Learning from Conflict in Multi-Agent, Classical, and Temporal Planning

Toby Oliver Davies
orcid.org/0000-0001-5328-4316

Submitted in partial fulfilment of the requirements of the degree of

Doctor of Philosophy

May 2017

Department of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

Abstract

Conflict-directed learning has been one of the most significant advances in combinatorial optimisation in recent decades, but until very recently has not been directly applied to planning problems. Iterated compilations to conflict-directed solving technologies have seen significant success in planning, but these compilations do not learn from conflicts encountered during earlier iterations, seriously limiting their scalability and applicability to cost-optimal planning. This thesis introduces a number of novel algorithms that take into account information derived from conflict on some important variants of the planning problem. These approaches use combinations of conflict-directed encodings, relaxation-refinement procedures, and state-based search. Our results show that conflict-directed reasoning is a highly effective approach to cost-optimal planning when appropriate decompositions and explanations are known. Automatically deriving the most effective of these decompositions and explanations from unannotated problem descriptions will be a fruitful avenue of further work. We also highlight the unique potential for learning strongly generalisable knowledge from conflict within plan-space planning algorithms. This learned knowledge enables our plan-space search algorithms to compete with state-space search on some standard domains, and outperform it by orders of magnitude in appropriately structured industrial ones.

Declaration

This is to certify that

1. the thesis comprises only my original work towards the PhD,
2. due acknowledgement has been made in the text to all other material used,
3. the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Toby Oliver Davies, May 2017

Acknowledgements

I would like to thank my supervisors: Adrian Pearce; Harald Søndergaard; Peter Stuckey; and Nir Lipovetzky, for guiding me in interesting directions, and helping me to produce a body of research of which I am immensely proud.

Specifically: Adrian, thank you for being consistently supportive and encouraging. Harald, thank you for being immensely helpful with *so many* little things. Peter, thank you for pushing me. Nir, thank you for the enormous time you put in to listening to and seriously discussing all of my crazy ideas.

To my committee chair, Tim Baldwin, thank you for making yourself available for meetings at far shorter notice than I had any right to expect.

Thank you to the whole ICAPS community who I have immensely enjoyed spending two fascinating conferences with, and especially the ICAPS 2015 chairs who awarded the best paper award to the work presented in chapter 5.

To my partner Brittney, thank you for convincing me I could finish what I started when I thought I had bitten off far more than I could chew, for keeping me clothed and fed when, in the midst of writing I might have forgotten such things. And perhaps most importantly, for reminding me that things other than work are important too!

To my mother, thanks for so many things over the years. I would never be here were it not for you.

Finally, thanks to Data61 (formerly National ICT Australia), the Australian Research Council, the University of Melbourne and, indirectly, Biarri, without whose financial support I could not have completed this research.

Preface

The research presented in this thesis was conducted in the Computing and Information Systems department at the University of Melbourne, in collaboration with my supervisors Adrian Pearce, Peter Stuckey, Nir Lipovetzky, and Harald Søndergaard, and additionally Graham Gange

This thesis includes material published in the following conference papers:

- Toby O. Davies, Adrian R. Pearce, Peter J. Stuckey, and Harald Søndergaard (2014). "Fragment-Based Planning using Column Generation". In: *International Conference on Automated Planning and Scheduling (ICAPS 14)*, pp. 83–91 (Presented in chapter 3)

- Toby O. Davies, Adrian R. Pearce, Peter J. Stuckey, and Harald Søndergaard (2015). "Optimisation and Relaxation in the Situation Calculus". In: *Autonomous Agents and Multiagent Systems (AAMAS 15)*, pp. 1141–1149 (Presented in chapter 4)

- Toby O. Davies, Adrian R. Pearce, Peter J. Stuckey, and Nir Lipovetzky (2015). "Sequencing Operator Counts". In: *International Conference on Automated Planning and Scheduling (ICAPS 15)*, pp. 61–69 (Presented in chapter 5)

- Toby O. Davies, Adrian R. Pearce, Peter J. Stuckey, and Nir Lipovetzky (2016). "Sequencing Operator Counts". In: *International Joint Conference on Artificial Intelligence (IJCAI 16)*, pp. 4140–4144 (Presented in chapter 5)

