calculate the solution to an LP that has no feasible region. If instead the algorithm records the  $\{b_1, c_1\}$  conflict discovered earlier, it can resolve the conflict and guide the search away from the infeasible subspace.



**Figure 15:** Illustration of multiple conflicts occurring while solving a Clausal LP. The disjuncts of the clause resolved at each level is shown to the left of the tree, with disjuncts numbered from left to right.

Recall that Conflict-directed A\* uses each conflict to create a *constituent kernel*, a set of assignments which are each sufficient to resolve the conflict. CD-A\* then uses these to guide the search process. Constituent kernels exist in the Clausal LP framework as well: a conflict's constituent kernel is a set of *selected disjuncts*, each of which is sufficient to resolve the conflict. A conflict is *resolved* by a disjunct when the inclusion of that disjunct in an LP problem guarantees that the LP will not contain at least one of the inequality constraints of the conflict. In general, resolving a conflict  $\{\alpha_x, \beta_w, \dots\}$  requires selecting either another disjunct from clause  $\alpha$  or (inclusively) another disjunct from clause  $\beta$ , and so on, for all other constraints in the conflict. Therefore the set of constituent kernels of a conflict is comprised of the alternate disjuncts for every clause that has a disjunct in the conflict.

For example, in the problem shown in Figure 15, the conflict discovered is  $\{b_1, c_1\}$ . To guarantee that this conflict does not appear in another node in the search tree, either  $b_1$  must be guaranteed to not appear, or  $c_1$  must be guaranteed to not appear. This can be accomplished either by resolving clause **b** without using  $b_1$ , or resolving clause **c** without using  $c_1$ . Once the clause containing either disjunct is resolved some other way,

there is no reason for the conflicting disjunct to be selected. Therefore, one possible constituent kernel for the conflict is  $b_2$ : this resolves clause  $b$  and ensures  $b_1$  will not appear in the LP. Another one is  $c_2$ , since it will resolve clause  $c$ . The total set of states that resolve the conflict  $\{b_1, c_1\}$  is therefore defined by the disjunction of the constituent kernels:  $b_2 \vee c_2$ .

Conflict-directed A\* also introduced the concept of a kernel (as opposed to a constituent kernel), which resolves all known conflicts. A kernel is built up from the set of currently identified constituent kernels. In the Clausal LP domain, a *kernel* is the set of selected disjuncts that resolve *all* known conflicts, and so contains one constituent kernel for every known conflict. However, instead of immediately creating a single candidate that resolves all conflicts as CD-A\* does, CDCL-B&B resolves conflicts incrementally, as will be discussed in Section 4.3.4.

### 4.3 Conflict-Directed Clausal LP Branch and Bound

Recall that conflict-directed search is based on the utilization of discovered infeasible states to avoid future infeasible regions of the search space. This section describes the use of conflicts in the Clausal LP framework, as defined in Section 4.2.1, for increasing the efficiency of the Clausal LP Branch and Bound algorithm described in Chapter 3. Section 4.3.1 introduces the idea of a two-mode search process, one mode for normal best-first search (BFS) without conflicts, and one mode for resolving conflicts. It also walks through the BFS mode pseudo code. Section 4.3.2 outlines the extraction and storage of conflicts in a global database for use by all the search nodes of the algorithm. Section 4.3.3 explains the policy that is used to retrieve from the global database the conflicts that are relevant at a particular node. Section 4.3.4 details the conflict-resolution mode of the search process, which resolves multiple conflicts incrementally before solving any relaxed LP problems. Folded within this section is the use of propositional unit propagation as a way to further narrow the search.

#### 4.3.1 Separating Best-first Search from Conflict Resolution

The overall search process of CDCL-B&B is divided into two modes of operation. The first mode (“BFS mode”) performs best-first search over normal search

tree nodes, and solves relaxed LPs at each node. This process builds upon the search structure of the Branch and Bound algorithm for Clausal LPs, discussed in Section 3.3. The conflict-directed search extension diverges from CL-B&B only when an infeasible node is found. When this takes place, the subtree is pruned as normal, but a conflict is also extracted from that infeasibility. The second mode of operation (“conflict resolution mode,” or “CR mode”) uses the extracted conflicts to guide the search process, without having to solve relaxed LPs.

#### 4.3.1.1. *The Two-Mode Algorithm: Motivation and High-Level Description*

The need to intersperse conflict resolution and normal BFS, like CD-A\* does, motivates the use of the two different search modes in CDCL-B&B. Recall that each state in the search space is comprised of a set of LP constraints, with at least one selected from every clause. Conflicts are used to describe subsets of the search space whose states are all known to be infeasible because the state contains a combination of constraints that cannot be concurrently satisfied. This means that search tree nodes corresponding to elements or subsets of these subspaces can be ignored without invoking a costly LP solver. It makes sense to use these cost-saving conflicts as early as possible in the search process, in order to minimize the amount of time wasted searching infeasible nodes. This suggests that before any node is expanded into children that might be infeasible, all known conflicts should be resolved. This parallels Conflict-Directed A\*, in which the next candidate assignment generated is one that resolves all known conflicts.

The first mode proceeds in a manner similar to the CL-B&B algorithm (from Section 3.3), but also performs conflict extraction and recording whenever it discovers an infeasibility. This first mode is like the generation of the next-best candidate in CD-A\*, with each node akin to a *partial* candidate. Whereas a candidate in CD-A\*, if feasible, assigns values to all variables in the problem, an arbitrary node in BFS mode solves a *relaxed* problem and hence assigns values to only a subset of all clauses.

The second mode expands the search tree through conflict resolution prior to solving any relaxed LP, and is like the kernel generation process in CD-A\*. As argued above, all known conflicts are resolved before returning to the first mode. The key difference between CR mode in CDCL-B&B and kernel generation in CD-A\* is that it is

easy to identify the most optimal next child that resolves all known conflicts in CD-A\* but not in CDCL-B&B. CD-A\* uses MPI to move from a kernel to a candidate in linear time, but identifying the optimal next child for a Clausal LP requires solving the LP contained in every potential optimal child. This would tend to undermine any efficiency improvement from conflict-directed search. Therefore, CR mode searches systematically and performs propositional unit propagation to limit the number of potential next-best children. It does this by pruning states that contain known conflicts and constraining the expansion of some nodes to ensure that known conflicts do not arise. Section 4.3.4 describes CR mode in more detail.

Figure 16 through Figure 35 illustrate the complete CDCL-B&B algorithm. Figures containing images distinguish between BFS expansions (solid lines) and CR expansions (dotted lines). The pseudo code for CDCL-B&B is divided into two major methods, to reflect the two modes. Figure 16 outlines BFS mode and Figure 24 shows the `resolveConf`, which performs CR mode search.

#### 4.3.1.2. *Implementation and Example of BFS Mode in CDCL-B&B*

The two separate modes of search require nodes with different characteristics. These nodes are distinguished as “LP” and “conflict resolving” (“CR”) nodes. An LP node requires solving an LP, and is used in the first mode. A CR node does not require solving an LP, and instead is used only to resolve conflicts in the conflict resolution mode (CR nodes are explained in more detail in Section 4.3.4). In the accompanying figures, LP nodes are indicated by circles and CR nodes by squares. All references to nodes in Figure 16 refer to LP nodes, while Figure 24 uses only CR nodes.

```

CDCL-B&B(f, clauses_unresolved)
1   root_constraints = {disjuncts d | d is in a unit clause}
2   root_unsolved = <f, root_constraints, clauses_unresolved>
3   q = empty queue
4   if (! solve&EnQ(root_unsolved, q))
5       return null
6   cDB = empty map
7   while (q not empty) {
8       current = q.removeFirst( )
9       activeList = empty list
10      for each c in current.clauses_unresolved
11          if (c is violated by current.state)
12              activeList.orderedInsert(c.length, c)
13      if (activeList is empty)
14          return <current.cost, current.state>
15      conflicts = retrieveConfs(current, cDB)
16      children = resolveConfs(current, conflicts)
17      if (children not empty) {
18          for each child_unsolved in children
19              solve&EnQ(child_unsolved, q)
20          go to line 7
21      }
22      c = activeList.removeSmallest( )
23      for each disjunct in c {
24          child_unsolved = copy of current
25          add disjunct to child_unsolved.constraints
26          if (! solve&EnQ(child_unsolved, q))
27              cDB.insert(extractConf(child_unsolved))
28      }
29  }

```

**solve&EnQ(unsolved, q)**

```

1   relaxed_lp = <unsolved.f, unsolved.constraints>
2   feas = true if relaxed_lp feasible, false otherwise
3   cost = cost of the solution that solves relaxed_lp if feasible, infinity otherwise
4   state = state that solves relaxed_lp if feasible, null otherwise
5   solved = <unsolved.f, unsolved.constraints, unsolved.clauses_unresolved, cost, state>
6   if (feas) {
7       q.orderedInsert(solved.cost, solved)
8       return true
9   } else
10      return false

```

**Figure 16:** Pseudo code for main execution of Conflict-directed Branch and Bound.

LP nodes symbolize subproblems that are expanded into descendant nodes by solving a relaxed LP. Since execution of an LP solver is a costly step, efficiency is likely increased by minimizing the number of LP nodes in the tree. LP nodes are equivalent to nodes in the search tree of the B&B algorithm applied to Clausal LPs (Section 3.3). These nodes contain all of the same elements: an objective function, a list of constraints,

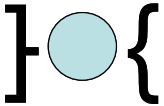
and a set of unresolved clauses. Some of these clauses might be inactive; that is, some clauses may contain one or more already selected disjuncts. Once an LP node's relaxed LP has been solved, the node also contains solution values for each variable and the resulting objective function value. The final datum stored in an LP node is the time of its creation, which is used to keep track of which conflicts remain unresolved (see Section 4.3.3). This time is set to zero at the start of the algorithm's execution, and is incremented by 1 after each additional search tree level is created during BFS mode.

The assignment of the variables in the solution to the relaxed LP allows the identification of which clauses are active, similar to CL-B&B. The definition of active clauses for CDCL-B&B is the same as for CL-B&B: active clauses at a node are those that consist entirely of disjuncts that are unresolved by the solution to the relaxed LP at that node.

$$\begin{array}{ll}
 \max & x + 3y \\
 \text{s.t.} & x \leq 200 \\
 & y \leq 200 \\
 & x \leq 100 \vee y \leq 50 \\
 & x \leq 10 \vee y \leq 5 \vee y \leq 4 \\
 & x \geq 80 \vee x \geq 60 \vee x \geq 30 \vee y \leq 0
 \end{array}$$

**Figure 17:** A hybrid decision-control problem modeled as a Clausal LP.

For example, consider the problem shown in Figure 17. This problem will be used throughout the rest of this chapter to illustrate the concepts of CDCL-B&B. The initialization of the search process (Figure 18) solves the root node relaxed LP, which relaxes all clauses and contains only the non-disjunctive (or unit clause) constraints. The LP solution at the root node is ( $x=200$ ,  $y=200$ ). Therefore clause  $c$  is not active at the root node, since the first three of its disjuncts are already satisfied by the root node assignment  $x=200$ . The satisfied disjuncts of clause  $c$  are italicized and faded. Because at least one disjunct of  $c$  is satisfied, the clause is inactive, as indicated by being faded in the figure. Clauses  $a$  and  $b$  consist entirely of disjuncts that are not satisfied by the root node solution, and are therefore active.

<u>Unresolved Clauses</u>		<u>Node</u>	<u>Relaxed LP</u>	<u>LP Solution</u>
a:	$x \leq 100 \vee y \leq 50$		$  \begin{array}{ll}  \max & x + 3y \\  \text{s.t.} & x \leq 200 \\  & y \leq 200  \end{array}  $	$  \begin{array}{ll}  x = 200 \\  y = 200 \\  f = 800  \end{array}  $
b:	$x \leq 10 \vee y \leq 5 \vee y \leq 4$			
c:	$x \geq 80 \vee x \geq 60 \vee x \geq 30 \vee y \leq 0$			