- Toby O. Davies, Graeme Gange, and Peter J. Stuckey (2017). "Automatic Logic-Based Benders Decomposition with MiniZinc". In: *AAAI Conference on Artificial Intelligence (AAAI)* (Presented in chapter 7)

This research was financially supported by the following scholarships and awards:

- Australian Postgraduate Award
- National ICT Australia (later Data61) Top-Up Scholarship
- George Lansell Mining Scholarship
- International Conference on Automated Planning and Scheduling 2015 Outstanding Paper Award

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Planning is the problem of finding a sequence of actions an agent can perform that leads them from some initial state to a state satisfying some desired goal condition. A simple example of such a problem is finding a sequence of actions that would enable a robot to move a ball between two rooms. Such a plan might be: pick up the ball, move to the other room, then drop the ball. This is an obviously contrived example, but planning has industrial applications too: for example if the robot was a train, the ball cargo, and the rooms cities or ports. Especially if the train has to somehow coordinate with a number of other trains sharing a busy track network so as not to crash.

Search is one of the most pervasive approaches to planning and scheduling problems, in simple terms, search consists of interleaving guesses with reasoning, to try to incrementally build a large number of different solutions. A conflict occurs when reasoning determines that one or more of the guesses made were wrong. For example, when planning a set of train services, a planner may guess that two trains should head towards each other at the same time, if there is no siding to pass on, this will inevitably lead to a crash.

Learning from this conflict might involve realising that the underlying cause is the relative order of the trains departing their stations and passing one-another. One might reason, for example, that the colour of those trains, who is driving, the particular cargo loaded onto them, or indeed the precise time of departure

Algorithm	Conflict	Knowledge Learned
FragPlan (Ch. 3+4)	Resource limitations	Improving Agent Plan
OpSeq (Ch. 5)	Unsequencable	Generalised Landmark
CDHL (Ch. 6)	Suboptimality	Clause
MznLBBD (Ch. 7)	Unschedulable	Benders Cut
OpSched (Sec. 7.6)	Unschedulable	More Events Needed

Table 1.1: Summary of contributions

is not relevant to the conflict. Recognising what is and is not responsible for conflict allows algorithms to avoid making the same mistake many times in many superficially distinct states generated during search.

The types of guesses made during search, and their interactions with the reasoning techniques employed determine the types of conflict that can be detected, which in turn define the kinds of knowledge that may be derived from these conflicts.

The focus of this thesis is on applying and extending this kind of conflict-directed reasoning to several important variants of the planning problem. To this end we introduce a number of novel algorithms, associated kinds of conflict, and representations of knowledge derived from these conflicts. These algorithms are summarised in table 1.1. All of the approaches we introduce are capable of finding the optimal plan and proving its optimality.

Thesis Organisation

The next chapter covers related work, expands on our view of the relationship between conflict and learning, and highlights the bigger questions asked and answered by this thesis. The chapters comprising the body of the thesis are intended to be self contained. A reader should be able to read any chapter without the need to refer to any other much like any paper. This does mean that some backgound information is repeated in several chapters, but this is intentional in order to make

reading the thesis out of order less daunting.

These chapters are grouped into 3 parts, the introductions to each of these parts highlight the common themes, the relationships between the chapters and the thesis as a whole.

Part I considers multi-agent planning problems, the agents in these chapters can be black boxes, but we use the Golog language to describe and analyse them. This section covers the **FragPlan** algorithm which enables agents to plan largely independently, and to learn from the conflicts they cause in terms of shared resources.

Part II considers classical planning problems, and covers both Operator Sequencing and Conflict-Directed Heuristic Learning. Operator Sequencing (**OpSeq**) separates the choice of the count of actions in a plan from the order of those actions, then learns information about the former when conflicts are detected while searching for the latter. Conflict-Directed Heuristic Learning (**CDHL**) attempts to learn how far from a goal each state is in terms of desirable properties that do not yet hold in a state.

Part III considers temporal problems, starting with Automatic LBBD for MiniZinc, which solves alternative scheduling problems, a half-way point between temporal planning and scheduling. We then move on to Operator Scheduling, a theoretical extension of the automatic LBBD approach to tackle true temporal planning problems.

Chapter 3 is derived from the paper Toby O. Davies, Adrian R. Pearce, Peter J. Stuckey, and Harald Søndergaard (2014). "Fragment-Based Planning using Column Generation". In: *International Conference on Automated Planning and Scheduling (ICAPS 14)*, pp. 83–91.

Chapter 4 is derived from the paper Toby O. Davies, Adrian R. Pearce, Peter J. Stuckey, and Harald Søndergaard (2015). "Optimisation and Relaxation in the Situation Calculus". In: *Autonomous Agents and Multiagent Systems (AAMAS 15)*,

pp. 1141–1149.

Chapter 5 is derived from the paper Toby O. Davies, Adrian R. Pearce, Peter J. Stuckey, and Nir Lipovetzky (2015). "Sequencing Operator Counts". In: *International Conference on Automated Planning and Scheduling (ICAPS 15)*, pp. 61–69. This paper was also reproduced with permission from AAAI in abridged form as Toby O. Davies, Adrian R. Pearce, Peter J. Stuckey, and Nir Lipovetzky (2016). "Sequencing Operator Counts". In: *International Joint Conference on Artificial Intelligence (IJCAI 16)*, pp. 4140–4144.

Chapter 7 is derived from the paper Toby O. Davies, Graeme Gange, and Peter J. Stuckey (2017). "Automatic Logic-Based Benders Decomposition with MiniZinc". In: *AAAI Conference on Artificial Intelligence (AAAI)*.

Chapter 2
Knowledge and Conflict

This chapter focuses on placing the thesis in a broader context, specific technical background and notation will be introduced in chapters as needed.

This thesis assumes knowledge of deterministic planning to the level of the textbook "A Concise Introduction to Models and Methods for Automated Planning" (Geffner and Bonet, 2013). Familiarity with Satisfiability (Biere, Heule, and Maaren, 2009) and Mathematical Programming (Papadimitriou and Steiglitz, 1982) would also be an advantage.

2.1 Preliminaries of combinatorial optimisation

Combinatorial problems are characterised by the need to choose values for a set or sequence of variables such that they satisfy some conditions. This means there are a finite, though possibly enormous, number of possible solutions: every possible combination or sequence of values. The most basic algorithm solving this class of problems is brute-force, where every one of these potential solutions is generated and tested to see if it satisfies the required conditions.

Problems in this class are often NP-hard, so algorithms must make a trade-off if they want to guarantee they will find a solution if one exists. Algorithms guaranteed to find a solution are called complete algorithms and, assuming $P \neq NP$, must have super-polynomial runtimes. Incomplete algorithms in contrast make

no such guarantee, but instead typically guarantee polynomial runtime and/or bounded memory usage. This thesis focuses on complete algorithms.

The most common class of complete algorithm is search. Searching for a solution consists of choosing variables to assign one at a time and guessing all possible values for that variable. Obviously the algorithm can only consider one of these guesses at a time, so the other possibilities must be stored for later consideration. The structure which stores such a partial assignment is called a "search node". The set of search nodes that still need to be considered is called an "open set" or "open list".

Search techniques can perform empirically better than brute force only when they can either detect conflicts in these partial solutions and skip generating many of the possible search nodes, or if they choose which partial solutions to extend in such a way to find a solution early. Search algorithms primarily differ in the data-structure used to implement the open list, and the types of reasoning they use to detect conflicts.

The basic outline of a search algorithm is:

1. Add a search node representing having made *no* choices to the open set.
2. Pick some search node from the open set.
3. Reason about the consequences of the choices represented by that node.
4. Optionally **discard the node** if there is a conflict between these choices.
5. If all variables have been assigned, you have found a solution, terminate.
6. Choose one variable whose value is not known and add a search node to the open list for each choice of value that variable can take.
7. Repeat steps 2-6 while there are any nodes remaining in the open set.
8. If there are no open nodes remaining, there is no solution.

Technology	Pruning method
Constraint Satisfaction	Propagation implicitly prunes nodes that would be in conflict within a single constraint. Additionally, nodes containing a variable with no valid value after propagation are pruned.
Branch-and-bound	A relaxation of the problem is used to prune nodes with objective lower-bounds no better than each solution found.
IDA*	A relaxation of the problem is used to prune nodes with objective lower-bounds worse than a predetermined lower bound.
(Weighted-)A*	A relaxation of the problem is used to implicitly prune (i.e. never generate) nodes with objective lower-bounds no better than the best solution found. Additionally, nodes representing paths to previously seen states are pruned.