**Figure 18:** Initialization of the CDCL-B&B algorithm with root node and initial LP solution.

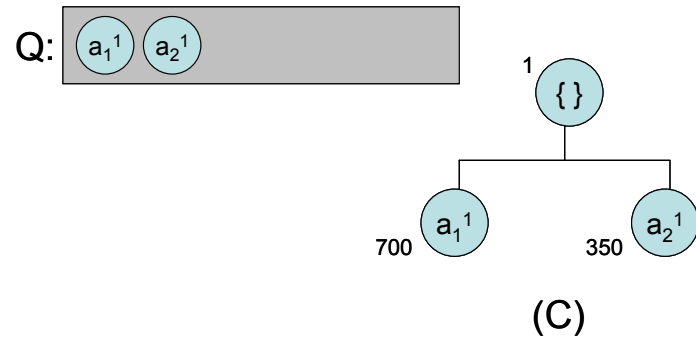
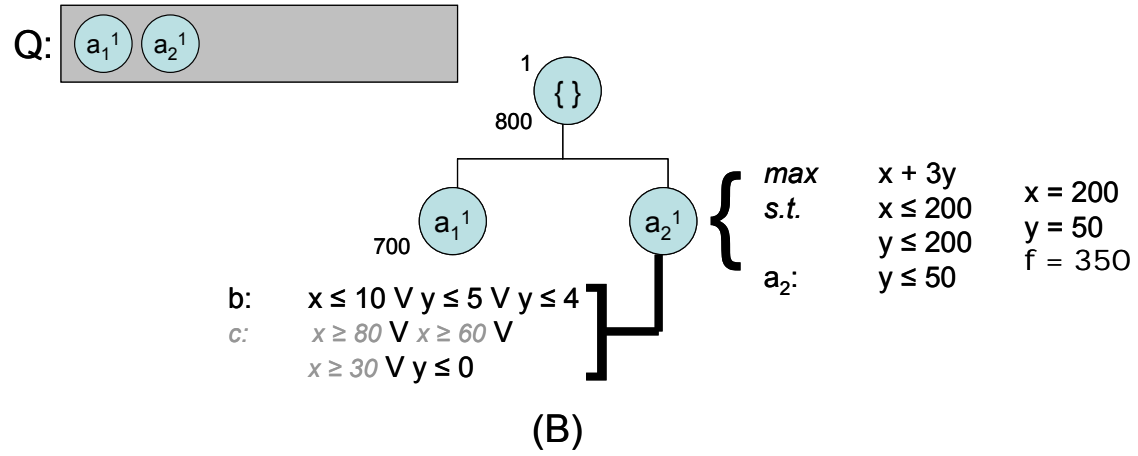
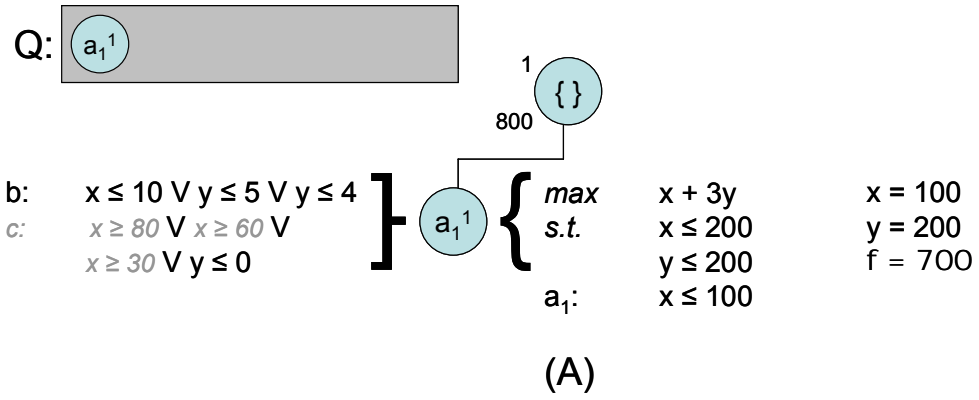
Figure 19 shows the first expansion of the search tree from the single root node. The expansion at this stage is like that in CL-B&B, and resolves the first of the active clauses from Figure 18, clause **a**. **a** is selected first because it has fewer disjuncts than clause **b**. Resolution of **a** in Figure 19 creates two children: one that includes disjunct  $a_1$ , in Figure 19(A), and one that includes disjunct  $a_2$ , in Figure 19(B). Figure 19(C) shows the state of the search tree after the first level is done expanding. The new children are inserted into the queue, **q**, after their relaxed LPs are solved (required for insertion into a priority queue, as discussed in Section 3.3.5).

In this and future figures, the newly added disjunct at every node is shown within the node (for example  $a_1$  or  $a_2$ ). The value of the objective function at each node is shown to the lower left of the node (for example, **800** in the case of the root node). The time step at which a parent node is expanded into its children is shown to the upper left of the node (**1** in the case of the root node). The time step at which a node is created is shown as a superscript within the node (both the nodes created at this expansion are created during time step **1** and so have a superscript **1**). Also displayed is the priority queue that is used to determine which node to expand next; nodes earlier (farther left) in the queue are those with greater value (since this is a maximization problem) and are expanded first. As in Figure 18, the faded disjuncts are those that are resolved by the current node and the faded clauses are inactive.

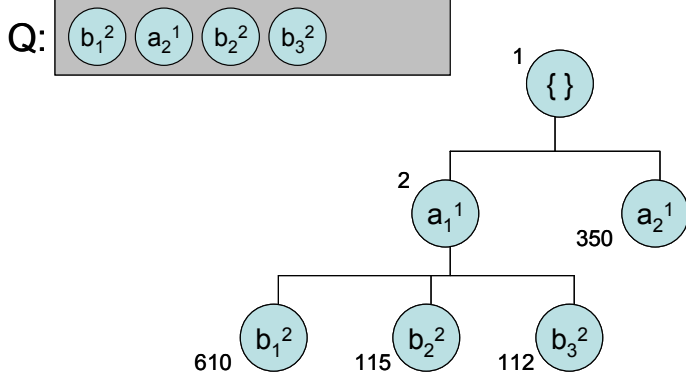
At the end of the first expansion, the node that added  $a_1$  is at the head of the queue, since it has the highest objective function value (**700**) of all nodes. Hence, this node is expanded next. Figure 20 continues the illustration of the algorithm's execution by showing the end result of this expansion, equivalent to Figure 19(C). As the expansion takes place, the newly created nodes are inserted into the queue based on their solution values. Therefore the node that added  $b_1$  at time step 2, having the highest solution value (**610**), is at the head of the queue and will be expanded next. The other newly created nodes have values lower than that of  $a_2^1$ , and so are inserted later in the queue.

The rest of the expansion process in this example is described in Sections 4.3.2 and 4.3.4.





**Figure 19:** First normal-mode expansion of search tree in CDCL-B&B algorithm.



**Figure 20:** Queue and search tree at the end of the second expansion in the execution of CDCL-B&B.

#### 4.3.2 Identification and Global Recording of Conflicts

Conflicts are extracted throughout the search process and are maintained independent of the specific node at which they are discovered. Each node is guaranteed to resolve all conflicts that were discovered prior to the creation of that node. Resolving all conflicts requires that conflicts be stored globally and be accessible by all nodes.

The global storage of conflicts is implemented in the CDCL-B&B pseudo code (Figure 16) using `cDB`. `cDB` is an associative database (called a “map” in the pseudo code) that contains the list of conflicts discovered up to the current point in the execution process. The conflicts are sorted and are accessible by their time of creation. LP nodes also store their time of creation, so it is easy to identify only those conflicts that were discovered after a particular node was created. Section 4.3.3 describes how these time stamps are used to keep track of what conflicts remain to be resolved. To achieve the time-conflict association, `cDB` maps from integer time stamps to conflicts created at that time.

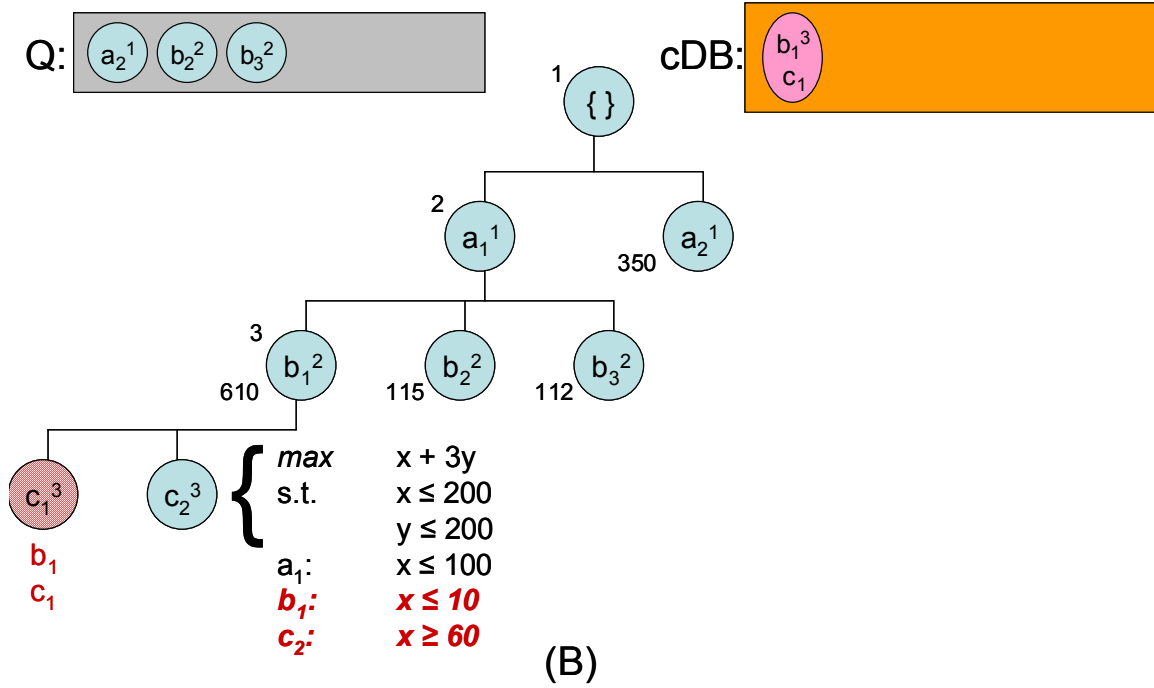
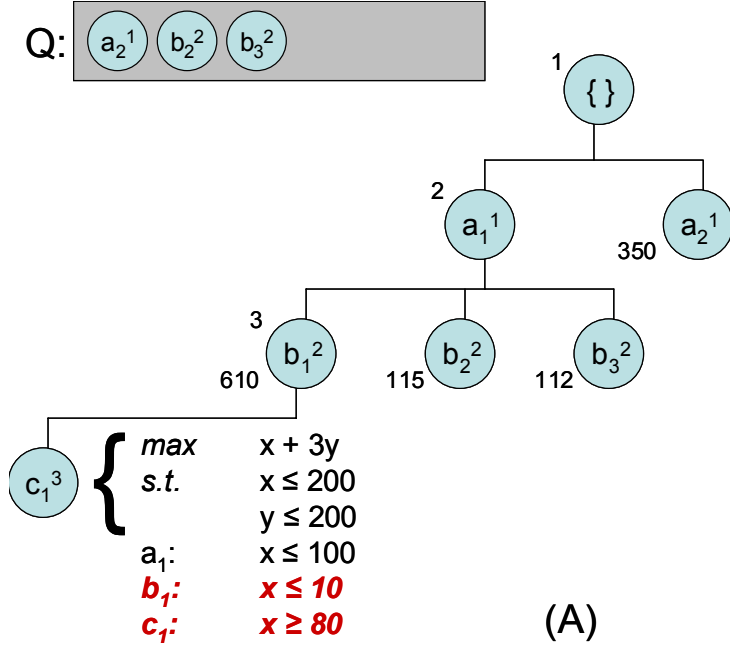
`cDB` is used in two places in the CDCL-B&B pseudo code. First, it is passed to the second search mode sub process (`resolveConf`, at line 13). Within that sub process it is accessed only to retrieve the conflicts that must be resolved. Second, it is accessed whenever an infeasibility is discovered. When this occurs, a conflict is generated from the infeasibility and stored in the database (line 29 of **CDCL-B&B**).

Recall from the discussion in Section 4.2.1 that multiple conflicts can sometimes be generated from a single infeasibility. The particular conflict that is generated and

stored can have significant impact on the efficiency of the search; in general, conflicts made up of fewer clause disjuncts guide the search better. Therefore the algorithm tries to extract a minimal conflict; that is, one such that no proper subset is also a conflict. Conflicts are generated (or *extracted*) using the `extractConf` method, and are then stored in `cDB` (29). A more detailed explanation of the need for small conflicts, the pseudo code of `extractConf`, and a comparison of different conflict generation methods are contained in Chapter 6. For the purposes of the discussion in this chapter, it is assumed that the minimal-size conflict will always be discovered and stored.

Continuing the example from Figure 20 illustrates the conflict extraction and storage process. Figure 21 shows the discovery of infeasible relaxed LPs and the process by which the CDCL-B&B algorithm extracts conflicts in response. As mentioned previously,  $b_1^2$  is the next node scheduled for expansion after the first two tree expansions. Since clause 3 is the only active clause at this node (in fact it is the only clause left in the entire problem), it is resolved next. The first child added in this expansion is therefore augmented with the inequality  $c_1$ , and is created at time step 3, as shown in Figure 21(A). However, the inclusion of the  $c_1$  inequality  $x \geq 80$  is impossible to satisfy with the  $b_1$  constraint  $x \leq 10$ , added at the previous tree level. Therefore an infeasibility is discovered at  $c_1^3$ .

In Figure 21(B), the infeasibility is translated into a conflict that identifies the source of the infeasibility:  $b_1$  and  $c_1$ . This conflict is stored in the database, labeled `cDB`.  $c_1^3$ , as an infeasible node, is pruned: it is not added to the queue and is not scheduled for expansion (in the figure this is represented with diagonal slashes through the node). Note that the time of the discovery of a conflict is recorded to its upper right within `cDB`. Expansion of  $b_1^2$  then continues with the creation of the next child, augmented with the inequality of  $c_2$ . However this node is also infeasible and a very similar conflict is generated:  $\{b_1, c_2\}$ .

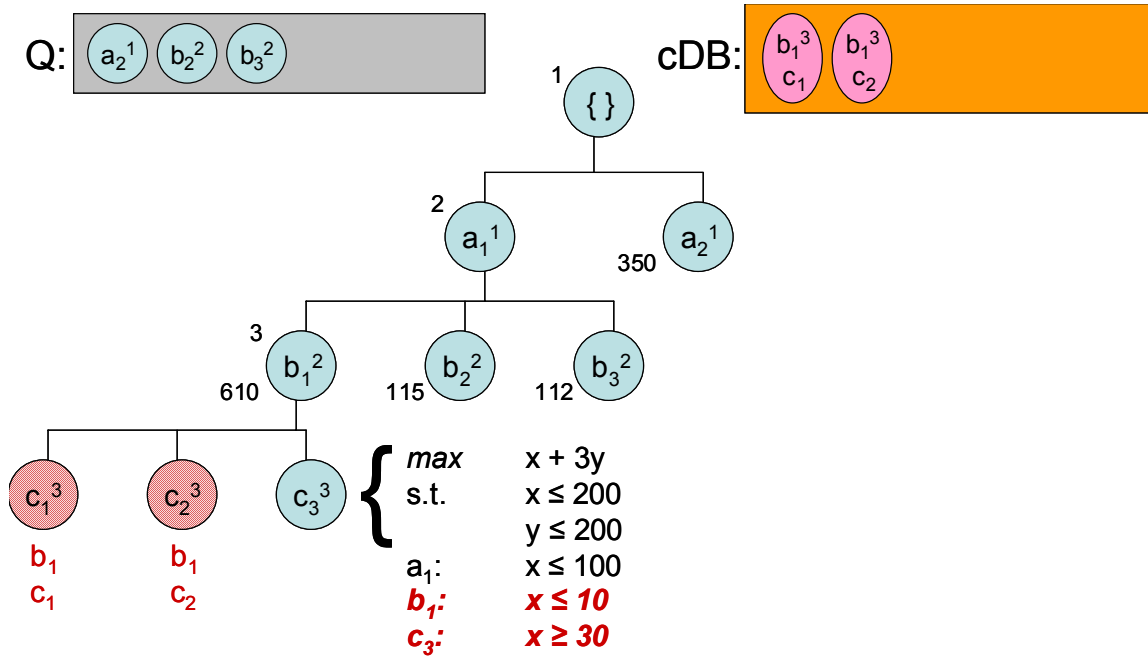


**Figure 21:** Discovery of infeasibility and subsequent conflict generation in CDCL-B&B algorithm.

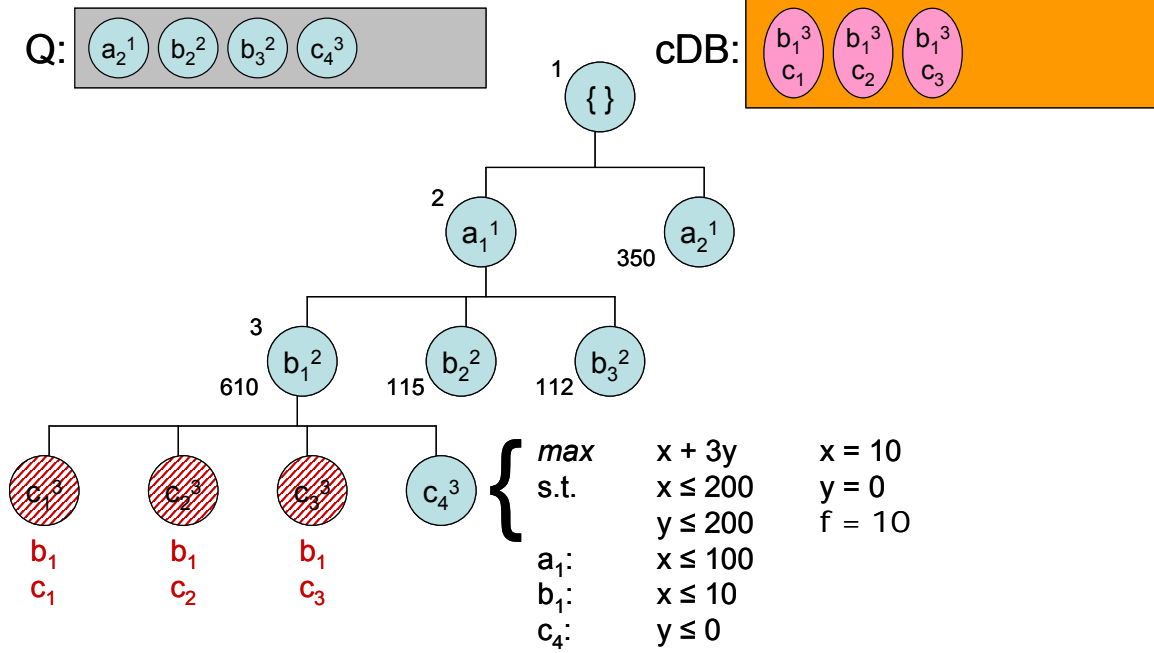
The rest of this expansion of the tree proceeds in a similar manner. The second conflict is added to the database, and a third child is added (Figure 22). Once again an infeasibility is discovered, the third child is also pruned, and a third conflict is generated.

The fourth child is not infeasible: the objective function has value 10 at this node (Figure 23). It has no active clauses, but is still added to the queue (inserted at the end) as is normally done. Recall that just like the CL-B&B algorithm, termination only occurs when a node that satisfies all clauses is *removed* from the queue, not when the node is added to the queue.

Sections 4.3.3 and 4.3.4 examine how discovered conflicts are retrieved and resolved. The remainder of the example problem is illustrated in these sections as well.



**Figure 22:** Another infeasibility discovered during execution, resulting in another conflict.



**Figure 23:** The third conflict is stored in the database, and the final child is discovered to be feasible. With an objective function value of 10, it is less optimal than all other nodes in the queue and is inserted at the end.

#### 4.3.3 Retrieving the Active Conflicts at a Node

As a prelude to resolving the conflicts at a particular node, the conflicts must be retrieved from cDB. The conflict database is global, so it contains conflicts discovered at different times and from different nodes; therefore, identifying which conflicts are relevant is non-trivial. In particular, the goal of a conflict retrieval policy is to quickly identify the *active* conflicts. In the CDCL-B&B algorithm, an active conflict at a particular node is simply a conflict that has not yet been explicitly resolved. CDCL-B&B uses the time stamps of conflicts and LP nodes to implement a retrieval policy that operates in constant time. The policy is: *expansion of an LP node  $n_p$  into its children requires the explicit resolution of any conflicts discovered after  $n_p$ 's creation, but before the time of the expansion.* If cDB is implemented as a hashtable, it can retrieve all such conflicts in a constant amount of time.

It can be proven through structural induction that this conflict retrieval policy guarantees that each LP node resolves all conflicts that were discovered prior to the time of its creation. The proof proceeds as follows. Let  $p(n)$  be defined to be true iff LP node  $n$  resolves all conflicts discovered prior to its time of creation.  $p(n_0)$ , where  $n_0$  is the root

node of the search tree, is true because there are no known conflicts at the start of execution. Assume a new node  $n_c$  is created as a child of  $n_p$ , and that  $p(n_p)$  is true. At this point all known conflicts can be divided into two categories: conflicts discovered after the creation of  $n_p$ , and conflicts discovered before.

Any conflicts discovered after the creation of  $n_p$  are explicitly resolved at  $n_c$  by the retrieval policy, because these conflicts were created after the parent's creation but before the time of tree expansion (the current time). Conflicts discovered before  $n_p$ 's creation are resolved by  $n_p$  because  $p(n_p)$  is true. Recall from Section 4.2.2 that resolving a conflict requires guaranteeing that the conflict will not appear; this is achieved by the inclusion of disjuncts. Also recall that descendent nodes inherit all the disjuncts of their ancestors (Section 3.1.2 used this property to show that descendants all have smaller feasible regions than their ancestors). Therefore,  $n_c$  also inherits the conflict-resolving disjuncts, and so also resolve all conflicts discovered before  $n_p$ 's creation. Since  $n_c$  resolves all conflicts,  $p(n_c)$  is true.

All nodes are descendents of the root node  $n_0$ . Therefore the base case of the inductive proof was established by proving  $p(n_0)$ , and the inductive step was proven by showing  $p(n_p)$  entails  $p(n_c)$  for an arbitrary parent  $n_p$  and child  $n_c$ . Hence, by induction,  $p(n)$  is true for all LP nodes. Thus the conflict retrieval policy guarantees that any LP node resolves all conflicts discovered prior to the time of its creation. Note that this policy only establishes how to *retrieve* conflicts from the database. Chapter 6 specifies how to extract them before recording them in the database, while Section 4.3.4 describes how to resolve them.

Conflict retrieval occurs in the first line of the pseudo code of `resolveConf` (Figure 24), the method that performs conflict resolution. To illustrate the conflict retrieval policy, consider again the running example used in this chapter. Throughout the BFS search mode, which has been described thus far, attempting conflict retrieval would return nothing, because no conflicts had been identified. After the expansion of the node that added  $b_1^2$ , however, three conflicts have been stored in `cDB`. Each of these have been stamped with the time 3.

The next node to expand in the BFS mode is  $a_2^1$ , stamped with time 1. The time of the expansion of  $a_2^1$ , which is the current algorithm time, is 4. Since all three conflicts

were created after  $a_2^1$  but before the time of expansion, all three must be resolved by CR mode before BFS mode can continue. However, any LP node children of  $a_2^1$  will have been created at time step 4. Therefore, when the time comes for their expansion, the three conflicts will not need to be resolved, since the conflicts were created earlier. In this way no node resolves a conflict that has already been resolved by an ancestor.

#### 4.3.4 Combined Conflict Resolution and Unit Propagation

When an LP node is removed from the head of the priority queue in order to be expanded, but there exists at least one conflict that has not been resolved by that node, the algorithm switches to conflict resolution mode. In CR mode, conflict resolution takes place instead of the normal BFS tree expansion; this begins by retrieving the active conflicts using the policy from 4.3.3. Recall that a conflict consists of a set of clause disjuncts that are together inconsistent. The key idea behind resolving such a conflict is to add the negation of one of these disjuncts to the expanded node. This specifies that any feasible states must lie outside of the conflict.

Inequalities contained in other clauses may hold as a logical consequence of this resolution. These consequences are identified quickly by performing propositional unit propagation on the LP clauses and the selected disjuncts. When this occurs, each disjunct is treated as a simple propositional variable that may be assigned true or false. This process is repeated for each active conflict before returning to BFS mode. In BFS mode, unit propagation is already implicitly performed. The only possible consequence of adding a disjunct is resolution of a clause, which is addressed by using activity as a criterion for selecting which clause to resolve.

##### 4.3.4.1. *High-level Conflict Resolution Strategy*

There are usually a number of negations of disjuncts that can be used to resolve any given conflict. In general, for a conflict  $\{\alpha_x, \beta_w, \dots\}$ , the possible negations that would resolve the conflict would be  $\neg\alpha_x$ ,  $\neg\beta_w$ , and so on for each disjunct  $\delta_i$  in the conflict. However, a given node may not have the full scope of choices for resolving the conflict. Generally, the possibilities for resolution are governed by four cases.



- *Case 1:* For some disjunct  $\delta_i$  in the conflict, the node includes some other disjunct of clause  $\delta$ , in which case the conflict is already resolved.
- *Case 2:* The node has already selected some subset of the conflict's disjuncts. This constrains the possible ways of resolving the conflict to only the clauses containing conflict disjuncts that have not been selected.
- *Case 3:* The node has not selected any disjunct in the conflict, which means that the entire range of choices for resolving the conflict is available.
- *Case 4:* All disjuncts of the conflict have already been included in the node, so the node explicitly contains an infeasibility and must be pruned.

Given a node to expand and a conflict to resolve, the algorithm uses the above case list to determine how to generate the list of possible negated disjuncts that achieve resolution. A new node is created for each possible negated disjunct, with each node augmented by its respective negated disjunct. The addition of a negated disjunct to a node may entail other disjuncts. These are identified using unit propagation, as mentioned above, and added to the node.

Once these additional disjuncts have been added, three cases are possible:

- *Case A:* If unit propagation indicates that there is no way to add any negated disjunct that resolves a known conflict without resulting in a propositional inconsistency, then the node contains an infeasibility and is pruned.
- *Case B:* If this is not the case and the node has resolved all known conflicts, the node is marked as an LP node and expanded using BFS.
- *Case C:* If neither case A nor case B holds, then other unresolved conflicts exist. The node is queued for resolution of another conflict.

#### 4.3.4.2. Nodes and Tree Branching in CR Mode

The CR nodes used for expansion in conflict resolution mode have a very different structure from the LP nodes used in BFS mode. CR nodes can be expanded into descendants without performing the computationally costly step of finding the solution to a relaxed LP.

Recall that BFS mode expanded nodes by resolving a clause at each tree level. Since only active clauses need to be resolved, and clauses are active based on the solution to the relaxed LP of the parent node, BDS mode must solve the parent's relaxed LP prior to each tree expansion. The relaxed LP of a node is in fact solved even earlier, because the cost of the solution is needed to establish the ordering of the node in the queue.

In CR mode, the Clausal LP formulation provides an abstraction barrier (see Section 2.2.3) that hides the LP details of each disjunct and enables the algorithm to avoid solving LPs when CR nodes are expanded. Each descendant explicitly resolves a conflict, and conflicts store disjuncts as abstracted symbolic terms rather than as particular inequalities; CR mode can therefore also view disjuncts as terms rather than LP inequalities. Instead of a new tree level depending on the inclusion of a new disjunct in the relaxed LP, each new tree level depends on the inclusion of a new symbolic term, independent of the inequality it represents. Additionally, without solving an LP for each CR node, the algorithm can process all CR nodes quickly, relative to BFS mode. Instead of a BFS order, CR mode instead uses a breadth-first search, and uses a FIFO queuing policy.

Adding negations to CR nodes also has the effect of reducing the number of disjunctive choices available for clause resolution. Sometimes clauses are completely resolved just because of the constraints that are entailed by the addition of the negations. This reduces the number of LP nodes that must be subsequently examined.

As a simple example, consider a node in BFS mode that has relaxed the clause  $a_1 \vee a_2$ . Additionally, the conflict  $(a_1, b_1)$  is active at this node. Two constituent kernels for this conflict would be  $\neg a_1$  or  $\neg b_1$ . If  $\neg a_1$  were added to the node in order to resolve the conflict, the new negated disjunct would rule out  $a_1$  as a way to resolve the relaxed clause. This would entail that  $a_2$  must be added to the node also. Hence the clause would be resolved merely by the addition of the negated disjunct. This entailment process is implemented using the unit propagation discussed in 4.3.4.4

As a prerequisite to determining what disjuncts are entailed, CR nodes need to contain the list of already-selected constraints and a list of unresolved clauses. As mentioned, the mathematical details of constraints and disjuncts are abstracted away: CR nodes only contain a symbolic representation of each disjunct in a clause. Instead of " $x >$

10  $\vee y < 50$ ”, for example, a CR node stores the abstracted representation “ $a_1 \vee a_2$ ” where  $a$  is the clause, and the subscripts refer to the disjunct. This formalizes the abstraction barrier described above.

CR nodes also track as-yet unresolved conflicts, but again abstract away the mathematical details. This is equivalent to recording the disjunction of the constituent kernels of a conflict in CD-A\*. For example, “ $\neg a_1 \vee \neg b_1$ ” would be the representation of conflict  $\{a_1, b_1\}$ . Note that CR nodes store conflicts in the same manner as the conflict database.

#### 4.3.4.3. Pseudo Code and Example Walkthrough for CR Mode

Recall that conflict resolution mode begins when a node scheduled for expansion has not yet resolved one or more known conflicts. CR mode therefore begins with an LP parent node and a list of conflicts that must be resolved. In the example used in this chapter, the LP parent node is  $a_2^1$  and the list of conflicts is the entire contents of **cDB**. Figure 24 shows the pseudo code for CR mode, in a method called **resolveConf**. This method accepts **lp\_parent**, the parent LP node scheduled for BFS expansion, and **conflicts**, the list of unresolved conflicts.