Table 2.1: Pruning in combinatorial search

Intelligently choosing the order to expand nodes, or discarding nodes, either implicitly or explicitly, are the only ways for such algorithms to be faster than brute-force. It is hoped that the cost of reasoning is paid for by a significant reduction in the number of nodes generated before a solution is found. The reason that a node can be **safely** discarded is what we refer to as a conflict.[1] A number of combinatorial search pruning strategies are summarised in table 2.1. We hope the reader is familiar with one or more of these techniques.

Satisfiability and Constraint Programming are two technologies which have made the greatest use of conflict-based reasoning. Constraint Programming (CP) is a generalisation of the Satisfiability problem (SAT) where a fixed finite set of variables are each assigned an arbitrary value from a finite domain (as opposed to just the values True or False for SAT). Both are NP-complete problems (Cook,

[1]Safely here means "without discarding the last optimal potential solution"

1971).

In these problems, the majority of nodes are pruned implicitly by (unit-)propagation. Additionally, a node may become "failed" when a variable has no values left in its domain. In this case, there exists some propagation rule that could have pruned the failed node earlier in the search tree. This rule is a consequence of some subset of the constraints, and is typically learned as a Boolean clause. This type of learning was originally introduced in the Chaff solver for SAT (Moskewicz et al., 2001), and by Lazy Clause Generation for general constraint programming (Ohrimenko, Stuckey, and Codish, 2009).

These learned clauses are often referred to as conflicts in the SAT and CP literature, but in this thesis we will show why it is interesting to maintain the distinction, especially in the context of planning and plan-space search.

2.1.1 Planning problems

Planning problems are characterised by the need to find a **sequence** of decisions satisfying some properties, as opposed to a set of assignments whose order is not significant.[2] The length of this sequence is typically exponential (or even unbounded) in the size of the problem description. A planning problem will have an initial state, a goal condition, and a set of available operators (equiv. actions). The solution to a planning problem is a sequence of operators called a plan. The operators in a plan will have preconditions and effects, such that the first operator's preconditions hold in the initial state; each subsequent action's preconditions hold in the state resulting from applying the effects of all prior actions in sequence; and the state resulting in applying the whole plan satisfies the goal condition.

There are many planning formalisms considered in the literature, in particu-

[2]Not significant for correctness, but often crucial for performance.

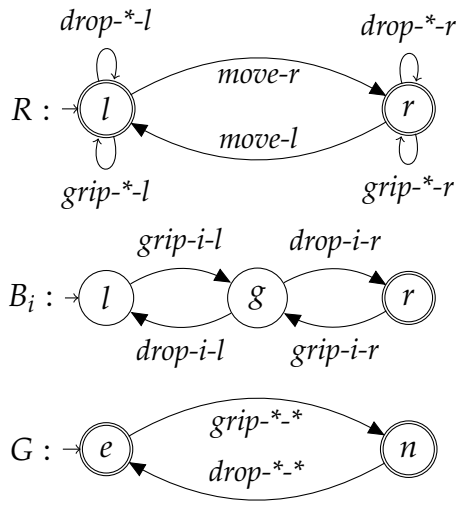


Figure 2.1: Domain Transition Graphs in gripper.

lar the equivalent STRIPS (Fikes and Nilsson, 1971) and SAS+ (Bäckström, 1992) classical formalisms. We describe the technical details of these formalisms in Part II. Both formalisms describe PSPACE-complete planning problems, requiring at most an exponential number of operators in a plan.

More expressive formalisms exist which incorporate programming-language constructs such as loops and recursion, and as such are typically Turing-complete. In particular we will describe the *Golog*-family of languages in Part I. Another important formalism of this kind is Hierarchical Task Networks (Sacerdoti, 1975; Tate, 1977), we expect the work described in Part I will have analogues in this formalism, but exploring this is beyond the scope of this thesis.

A SAS+ planning problem can be visually represented by a set of Domain Transition Graphs such as those seen in Figure 2.1. Domain Transition Graphs can be very simply transformed into Deterministic Finite Automata (DFAs) by adding a self-loop at every node in each graph for each operator that does not appear anywhere else in that graph. A SAS+ planning problem can then be seen as the problem of finding a sequence that is simultaneously accepted by all of the DFAs.