This mode ends when all the CR in the current tree meet case B; that is, when all unexpanded CR nodes resolve all conflicts. At this point the CR leaf nodes are marked as LP nodes and returned for BFS expansion. **resolveConf** returns this list of LP nodes on line 25, contained in the list **children**. The nodes in the returned list do *not* have their relaxed LPs solved; this is why line 19 of the **CDCL-B&B** method solves the nodes in the list **children** before inserting them into the BFS queue. 4.3.4.5 discusses the return to BFS mode in more detail.

During initialization, the constraints, clauses, and unresolved known conflicts are stored in a CR node. In line 1 of **resolveConf**, a root CR node called **cr\_parent** is created to hold this information from **lp\_parent** and **conflicts**. Figure 25 illustrates this initialization process. Recall that the expansion of  $b_1^2$  completed with the discovery of three conflicts and one feasible child.  $a_2^1$ , as the head of the queue **Q**, is scheduled next for BFS expansion. BFS expansion is suspended for  $a_2^1$  because of the existence of three unresolved conflicts in **cDB**. These conflicts are identified using the retrieval policy of

Section 4.3.3. Additionally, if any constituent kernel of a conflict is contained in `lp_parent.constraints`, then the conflict is implicitly resolved and is not stored in `cr_parent.conflicts`. In this example, none of the three conflicts are implicitly resolved by  $a_2^1$ . The list of constraints, unresolved clauses, and conflicts are listed to the right of the node, with the constraints shown in their negative disjunctive representation, as described earlier.

```

resolveConfs(lp_parent, conflicts)
1   cr_parent = <lp_parent.constraints, lp_parent.clauses_unresolved, conflicts>
2   cr_q = empty list
3   cr_q.add(cr_parent)
4   completedList = empty list
5   while (cr_q not empty) {
6       cr_curr = cr_q.remove( )
7       conf = cr_curr.conflicts.mostConstrained()
8       termsSoFar = empty list
9       for each <disjunct,  $\neg$ > in conf {
10          cr_child = copy of cr_curr
11          newTerms = empty list
12          add <disjunct,  $\neg$ > to cr_child.constraints
13          add <disjunct,  $\neg$ > to newTerms
14          remove conf from cr_child.conflicts
15          addSystematicityTerms(cr_child, termsSoFar, newTerms)
16          if (! propagate(cr_child, newTerms))
17              go to line 9
18          if (cr_child.conflicts is empty)
19              completedList.add(cr_child)
20          else
21              cr_q.add(cr_child)
22      }
23  }
24  children = empty list
25  for each cr_leaf in completedList
26      children.add(crToLp(cr_leaf, lp_parent))
27  return children

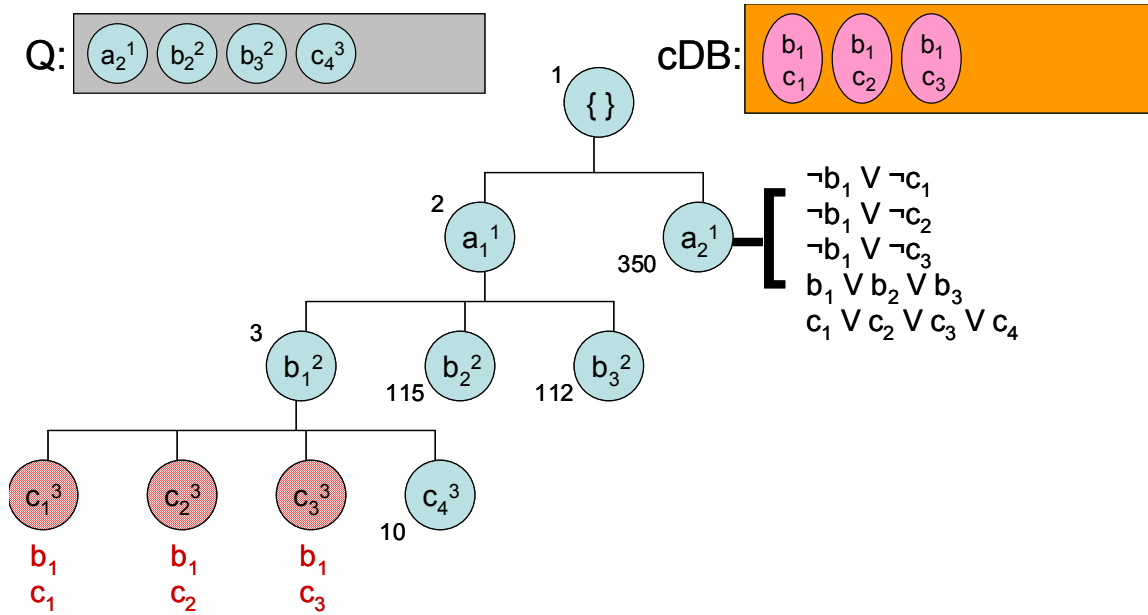
```

**Figure 24:** Sub-method for resolving conflicts

The main body of **resolveConf** is a loop that iterates until the queue `cr_q` becomes empty. `cr_q` contains the CR nodes that have not yet resolved all conflicts in `confList` and still need to be processed. In Figure 25, only `cr_parent` has been added to `cr_q` (specified by line 3 of **resolveConf**). `cr_q` uses the `.add()` and `.remove()` method and is structured as a list. Therefore CR nodes are added to the end and removed from the start, so expansion takes place in a breadth-first manner. As CR leaf nodes are discovered that resolve all conflicts, they are stored in `completedList`, initially empty.

When all nodes have been expanded or moved into **completedList**, the queue of pending CR nodes is empty (line 5). The post-processing to convert the CR nodes in **completedList** into LP nodes then occurs (lines 22-25).

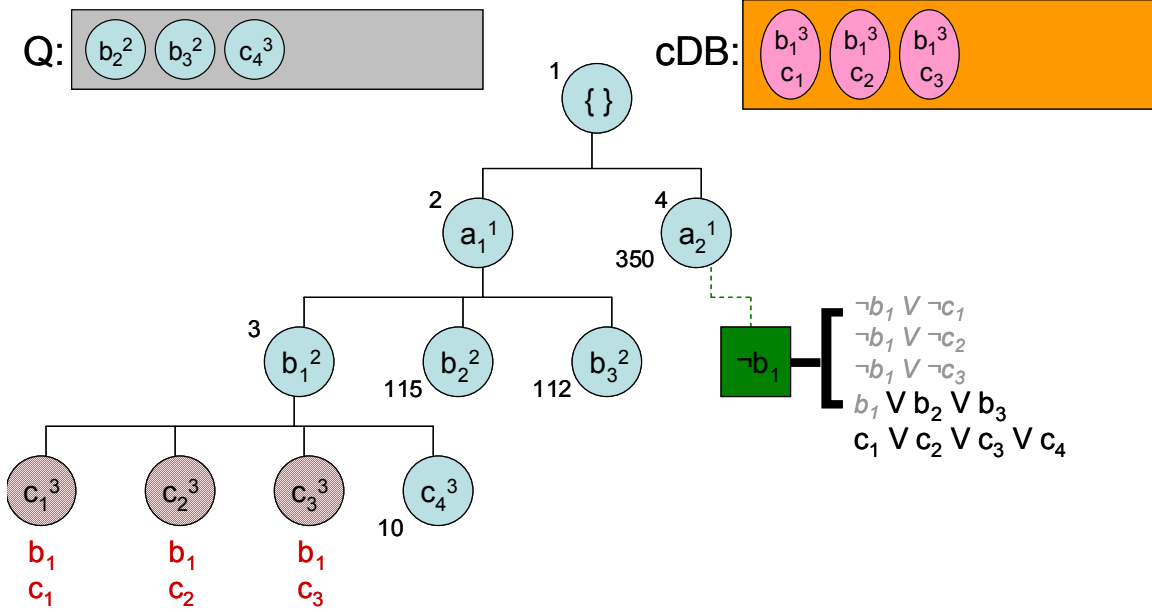
During the main loop, **cr\_curr** refers to the CR node currently being expanded, and **conf** refers to the next conflict that will be used for expanding **cr\_curr**. During each iteration of the loop, **cr\_curr** is removed from **cr\_q** (line 6). Since **cr\_q** only stores nodes that have one or more unresolved conflicts, **cr\_curr** is guaranteed to have at least one unresolved conflict. If there are multiple unresolved conflicts at **cr\_curr**, the conflict that is selected is the one such that the number of disjuncts in the clauses that are mentioned in the conflict is at a minimum. For example,  $\{b_1, c_1\}$  mentions two clauses, **b** and **c**. **b** has three disjuncts and **c** has four, so the number of disjuncts in mentioned clauses is seven. In fact all three unresolved conflicts are identical based on this metric, so the earliest discovered conflict,  $\{b_1, c_1\}$ , is picked (line 7).



**Figure 25:** Initial constraint and clause list at the start of conflict-mode expansion to resolve constraints.

During each iteration of the subloop (lines 9-20 in **resolveConf**), a child **cr\_child** is created to include a negated disjunct, **neg\_disjunct**, from **conf**. **cr\_child** is initially only distinguished from its parent **cr\_curr** through the addition of **neg\_disjunct** to its list of constraints and the removal of **conf** from its list of conflicts (10-12). For example,  $\{b_1, c_1\}$ , can be resolved using one of two constituent kernels:  $\neg b_1$  or  $\neg c_1$ . Figure 26

shows the first CR child (a square node), which adds  $\neg b_1$ . Figure 27 shows the addition of the second CR node into the tree, augmented with  $\neg c_1$ .

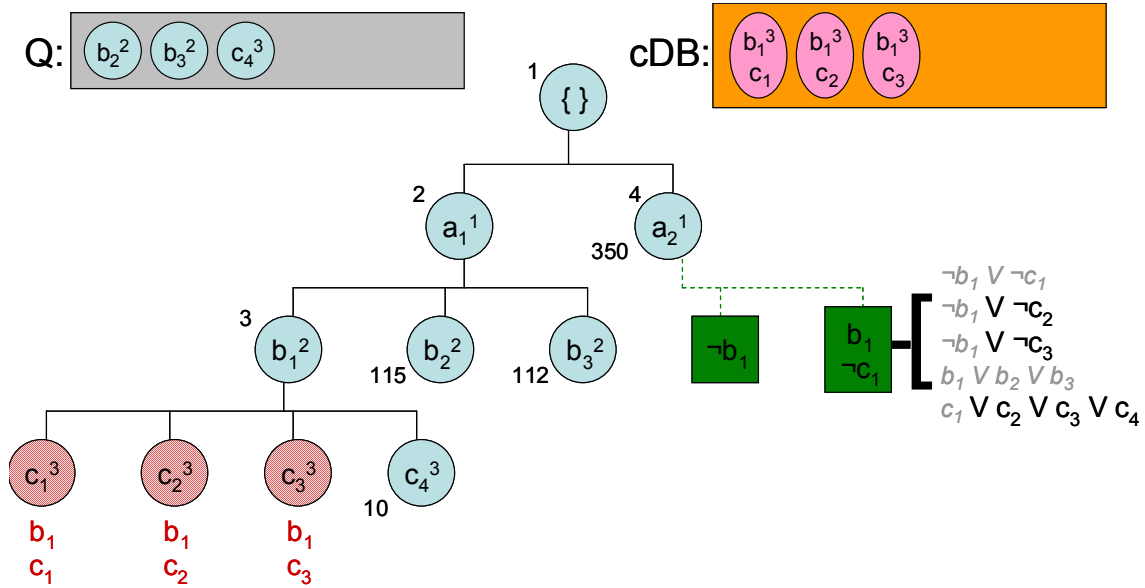


**Figure 26:** Creation of first conflict-mode node with a negation, followed by unit propagation.

“Younger” siblings (the 2<sup>nd</sup>, 3<sup>rd</sup>, etc. child of `cr_curr`) are also augmented with the negations of their older siblings’ disjuncts by the `addSystematicityTerms` helper method on line 13. These terms are added to ensure that siblings’ descendents never look alike; this is known as ensuring systematicity. As an example of this effect, consider Figure 27, where the second CR child of  $a_2^1$  is added to the tree with disjunct  $\neg c_1$ . Here `addSystematicityTerms` also adds the negation of its older sibling’s disjunct,  $\neg b_1$ . The negation of  $\neg b_1$  should logically be  $b_1$ , and so  $b_1$  is also added to the second sibling.

After the required disjuncts and/or negated disjuncts are added to each child, unit propagation is invoked to determine what effect the new constraints have on the unresolved clauses and conflicts. This process is discussed in more detail in 4.3.4.4. Recall from 4.3.4.1 that three possible cases can result after propagation completes. These three cases are dealt with in lines 14 to 19 of `resolveConf`. In case A, the addition of the new disjuncts resulted in a contradiction on the propositional level. The `propagate` method that does unit propagation on line 14 is defined to detect this and return `false` if it occurs; in this case, the algorithm returns to line 9 to consider the next

child, and no action further action is done to the child that was just created. Therefore it is dropped from the queue and pruned, as specified by case A. In case B, the CR child node resolves all conflicts, and no further CR expansion needs to be done to it. Thus `cr_child.conflicts` will be empty on line 16 and `cr_child` is moved to `completedList` for conversion into a BFS mode (line 17). When conversion occurs, all added disjuncts are transferred to an LP node and so are retained for BFS mode (see 4.3.4.5). If neither case A nor case B holds, then more conflicts remain for resolution, and the child is reinserted into the queue on line 19.



**Figure 27:** Creation of a second CR node, containing another negation and an additional constraint ( $b_1$ ) to satisfy the demands of systematic search. Unit propagation also occurs on this node.

#### 4.3.4.4. Unit Propagation to find the Constraints Entailed by a Negated Disjunct

Unit propagation occurs immediately after a new node is augmented with negated disjuncts. The goal of propagation is to quickly discover what consequences the new negated disjuncts have on the clauses and conflicts that must be resolved. Propagation also adds any additional terms that are entailed by the current list of clauses and constraints. A *term* in this context is a single (positive) disjunct of a clause, or the negation of a disjunct of a clause, in a purely symbolic form. In other words, the LP elements are abstracted away, as is the case throughout CR mode. However, recall from 4.3.4.2 that propagation can reduce the number of possible disjuncts that can be selected

for a particular clause, and therefore reduce the tree branching factor. Repeated propagation can also lead to full resolution of a clause, reducing tree depth.