The simplified gripper domain shown in Figure 2.1 with 2 balls, B_1 and B_2 is a simple example of a classical planning problem. The goal is to move both balls from the left room to the right, using a robot with a single "gripper" which can hold only one ball at a time. The robot starts in the left room, and can only pick up and drop balls in the room it is currently occupying. For each automaton in Figure 2.1, the initial state is marked by an incoming arrow, and the states consistent with the goal are double circled. The variable R represents the location of the robot (in the left or right room) ; B_i represents the location of ball i, (in the left or right room, or in the gripper); G represents the state of the gripper (empty or non-empty).

State-space search has been the most widely-used and widely studied approach to planning over the last decade. It is characterised by searching in the state-space, starting with the initial state in the open list, search proceeds directionally by adding successors of the current state to the open list.[3] This effectively means that state-space search considers partial plans only as prefixes, and inserts operators into the plan only at the end.

Heuristic search is a particularly common approach to state-based search, it reasons about the cost from the current state to the goal, this focuses on intelligently choosing plan prefixes to extend, instead of pruning. Heuristic functions, or just heuristics, map states to an estimate of the remaining cost. Heuristics may be "admissible", meaning that their estimate is guaranteed to be a lower bound on the true minimum cost to the goal. Such admissible heuristics can be used as in a pruning method in addition to guiding the order of expanding nodes in the open list.

Plan-space search is, broadly, any planning approach whose nodes do not map directly to a state in the state-space, searching over the space of possible plans,

[3]This process also works in reverse, starting from the goal, but is typically less effective.

rather than the state-space itself. Partial-order planning is the most widely known plan-space search algorithm, commonly found in AI textbooks (Penberthy, Weld, et al., 1992; Russell and Norvig, 2003). In partial-order planning, operators are inserted as supporters of unachieved goals and sub-goals, and ordering constraints are added only between a subset of operators to resolve "threats".

2.2 Types of conflict

A conflict is a flaw that can be detected at a search node. Obviously the conflicts that can be detected are a function of the decisions encoded in each search node, and the kind(s) of reasoning conducted at each node.

In CP and SAT, each search node encodes a restricted domain of the variables. A conflict will occur iff some variable has no valid values after propagation (Marriott and Stuckey, 1998). This is the only kind of failure that can be detected given the representation of search nodes. Because of this limitation, it is common for constraint programming models to introduce redundant variables and constraints which allow certain domain-specific conflicts to be detected earlier in the search (Cheng et al., 1999). Thus effectively delegating the problem of identifying the most useful kind of conflict to the modeller.

In Branch-and-Bound, each node stores the optimal solution to a relaxed problem. This allows nodes to be pruned when either the relaxation is infeasible, or the objective of the relaxed solution is larger than the incumbent. Depending on the kind of relaxation, it may be strengthened by further inference, such as cut generation in Mixed-Integer Programming.

In (forward) state-space search, a search node is a state in the state space, together with the cheapest path reaching that state from the initial state. Necessarily then, any conflict that can be detected in such a planning algorithm must be a function of the state variables, plus possibly the path cost ("g-value") and/or

Algorithm	Search Node	Open List	Conflict	Knowledge Learned
IDA* (Korf, 1985)	Prefix + f	Stack	Insufficient f-budget	Increase $h(s_0)$
Alpha-Pruning (Korf, 1990)	Prefix	Stack	$h(s)$ too low	Increase $h(s)$
SAT-Plan (Kautz and Selman, 1992)	SAT Formula	Number of layers	Insufficient layers	Add layer
h^{++}(Haslum, Slaney, and Thiébaux, 2012a)	Relaxed-plan	MIP	Deleted goal/prec	Prevailing conjunction
CEGAR (Seipp and Helmert, 2013)	Abstract plan	Abstraction Graph	Invalid Path	Split abstraction node
PDR (Suda, 2014)	Prefix + len.	Stack or queue	Dead-end	Blocked-cube
FragPlan (Ch. 3+4)	Agent-Plans	Linear Program	Resource limitations	Improving agent plan
OpSeq (Ch. 5)	Operator Count	MIP	Unsequencable	Generalised Landmark
DFS-CL (Steinmetz and Hoffmann, 2016)	Prefix	Stack	Dead-end	Critical-path conjunction
CDHL (Ch. 6)	Prefix + f	Stack	Suboptimality	Cost-Clause
MznLBBD (Ch. 7)	Resource-alloc.	MIP	Unschedulable	Benders Cut
OpSched (Sec. 7.6)	Sched. + count	CP	Non-empty count	Add Events

Table 2.2: Characterising learning planning algorithms.

heuristic estimate ("h-value"), as this is the only information represented in the search node.