At the highest level, three principles govern the unit propagation in the CDCL-B&B algorithm. First, adding a negated disjunct to a CR node may result in infeasibility, clause *contraction* [*can't think of a good word to use here*] (the removal of a disjunct from the clause), or conflict resolution. Second, adding a positive disjunct to a CR node may result in infeasibility, clause resolution, or conflict contraction. Third, clause or conflict contraction may result in infeasibility or addition of a term (or may have no effect). Whenever a term is added to a CR node, principle one or two is applied. If this results in the contraction of a clause or conflict, principle three is applied. If this results in the addition of a term, principle one or two is applied again, and so on. This process terminates when an infeasibility is detected or when one round of applying all three principles results in no changes to the list of terms or the list of constraints and conflicts.

To implement the first principle, unit propagation checks all clauses, conflicts, and positive disjuncts in a CR node if a negated disjunct is added to the node. At this time, one or more of the following cases might apply.

- *Case 1:* If the disjunct portion of the negated disjunct is already included as a constraint at this node, then a contradiction has been detected. The node is immediately pruned, and no other case is considered.
- *Case 2:* If the disjunct portion of the negated disjunct is contained in a clause, then that clause cannot be resolved by the selection of this particular disjunct. Hence the clause is contracted: the disjunct is removed from the clause. This clause contraction only applies at this node and for all its descendants, which inherit all terms.
- *Case 3:* If the negated disjunct is one of the constituent kernels of a conflict, then adding it resolves the conflict, which is removed from the list of unresolved conflicts.
- *Case 4:* The disjunct does not appear anywhere, so the first principle has no effect.



To implement the second principle, unit propagation checks all clauses, conflicts, and negated disjuncts in a CR node if a positive disjunct is added to the node. At this time, one or more of the following cases might apply.

- *Case 1:* If the disjunct portion of any negated disjunct in the list matches the newly added positive disjunct, then a contradiction has been detected. The node is immediately pruned, and no other case is considered.
- *Case 2:* If the positive disjunct is contained in a clause, then that clause is resolved. It is therefore removed from the list of unresolved clauses.
- *Case 3:* If the negation of the added disjunct is a constituent kernels of a conflict, the conflict cannot be resolved with that disjunct. Therefore the conflict is contracted by removing that disjunct (at this node and all its descendents).
- *Case 4:* The disjunct does not appear anywhere, so the identical situation arises as in Case 4 of the first principle.

To implement the third principle, any clauses or conflicts reduced due to the first or second principle are checked. If any of them has zero disjuncts remaining, then it cannot be satisfied. Hence a contradiction has arisen and the node is pruned. If instead a clause or conflict has exactly one disjunct or constituent kernel remaining, then there is only one resolution possible. The positive disjunct or constituent kernel is added to the list, and one of the first two principles is re-applied. If the first two principles are applied and no clauses or conflicts have been contracted, then unit propagation terminates.

Figure 26 and Figure 27 both illustrate unit propagation in the context of the example used throughout this chapter. Note that both CR nodes have beside them their respective constraint and clause list. These lists are identical to each other and to the list initially constructed before conflict resolution took place, shown in Figure 25, because the clauses and constraints are inherited by the descendant nodes. However, a different set of terms are faded, to reflect the differing effect of propagation on each node.

The first CR node expanded from LP node  $a_2^1$  is augmented by a single negation,  $\neg b_1$ . This negation is discovered to appear in three conflicts ( $\neg b_1 \vee \neg c_1$ ,  $\neg b_1 \vee \neg c_2$ , and  $\neg b_1 \vee \neg c_3$ ) and one clause ( $b_1 \vee b_2 \vee b_3$ ). Since each of the three conflicts

contain the actual negation, all three are resolved immediately (therefore all three conflicts are faded in Figure 26).

```

propagate(cr, newTerms)
1   contracted = empty list
2   for each <disjunct,  $\neg$ > in newTerms {
3       if (disjunct is in cr.constraints)
4           return false
5       if (any clause containing disjunct is in cr.clauses_unresolved) {
6           remove disjunct from clause
7           contracted.add(clause)
8       }
9       if (any conflict containing <disjunct,  $\neg$ > is in cr.conflicts)
10          remove conflict from cr.conflicts
11   }
12   for each disjunct in newTerms {
13       if (<disjunct,  $\neg$ > is in cr.constraints)
14           return false
15       if (any clause containing disjunct is in cr.clauses_unresolved)
16           remove clause from cr.clauses_unresolved
17       if (any conflict containing <disjunct,  $\neg$ > is in cr.conflicts) {
18           remove <disjunct,  $\neg$ > from conflict
19           contracted.add(conflict)
20       }
21   }
22   newTerms = empty list
23   for each clause/conflict c in contracted
24       if (c.length = 0)
25           return false
26       else if (c.length = 1) {
27           add c.first( ) to cr.constraints
28           add c.first( ) to newTerms
29       }
30   if (newTerms is empty)
31       return true;
32   else
33       return propagate(cr, newTerms)

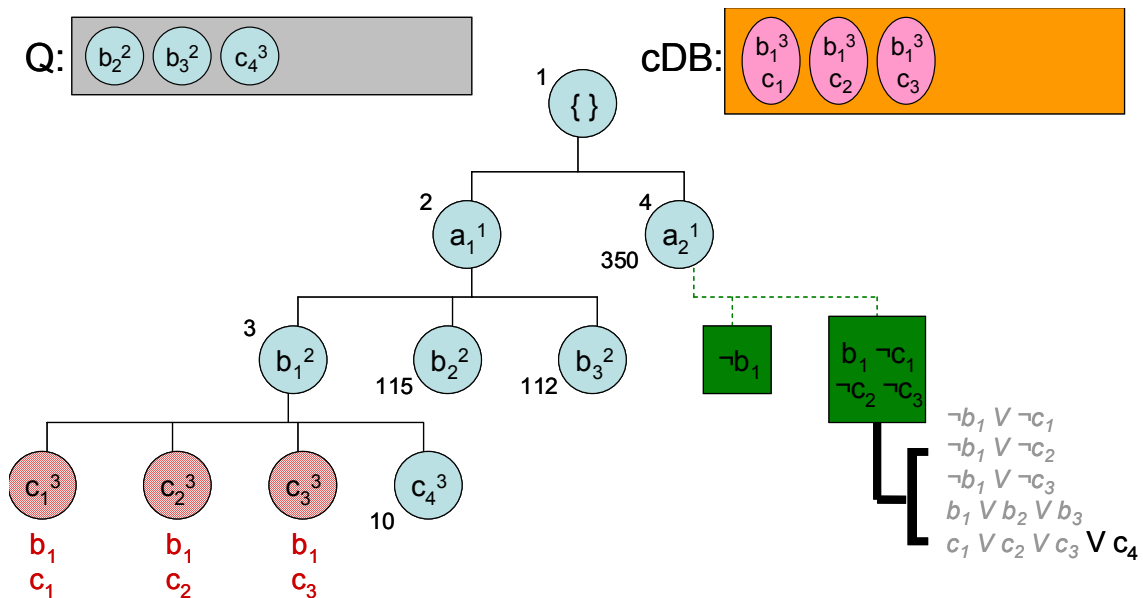
```

**Figure 28:** Pseudo code for method that performs propositional unit propagation.

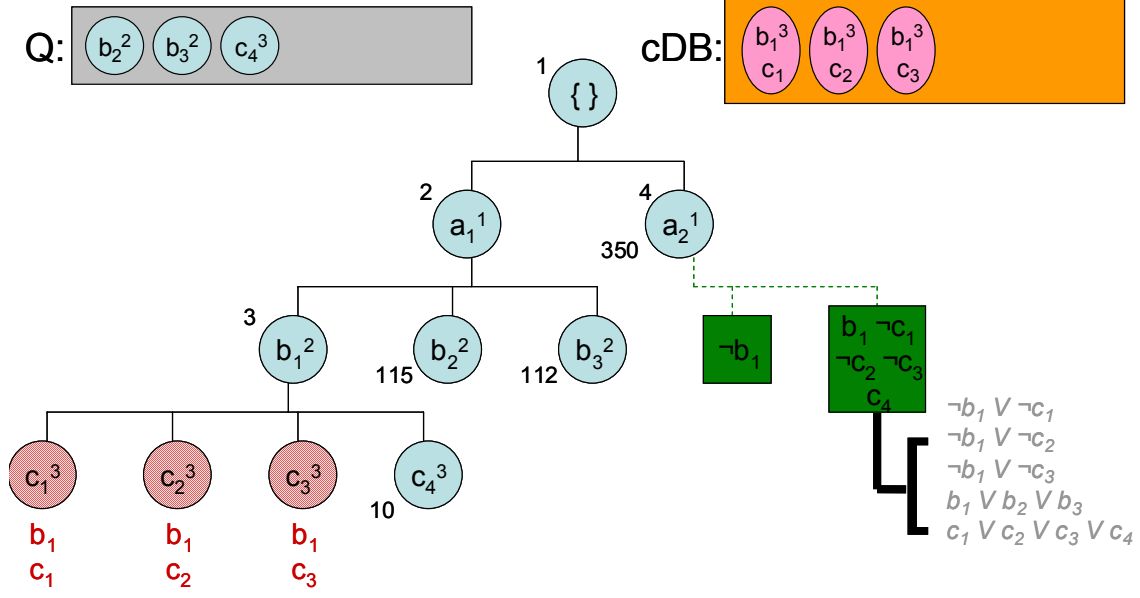
The clause has one of its disjuncts removed (the single disjunct  $b_1$  is therefore faded). The next step in the propagation process is to check if any clause or unresolved conflict has one or zero terms. There are no unresolved conflicts, and the two clauses contain two and four terms, respectively. Propagation terminates at this point. Since there are no conflicts unresolved, this node is scheduled for return to normal mode expansion.

The second node is augmented by both  $\neg c_1$  and  $b_1$ .  $\neg c_1$  resolves the  $\{b_1, c_1\}$  conflict and removes one term from clause  $c$ .  $b_1$  resolves clause  $b$ , but removes one term from each of the unresolved conflicts. This is the condition that Figure 27 shows. Checking the clauses and conflicts shows that the two unresolved conflicts are both reduced to a single term, which means that propagation continues. Figure 29 shows the result of the next round of unit propagation. Each of the conflicts that were reduced to a single term contribute that term to the constraint list:  $\neg c_2$  and  $\neg c_3$  are added. Propagating on these terms further reduce the number of disjuncts in clause  $c$ , so that only  $c_4$  remains. This means that clause  $c$  is down to a single term, so that term must be added and propagated. Figure 30 shows the result of this final propagation:  $c_4$  is added and clause  $c$  is resolved, leaving no unresolved conflicts or constraints. *[reference rob ragno's thesis]*

The **propagate** helper method (Figure 28) is invoked in line 14 of **resolveConf** and returns false if any contradictions are discovered. Recall that in this case **cr\_child** is an infeasible node and is pruned, as mentioned previously (case A of 4.3.4.1). Execution then moves on to the next child of **cr\_curr** (line 15). If **propagate** instead returns true, indicating no contradictions, **cr\_child** is either put into **completedList** (case B, on line 17) if it has resolved all conflicts or put back into **cr\_q** if it has not (case C, line 19). In the example in this chapter, both nodes are put into **completedList** for conversion back into LP nodes. This conversion is discussed next in 4.3.4.5.



**Figure 29:** Continued unit propagation results in the inclusion of more negations, which further impact the remaining unresolved clause.



**Figure 30:** Final propagation for the CR node.

#### 4.3.4.5. Conversion of CR Nodes and Return to BFS Mode

Leaf nodes in conflict-mode expansion eventually achieve a state where they have resolved all known conflicts. At this point they are scheduled for return into the first expansion mode, by being stored in `completedList`. The final step of `resolveConf` is the post-loop processing, where the completed CR nodes are converted back into LP nodes and returned to the main CDCL-B&B method (see lines 22-25 of `resolveConf` and Figure 31). Both the CR nodes in the example have resolved all three known conflicts with a single conflict-mode expansion. Therefore both are ready for conversion back into LP nodes.

```

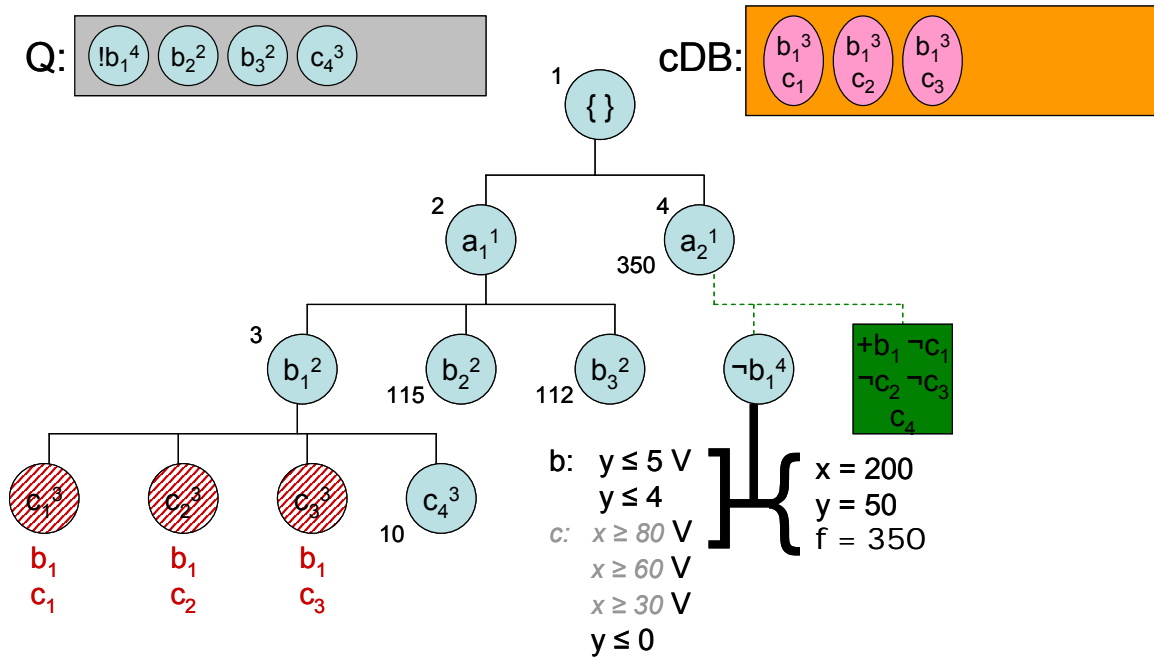
crToLp(cr, parent)
1   lp_unsolved = <parent.f, cr.constraints, cr.clauses>
2   return lp_unsolved

```

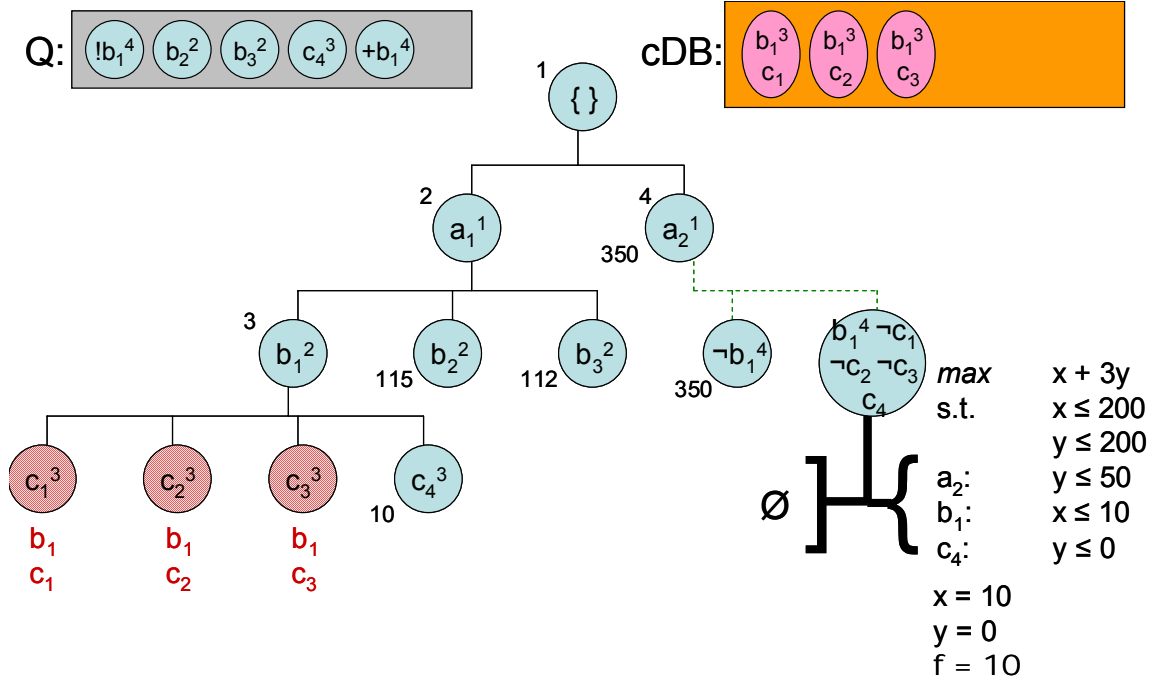
**Figure 31:** Helper method for converting from CR nodes to LP nodes.

Figure 32 shows the conversion of the first child, and Figure 33 shows the conversion of the second. Note that since the first child was only modified from its parent,  $a_2^1$ , with the addition of a negation, which will not affect the LP, the solution value is the same as  $a_2^1$ . Also note that the new LP node is stamped with the time of its

creation, which is still time step 4, since conflict-mode expansions do not count as extra time steps. The node's unresolved clauses reflect the state of the clause and constraint list at the end of propagation (Figure 26): clause b, which had one of its terms removed during propagation, only has two disjuncts here. The second child, which did have new non-negated constraints added to it, has a different solution than its parent. It also has no unresolved clauses, since propagation eliminated them all. It too is marked as being created at time step 4. Both nodes are inserted into the queue Q at the appropriate location (the first child at the head of the queue and the second child at the end) based on their new solution values (350 and 10, respectively).



**Figure 32:** Conversion of first CR node back into an LP node.



**Figure 33:** Conversion of second CR node back into an LP node.

The remainder of this example problem is completed by using only non-CR search mode expansion. Figure 34 shows the expansion of the node into two more children nodes, one of which is inserted at the head of the queue. This node,  $b_2^5$ , has resolved all clauses, and so when it is removed the algorithm terminates with  $x = 200$ ,  $y = 5$ , and the final solution value (returned by the CDCL-B&B method) equal to 215 (Figure 35).

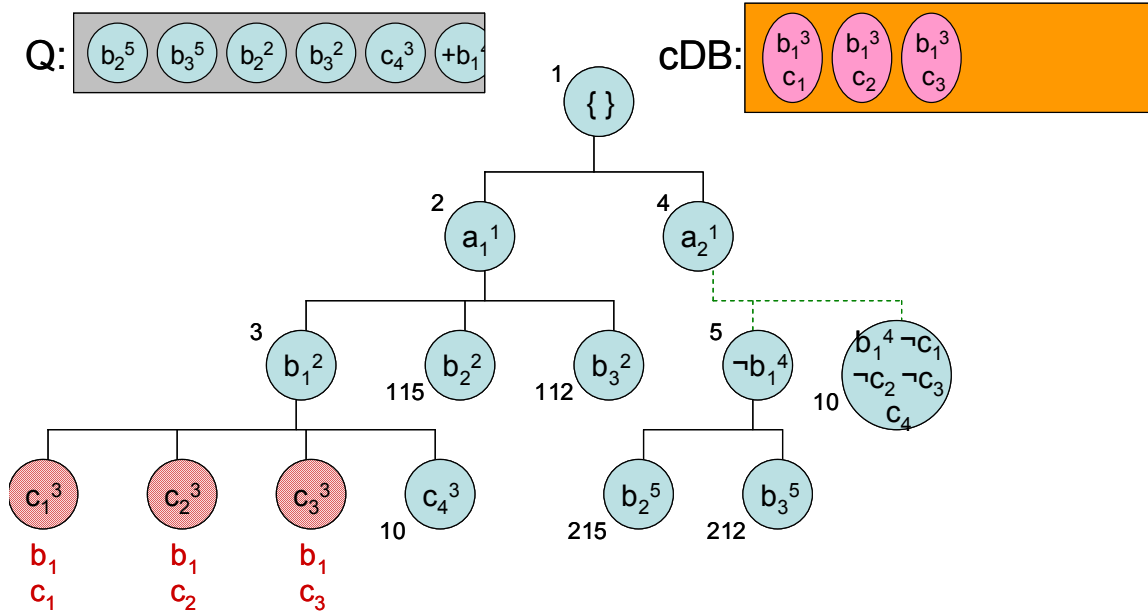
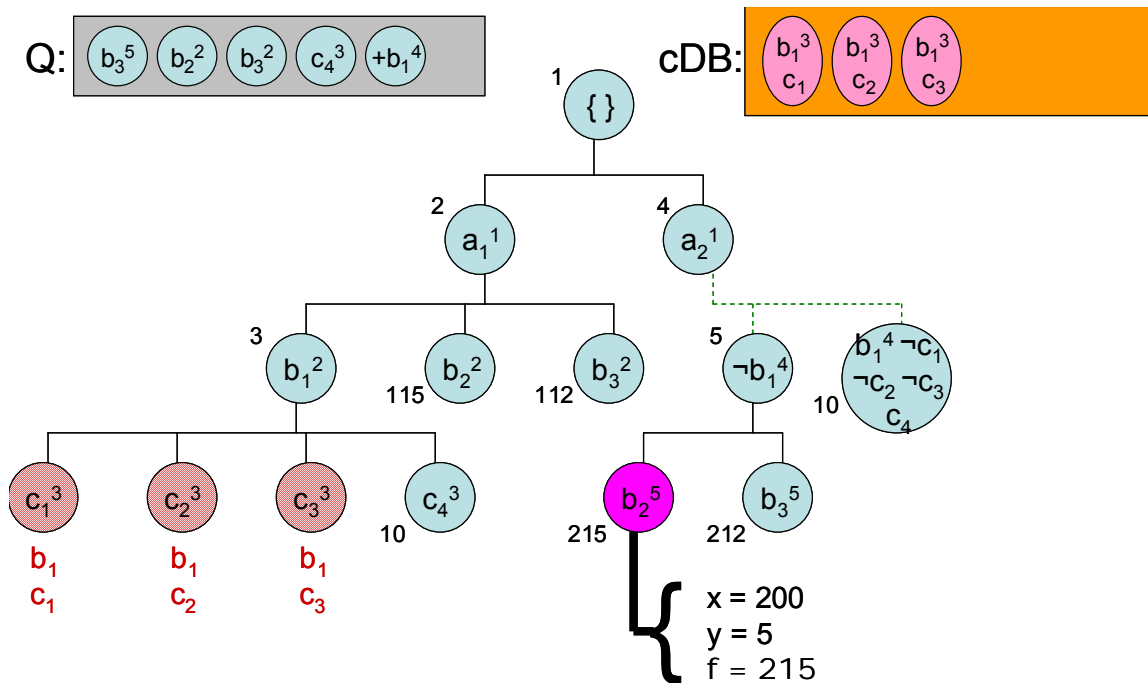


Figure 34: End of LP expansion at time step 5.



#### 4.3.5 Algorithm Summary



## 5 Clausal LPs applied to the Cooperative Path Planning Domain

This chapter applies the Clausal LP framework to a particular domain of problems: cooperative path planning (CPP). In general, the goal of a CPP is to guide multiple vehicles to cooperatively achieve a set of goals at different locations while avoiding certain regions (obstacles, no-fly zones, etc.). One example is control of multiple planetary rovers to explore a list of sites of scientific interest while avoiding craters. The CPP problem provides a good domain in which to compare the CDCL-B&B algorithm from Chapter 4 with the algorithm that will be used as an efficiency benchmark, BIP-B&B. The BIP and Clausal LP formulations of a CPP problem are discussed in Sections 5.1 and 5.2, respectively.

Both the BIP and Clausal LP frameworks can represent key elements of a simple CPP problem; their only difference lies in the way they encode obstacles. Additionally, CPP problems have clear criteria upon which to evaluate problem complexity: as the numbers of obstacles, vehicles, and waypoints increase, so does the difficulty of solving the problem. Another benefit of using the CPP problem as a domain for evaluation is that there is a clear concept of what a conflict is, so the full conflict-directed element of CDCL-B&B can be evaluated.

### 5.1 CPP Problems Modeled as BIP Problems

A CPP problem aims to find a set of control vectors for  $v$  vehicles that maneuver them to reach a set of goal locations, while avoiding  $z$  zones where the vehicles are not allowed. In addition, the control vectors that are discovered must minimize the value of the objective function  $f(s)$ . It may be possible to model a complete CPP with multiple intermediary waypoints and final goal points as a BIP or Clausal LP; however, this complete modeling is beyond the scope of this thesis. Instead only CPP problems limited to a single final goal state will be modeled. Note that an executive planner can break up a multi-waypoint problem into separate single-waypoint problems in order to develop a solution to the overall problem. Constraints between different goal points, such as timing bounds, can also be resolved outside of the single-waypoint model. This method does not

guarantee overall optimality of the solution, however. Chapter 7 discusses this idea further.

[*John How paper reference*] presents a method for modeling problems in the limited CPP domain as BIPs. The initial encoding to be examined is for a single vehicle; multi-vehicle encodings simply duplicate the constraints for all vehicles and add some additional collision-avoidance constraints. The variables that must be assigned values in this encoding are the state vectors at each instant in time and the control or force vectors at each instant in time. The BIP objective function encodes three solution goals. First, each vehicle absolutely must achieve the final goal state; solutions that do not achieve this requirement should be significantly penalized (assigned a much lower utility). Second, the solution should minimize fuel usage. Finally, this BIP encoding also prioritizes solutions wherein each vehicle reaches its goal state faster. An objective function that encodes these requirements is:

$$\min f(s) = \sum_t [q' |s_i - s_f| + r' |u_i|] + p' |s_T - s_f| \quad \{5.1-1\}$$

where  $s_i$  is the state at time  $i$ ,  $s_f$  is the goal state,  $s_T$  is the state at time  $T$ , or the final state, and  $u_i$  is the force vector at time  $i$ .  $q'$ ,  $p'$ , and  $r'$  are different weights. The  $q' |s_i - s_f|$  term minimizes the average distance from the goal across all states,  $r' |u_i|$  minimizes fuel usage across all states, and  $p' |s_T - s_f|$  minimizes the distance between the final state and the goal state. If  $p'$  is given a much higher value than  $q'$  and  $r'$ , any solution that does not achieve the goal state will be severely penalized, which means every solution will reach the goal.

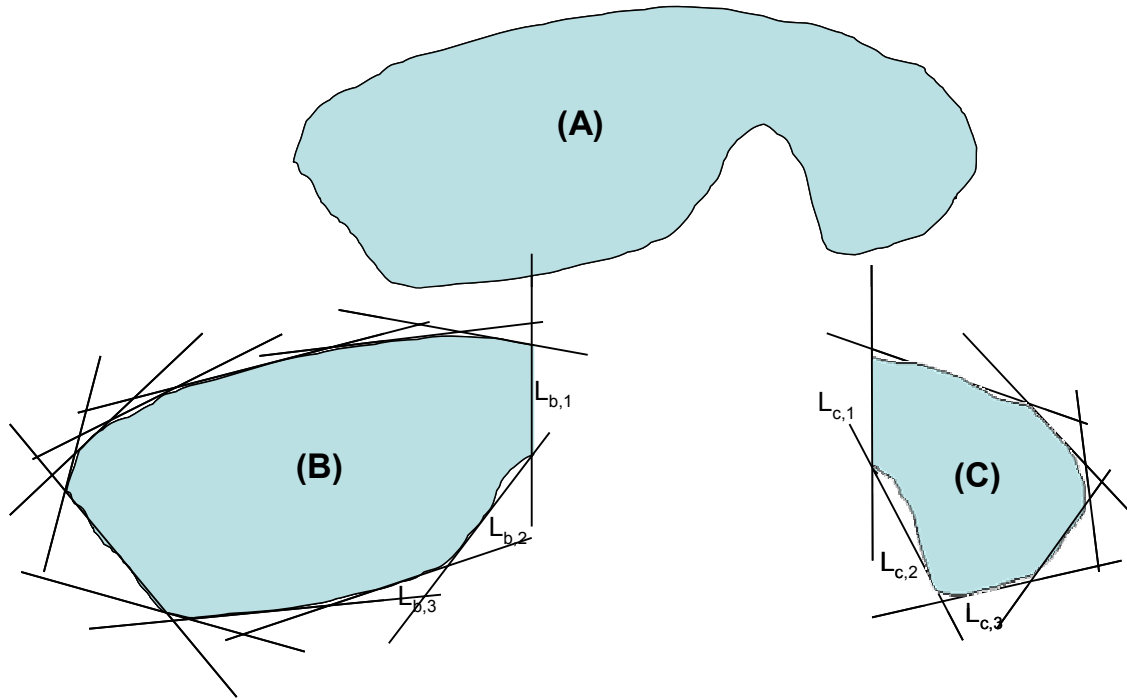
The first set of constraints in the BIP encodes the dynamics of each vehicle; in the simplest case, this requires constraints that indicate how the state vector changes due to the force vector. These changes will follow standard physics rules; therefore examples of such constraint for a single vehicle include:

$$v_{i,x} = v_{i-1,x} + u_{i-1,x} \quad \{5.1-2\}$$

$$p_{i,x} = p_{i-1,x} + v_{i-1,x} \quad \{5.1-3\}$$

where  $v_{i,x}$  indicates the velocity at time  $i$  in the  $x$  direction,  $p_{i,x}$  indicates the  $x$ -position at time  $i$ , and vectors  $v$  and  $p$  together comprise the  $s$  vector used in {5.1-1}. These equations would then be repeated for all vehicles, at all times, in all dimensions of the problem.

Finally, the obstacle avoidance constraints must be encoded. Avoidance of a convex obstacle can be viewed as a selection of a particular side on which to pass the obstacle. For instance, avoiding a square obstacle could be accomplished by passing it on the north, south, east, or west. Thus avoiding an obstacle involves the *decision* to select a particular *choice* of direction; this language implies that obstacles can be naturally represented as logical decisions in an HDCP. To create this representation for an arbitrary convex obstacle, the outside boundaries of the obstacle are linearized and an inequality constraint is introduced to describe each linear segment. The clause that consists of a disjunction of each inequality constraint therefore defines a boundary that is equivalent to the obstacle.



**Figure 36:** Illustration of the conversion of a non-convex obstacle, (A), into two convex obstacles, (B) and (C), which are then linearized.

This idea was applied to simple square objects in Figure 14 in Chapter 4. For an example of this process for an arbitrary object, consider Figure 36. Here a non-convex obstacle, (A), is first split into two convex obstacles, (B) and (C). The boundaries of each of these obstacles are then divided into segments which can each be linearized and represented as linear inequalities ( $L_{b,1}$ ,  $L_{c,1}$ , etc.). Examples of some of the resultant inequalities might include

$$L_{b,1}: -100 \cdot x + y + p_1 \geq 0 \quad \{5.1-4\}$$

$$L_{b,2}: -2 \cdot x + y + p_2 \leq 0 \quad \{5.1-5\}$$

$$L_{c,1}: -100 \cdot x + y + p_1 \leq 0 \quad \{5.1-6\}$$

$$L_{c,2}: 3 \cdot x + y + p_3 \geq 0 \quad \{5.1-7\}$$

and so on, where  $p_1$ ,  $p_2$ , and  $p_3$  are various points on the map. This process of linearization enables any level of detail: to more accurately represent the obstacle boundary, break the boundary up into smaller segments. Note that no claim is made regarding the efficiency of the conversion to this representation, only that obstacles in a CPP *can* be accurately represented in this way.

The requirement that one boundary segment must be selected for each obstacle can be encoded in a BIP program in the form of equations {2.1-3} and {2.1-4} from Chapter 2. Thus equations {5.1-4} through {5.1-7} would become

$$L_{b,1}: -100 \cdot x + y + p_1 \geq -R(1 - b_{b,1}) \quad \{5.1-8\}$$

$$L_{b,2}: -2 \cdot x + y + p_2 \leq R(1 - b_{b,2}) \quad \{5.1-9\}$$

$$L_{c,1}: -100 \cdot x + y + p_1 \leq R(1 - b_{c,1}) \quad \{5.1-10\}$$

$$L_{c,2}: 3 \cdot x + y + p_3 \geq -R(1 - b_{c,2}) \quad \{5.1-11\}$$

and so on, where each of the  $b_{\alpha,i}$ 's are binary and  $R$  is a very large number. To make sure that at least one segment is selected for each obstacle the constraints

$$\sum_i b_{b,i} \geq 1 \quad \{5.1-12\}$$

$$\sum_i b_{c,i} \geq 1 \quad \{5.1-13\}$$

and so on, are added to the BIP as well.

## 5.2 CPP Problems Modeled as Clausal LP Problems

Many of the constraints used to encode a Cooperative Path Planning problem as a BIP are also used in the Clausal LP representation of CPPs. {5.1-1}, {5.1-2} and {5.1-3} do involve any logical choices, and so are used exactly as they are. Thus the objective function and state transition constraints are identical between the BIP and CLP versions of a CPP. The CLP representation also relies on the linearization of convex obstacles as described in 5.1. However, instead of using a series of  $n+1$  constraints for each obstacle that is linearized into  $n$  inequalities, a CLP uses a single  $n$ -disjunct clause to model the selection. The form of this clause is like that of {2.2-1} from Chapter 2. Hence instead of constraints {5.1-8} through {5.1-13}, the clauses

$$L_{b,1} \vee L_{b,2} \vee \dots \quad \{5.1-14\}$$

$$L_{c,1} \vee L_{c,2} \vee \dots \quad \{5.1-15\}$$

would be added to the Clausal LP. These avoidance constraints must be repeated for every combination of obstacle, vehicle, and time instant, because every vehicle must avoid every obstacle all the time. Thus a  $v$ -vehicle,  $z$ -obstacle,  $t$ -timestamp CPP problem would result in a CLP with  $v \cdot z \cdot t$  total clauses.

## 6 Performance Analysis

This chapter provides an analysis of performance of the Conflict-directed Clausal LP Branch and Bound algorithm (CDCL-B&B). This algorithm, introduced in Chapter 4, solves hybrid decision-control problems represented as Clausal LPs. The Cooperative Path Planning (CPP) domain described in Chapter 5 is used to provide the evaluation problems. The Binary Integer Programming Branch and Bound (BIP-B&B) algorithm, described in Chapter 3, is used as a benchmark against which to compare the performance of CDCL-B&B.

Section 6.1 explains the experimental methodology used to compare the algorithms. The two metrics used for comparing the time efficiency of the methods are the number of LP nodes expanded during the search process and the average size of the tableaux that are solved at each LP. These provide processor-independent measures of computational time. The maximum size of the queue generated during each method is used to compare the space efficiency of the algorithms. This section also describes the types of problems generated to perform the comparison. Section 6.2 then presents the performance results of the two algorithms and analyzes these numbers.

### 6.1 Experimental Methodology

Comparing BIP-B&B and CDCL-B&B can be accomplished on the basis of time efficiency and space efficiency. Both of these metrics are affected by both the efficiency of the algorithms themselves as well as the compactness of the Clausal LP encoding as compared with the BIP encoding. More specifically, the time efficiency of the methods is compared using:

1. The total number of LP problems that are solved before reaching an optimal solution to the problem.
2. The average size of all LP problems solved.

and the space efficiency is compared using the maximum queue size. These two criteria provide significant indication as to how efficient an LP-based search algorithm is. The bulk of computational effort in both algorithms takes place in solving repeated LP problems. The total number and size of these problems defines the total computational

effort involved in solving the decision problem. Comparing the optimality of the final solutions generated by each method would provide no information, because they both terminate only when one of the optimal solutions is discovered. Using the amount of real-world time elapsed until termination as a metric would likely distinguish the methods, but this criterion is easily influenced by processor speed, number of background processes, efficiency of the LP solver used, and other factors of secondary importance. In particular, processor speeds are a continuously moving target, with typical upgrades occurring once every eighteen months.

In our metrics for assessing CDCL-B&B, the total number of LP problems solved does not include the cost of conflict extraction. Although different extraction procedures were examined during the course of this research project, optimizing the extraction process is outside the scope of this thesis.

The average LP size affects the time for solving each relaxed problem insofar as computational time increases with the number of constraints in the LP (the rate of increase is dependent on the LP solver used). The average LP size is therefore identified by looking at the size of the tableaux that must be solved. The size of a tableau is defined by its width, the number of variables in the LP, and its height, the number of constraints in the LP. The number of variables in an encoding of a particular HDCCP is different depending on whether the encoding is a Clausal LP or BIP. However, the number of variables is the same for all search tree nodes for a particular problem. The value used to determine the average tableau size in this section is the average number of constraints across all LPs solved. Because the nodes shallower in the tree will have LPs with fewer constraints, and an efficient algorithm will try to focus on expanding shallow tree nodes, the average number of constraints is a valid efficiency criterion.

To empirically evaluate performance, the CDCL-B&B and BIP-B&B methods were coded in Java. Both implemented systems used a java Operations Research library created by [reference] to solve any LP problems. The only types of problems handed to this library are linear programming problems with all variables real-valued; any binary variables in the BIP B&B program are either explicitly assigned a value or relaxed. A random CPP problem generator is used to create path planning problems with varying numbers of variables and decisions (clauses). For a given number of variables and

clauses, a set of ten problems is created with vehicle starting points, obstacle locations, and goal points randomly generated. Problems that are discovered to have no feasible solution are not included in the data.

Although the LP problems solved during conflict extraction in CDCL-B&B are not included in the results, the policy used to perform conflict extraction is still relevant because the effectiveness of this policy in identifying minimal conflicts impacts the overall performance of CDCL-B&B. Recall from Section 4.2.1 that for a given infeasibility, multiple conflicts can generally be identified. Smaller conflicts result in fewer CR mode branches and fewer LP nodes to explore. Discovering the conflict with the *smallest size*, however, is exponential in the number of constraints in the worst case. The extraction policy used during the analysis process was Incremental Linear Addition & Removal (ILAR). Section **Error! Reference source not found.** describes ILAR in more detail and briefly examines alternative extraction policies.

## 6.2 Results and Analysis

The results of the analysis described above is:

NUMBER OF RELAXED LPS SOLVED													
Variables:		18			36			60			90		
		BIP B&B	CDCL-B&B	%	BIP B&B	CDCL-B&B	%	BIP B&B	CDCL-B&B	%	BIP B&B	CDCL-B&B	%
Clauses:	0	1	1	0%	1	1	0%	1	1	0%	1	1	0%
	20	2	2	0%	5.1	3.2	-37%	13.3	4.8	-64%	17.4	6.1	-65%
	48				23.1	5.4	-77%	34.9	9.3	-73%	45.9	15.5	-66%
	80				41.5	9.2	-78%	118.7	16	-87%	163.2	24.7	-85%

RELAXED LP SIZE (NUMBER OF CONSTRAINTS PER PROBLEM)													
Variables:		18			36			60			90		
		BIP B&B	CDCL-B&B	%	BIP B&B	CDCL-B&B	%	BIP B&B	CDCL-B&B	%	BIP B&B	CDCL-B&B	%
Clauses:	0	33	33	0%	56	56	0%	114	114	0%	156	156	0%
	20	54	38	-30%	81.5	54.1	-34%	132	114.9	-13%	170.5	158.4	-7%
	48				131	64.3	-51%	160.7	118.4	-26%	200.8	158.2	-21%
	80				188.5	74.4	-61%	216.5	117.2	-46%	240.1	159.3	-34%

MAXIMUM QUEUE SIZE													
Variables:		18			36			60			90		
		BIP B&B	CDCL-B&B	%	BIP B&B	CDCL-B&B	%	BIP B&B	CDCL-B&B	%	BIP B&B	CDCL-B&B	%
Clauses:	0	1	1	0%	1	1	0%	1	1	0%	1	1	0%
	20	2	2	0%	2.4	2.6	8%	2.8	2.4	-14%	3.4	3.8	12%
	48				4.2	2.4	-43%	3.5	3.6	3%	3.8	3.2	-16%
	80				4.4	3.6	-18%	5.1	4.4	-14%	5.8	5	-14%

The most important result of note is that the CDCL-B&B algorithm explores fewer nodes during its BFS search than BIP-B&B does, for any problem with at least one clause. This implies that the inclusion of conflict-direction improves search efficiency, in



some larger problems by a factor of over 50%. Additionally, the number of constraints in the average relaxed CDCL-B&B LP problem is significantly smaller than the number of constraints in the average relaxed BIP-B&B problem. This is because every possible choice for avoiding every obstacle is encoded as a constraint in a BIP, while obstacles are encoded as disjunctions in a Clausal LP. These disjuncts are not included in the LPs that must be solved at each node. Resolution of a clause does increase the number of constraints by 1 at each deeper tree level, but even in the worst case this adds  $z$  constraints to a leaf node, where  $z$  is the number of obstacles. By comparison, even the *root* node of a BIP search tree has  $5 \cdot z$  extra constraints because of obstacles: one for each direction and one for each the obstacle to require the selection of a direction.

The performance of the algorithms is identical for problems involving no obstacles (meaning no disjunctive clauses), because these problems require encoding no decisions. In general there are  $v \cdot z \cdot t$  decisions to encode for a problem with  $v$  vehicles,  $z$  obstacles, and  $t$  time steps. Each decision has four choices, for the four ways to avoid an obstacle (north, south, east, and west). A CPP problem with no obstacles means that the BIP encoding requires no binary variables and no extra constraints, and that the Clausal LP encoding requires no clauses. Therefore no search takes place, and a single, identical LP is solved by both algorithms

The next result to note is that the difference in the number of LPs solved and in LP size increases in real numbers with both the number of vehicles and the number of obstacles. This result is also not surprising, since the number of total decisions is a function of both these values, as previously shown. As the number of decisions increases, the comparative benefits of encoding in a Clausal LP form and of conflict-directed search increases as well.

The final result of note is that the maximum queue size of CDCL-B&B is around 10% smaller than BIP-B&B. This is despite the fact that CDCL-B&B uses a best-first search policy, which has a larger memory usage,  $O(b^d)$ , than the depth-first search used by BIP-B&B,  $O(b+d)$ . The reason for the reduced CDCL-B&B queue size is likely because the number of nodes to explore in the search tree is so much smaller than in BIP-B&B. In other words,  $b$  and  $d$  are so much smaller for an HDCP encoded as a Clausal LP than for the same HDCP encoded as a BIP that the overall memory use by CDCL-

B&B is slightly smaller. Another factor in the smaller queue size is the more effective pruning policies used by CDCL-B&B.

In summary, using the Clausal LP form and conflict-directed search can significantly improve the efficiency of solving hybrid decision-control programs, and this improvement becomes greater as problem complexity increases. This efficiency improvement is dependent on identifying minimal conflicts for infeasibilities, which can be a time-consuming process. Hence, a vital area for future investigation is improving the efficiency of the conflict extraction process. This and other potential enhancement will be examined in the next chapter.

## 7 Future Work

This chapter reviews some areas for potential future work building on the Clausal LP framework and the CDCL-B&B algorithm presented in this thesis. Section **Error! Reference source not found.** describes methods for reducing the total number of states in a CLP that is build directly from a BIP. Section **Error! Reference source not found.** looks briefly at the problem of *extracting* conflicts, which was deferred when conflict resolution was discussed earlier. Finally, Section 7.3 describes how the CDCL-B&B algorithm can be integrated with a cooperative path planning system.

### 7.1 Modifications to Clausal LPs

It may be possible to reduce the number of assignments that need to be explicitly considered from the complete enumeration of  $2^{N_b}$  (identified in Chapter 2) to a smaller number. If two binary variables do not ever appear in the same constraint, then the constraints that contain them can be mapped into independent clauses. More generally, each clause in the Clausal LP corresponds to a partition of the total set of binary variables; each variable in the partition must be independent of any variable in any other partition. The greater the number of partitions in the binary variable set, the further reduced the enumeration of assignment possibilities. For example, if a BIP included only constraint {2.2-18} and constraint

$$x \cdot b_3 + y \geq 2 \quad \{\text{Error! Reference source not found.-21}\}$$

then the BIP could be translated into two a Clausal LP containing clauses {2.2-20} and

$$y \geq 2 \vee x + y \geq 2 \quad \{\text{Error! Reference source not found.-22}\}$$

Together these two clauses enumerate only 6 possible disjuncts instead of  $2^{N_b} = 2^3 = 8$ . The extreme case occurs when every binary variable in the BIP never appears in the same constraint as any other binary variable, in which case only  $2 \cdot N_b$  disjuncts are required.

A final possibility to consider in translating a BIP to a Clausal LP occurs when a binary variable appears in multiple constraints. Enumeration of the possible assignments in this scenario requires a Clausal LP that can add an arbitrary number of additional

inequalities when a particular choice is made, not just one inequality. For example, if a BIP included the constraints

$$\begin{array}{ll} x \cdot b_1 + y \geq 5 & \{\text{Error!} \\ \text{Reference source not found.-23}\} \\ x - y \cdot b_1 \geq 3 & \{\text{Error!} \\ \text{Reference source not found.-24}\} \end{array}$$

then the Clausal LP would need to contain a clause

$$(y \geq 5 \ \& \ x \geq 3) \vee (x + y \geq 5 \ \& \ x - y \geq 3) \quad \{\text{Error!} \\ \text{Reference source not found.-25}\}$$

While a disjunct containing this kind of conjunction of inequalities is not considered in this thesis, a slightly modified version of the Clausal LP framework would easily be able to deal with this more general type of clause. Instead of associating each disjunct with a single inequality, a given disjunct could be associated with multiple inequalities that would all be added to a relaxed LP when the disjunct is selected. When clauses are viewed as simple propositional variables, the multiple inequalities would all be abstracted away, and each disjunct could be viewed as a single symbolic term.

## 7.2 Conflict Extraction

Conflict extraction and kernel generation must balance two competing requirements. On the one hand, as the number of constraints used in a constituent kernel decreases, the number of ways to resolve the conflict decreases as well. Therefore conflict resolution would identify fewer states. This is very cost-efficient since more states can be ruled out without performing any intensive computation. On the other hand, exhaustively searching all possible combinations of constraints for the smallest conflict that results in infeasibility is a problem of non-polynomial complexity. This computational cost is worsened by the fact that the only way to check feasibility of a subset of constraints is to run Simplex or an equivalent algorithm. This is self-defeating since the entire motivation behind conflict-direction is to reduce the total number of LPs that had to be solved. Therefore performing an exhaustive search to find the smallest-size conflict is not viable.

We can use heuristic methods for identifying “sufficiently” small conflicts. The use of heuristic methods during *conflict extraction* does not in any way compromise the

optimality of the overall solution. Failure to find the smallest possible conflict only reduces the effectiveness of the conflict-direction technique in guiding the overall search, and so reduces speed, not optimality. In the extreme, failure to find *any* conflicts at all would nullify the conflict-direction element of the algorithm entirely, and simplify it to a simple best-first search, which will still find the optimal solution but only through an inefficient search that investigates many infeasible nodes.

### **7.3 Integrating CDCL-B&B with a Planning System**

The application of the conflict-directed algorithm is much more significant when described in light of integration with a full cooperative path-planning system. The Kirk path planning system performs high-level goal planning for multiple vehicles, goals, and obstacles, and creates a set of control vectors that can be passed to simulated or real vehicles. Section 7.3.1 introduces problems in this domain as illustrative examples for the CDCL-B&B algorithm. These problems are also used as a metric for comparison with the benchmark: a mixed integer-linear program formulation solved using a combined Simplex-Branch and Bound technique. This chapter also covers the Kirk planning system, which performs higher-level activity planning for cooperative vehicles, and provides a system into which the CDCL-B&B algorithm could potentially be integrated and which would serve as a test bed (Section 7.3.2). Finally, Section 7.3.3 details the architecture of the unified Kirk-CDCL-B&B system; meeting the interface required by this architecture places certain demands and constraints on the HDCP problem formulation, which are reflected in Chapter 2.

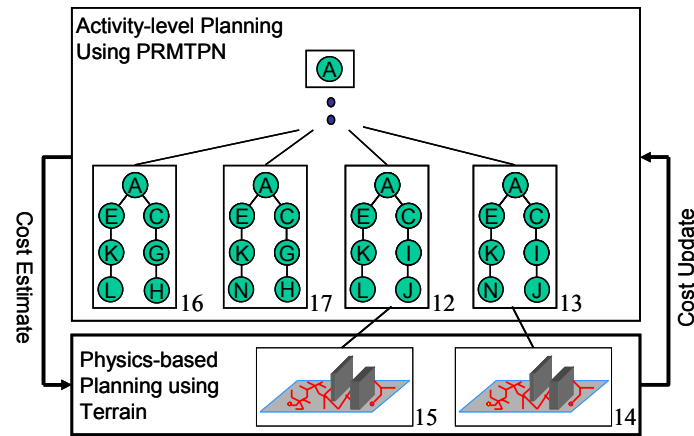
#### **7.3.1 Unified Activity and Trajectory Planning**

Integration of Kirk's higher-level activity planner and the CDCL-B&B-based trajectory planner must maintain two characteristics: optimality of the final hybrid solution, and efficiency of plan generation. Plan optimality is threatened by the fact that the Kirk activity planner ignores terrain details and vehicle dynamics when it selects a roadmap path-plan. This means that its cost estimate for each candidate will be optimistic, and it may not return a candidate that is optimal under the detailed model. For example, Figure 37 shows that pursuing trajectory planning for the activity plan with cost

12 results in a solution with cost 15, while a less optimal activity plan might result in a more optimal set of trajectories. Efficiency must be explicitly ensured because trajectory planning for an unbounded problem for all possible plans is highly complex and is likely to be intractable. Since much of the planning may take place online, it is vital to minimize planning cost.

### 7.3.2 A Unified Architecture for Global Optimality

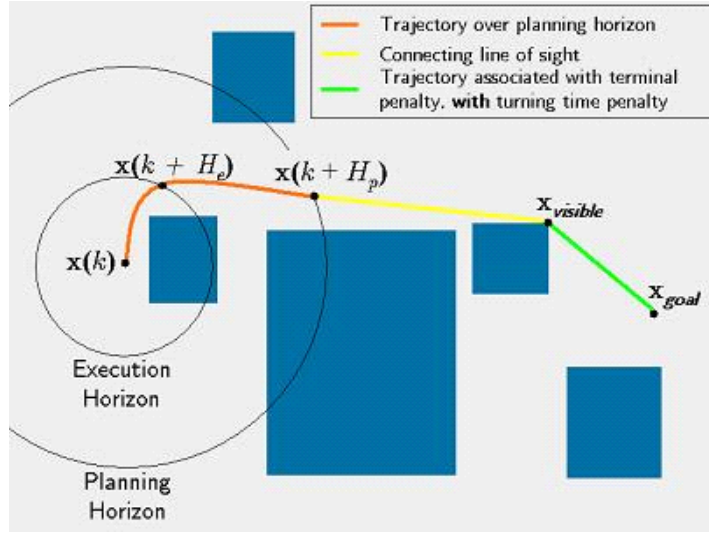
The two goals of efficiency and optimality can be met by integrating Kirk and CDCL-B&B performing A\* search within a generate-and-test cycle. The higher-level activity planner will generate a candidate plan using an optimistic cost estimate, and this cost will be updated using the trajectory planner. Figure 37 shows this separation between the activity planner (top) and the trajectory creation (bottom). The combined planner generates candidate activity plans and elaborates their trajectory plan up until the point that the next best candidate plan has cost worse than that of the best elaborated trajectory plan.



**Figure 37:** Generate-and-test cycle between activity-level planning and trajectory planning.

So long as a purely optimistic admissible heuristic is used to estimate the cost of the solutions generated by the activity planner, overall optimality is guaranteed. This can be accomplished using, for example, a visibility graph-based estimate of cost, which is optimistic since it does not take variable terrain complexities into consideration. The more rigorous trajectory planner, using detailed physics, vehicle dynamics, and terrain

maps, can extend the visibility graph estimates. Any admissible heuristic estimate for plan cost can be substituted for a visibility graph for this methodology to be effective.



**Figure 38:** Trajectory planning using a receding limited horizon; courtesy of [REFERENCE]

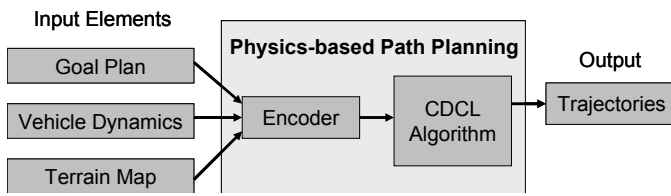
In Figure 37, for instance, initial optimistic cost estimates for the plans using a visibility graph are 16, 17, 12, and 13. Extending the lowest-cost plan, the combined algorithm achieves a trajectory with cost 15; but there is a candidate activity plan with lower cost. When Kirk-CDCL-B&B extends the cost-13 candidate plan, it discovers a trajectory of cost 14, which is superior to all other complete trajectory plans and all candidate activity plans, and hence the algorithm terminates.

Because trajectory calculations are generally more costly than activity plan generation, the complete trajectory plan is created only over a *receding limited horizon* [citation]. Instead of using a model of the entire map, CDCL-B&B only plans within a limited *planning horizon*, shown in Figure 38. Beyond this horizon, a faster, less accurate cost estimate such as the visibility graphs is used. As execution continues, more and more of the estimated path is brought inside the planning horizon and the cost and path are updated using CDCL-B&B. [REFERENCE] has demonstrated successful full-path vehicle control using MILP and visibility graph-based receding horizon control.

### 7.3.3 Impact of Unified Planning on Kirk-CDCL-B&B Interface

The described unification of activity and trajectory planning has an impact on the required encoding of HDCPs in three ways. First, the CDCL-B&B-based trajectory planning algorithm must be able to interface with Kirk and accept as one of its inputs the goal plan generated by Kirk. Second, use of a generate-and-test cycle makes it possible to run the more costly trajectory planning on only a single goal in the goal plan. Finally, the limited-horizon approach restricts the total number of time steps in the problem and reduces the problem complexity. The effect of these three factors on the problem encoding is discussed further here and is reflected in Sections 5.1 (for the BIP formulation) and 5.2 (for Clausal LPs).

The first requirement of interfacing with the Kirk planner impacts the I/O specifications for the module containing the CDCL-B&B-based planner. Figure 39



**Figure 39:** I/O specification for trajectory planner.

shows the blackbox-view of this module. The inputs consist of the goal plan, a description of the vehicle dynamics, and a representation of the terrain map. After planning is complete, the

module returns as output the optimal trajectories or control vectors that achieve the goal plan and achieve obstacle and collision avoidance for all the vehicles in the problem.

Integration with Kirk imposes a clearly-defined goal plan syntax that must be recognized by the trajectory planner. The goal plan takes the form of a *temporally flexible plan* where individual nodes specify activities that must be accomplished by vehicles, but without specific time stamps assigned to the nodes *a priori*. The time at which these nodes are resolved must instead be explicitly *scheduled*.

However, the goal plan does contain *temporal constraints* that place upper and lower bounds on the time duration of the activities and the arcs between activities. Only the nodes that involve movement of the vehicles between locations are relevant during trajectory planning; these must be extracted along with their associated temporal constraints. The relevant goal plan structure, therefore, contains nodes that associate



vehicles with locations on the terrain map, combined with upper and lower bound temporal constraints between nodes.