In contrast to SAT, CP, or state-space search, plan-space search does not have a universally agreed-upon search-node representation. As a consequence, algorithms can conceivably be designed that can detect types of conflict as yet unconsidered. We introduce a very general class of conflict for classical planning ("unsequenceability") in chapter 5.

2.3 Knowledge derived from conflict

Conflict-directed algorithms learn something from each conflict that occurs while solving a single instance. This knowledge guarantees that the same conflict will never be seen in any other search node.[4]

Importantly this is distinct from the relatively well studied problem of learning from separate but related instances. There have been several "learning tracks" of the international planning competition dedicated to this latter problem, and is typically solved by techniques such as algorithm configuration and portfolio construction.

Effective conflict-derived knowledge must avoid an exponential amount of future work. In order to do so, it must generalise to prune an exponential number of future nodes; and resolve with other learned knowledge to prune whole subtrees, not just leaves. Ideally it will also synergise with the branching strategy in some way, enabling the search to focus on the aspects of the problem that are responsible for failure.

The ubiquitous A* algorithm can definitely be described as learning path dominance relationships. However it does not do so based on conflicts: it learns a

[4]So long as that knowledge is kept: many algorithms "garbage collect" knowledge that is not significantly pruning search. This knowledge can be re-learned if it becomes relevant again later.

closed list regardless of whether multiple paths to a state exist. In this sense it learns from (partial) success rather than conflict.

In table 2.2, we list several algorithms that have some conflict-directed properties. The earliest algorithms in the first part of table 2.2 learn only basic information from conflict, specifically they do not attempt to generalise the learned knowledge.

Iterative Deepening A* learns only about the initial state. With the addition of a transposition table it eventually learns the perfect heuristic value for each node on the optimal path (Reinefeld and Marsland, 1994), however the learned knowledge applies only to states that have already been seen, and is not generalised in any way. Alpha-Pruning (Korf, 1990) is substantially similar to IDA* with a transposition table, introduced in the same paper as (Learning) Real-Time A*, as the offline improvement procedure for those algorithms. It differs from IDA* with a transposition table only in that it allows the specification of a finite look-ahead depth to use to learn a better heuristic value. Since this distinction is not particularly significant in the context of this thesis, we will focus on IDA* with transposition tables over these more complex algorithms.

DFS-CL (Steinmetz and Hoffmann, 2016), and our "Conflict-Directed Heuristic Learning" (CDHL, chapter 6) both attempt to solve planning problems depth-first, similarly to IDA*. They are differentiated by DFS-CL focusing on finding *a* solution, thus recognising only dead-ends as conflicts, whereas CDHL attempts to find the *optimal* solution. It does this by learning the perfect heuristic, thus treating particular kinds of inaccuracy in the heuristic as a conflict. In this sense CDHL generalises IDA* with a transposition table by generalising the reason for the heuristic's value to states that have not necessarily been visited yet.

Planning as Satisfiability (Kautz and Selman, 1992; Rintanen, 2009) is a family of approaches which compile the planning problem into a sequence of length-limited SAT problems. These algorithms will test if the planning problem can be

solved in n "steps",[5] if the answer is UNSAT, they will proceed to test if a plan of $n + k$ steps exists for some small constant k. Variants exist which test several of these formula lengths concurrently. In all cases, all that is necessarily learned from each conflict (UNSAT formula) is that the plan must be longer than n. Many SAT solvers allow the use of assumptions which can be used to enable clauses learned at shorter horizons to be reused. We describe how this is possible in our simple SAT encoding used in chapter 5, however to our knowledge this approach has not been investigated in traditional SAT-planners.

Other planning algorithms make use of SAT formulae. Most notably Property Directed Reachability (PDR), a verification algorithm adapted to planning by Suda (2014). PDR maintains a set of CNF formulas, where the i-th formula represents an over-approximation of the set of states which are at most i steps from a goal state. What really separates PDR from traditional SAT-planning techniques is its ability to detect unsolvability: if, at certain points in the algorithm, the formula representing the set of states at most n steps from the goal is identical to the formula for states at most $n + 1$ steps from the goal, then all subsequent formulae will be identical too, and the algorithm has reached a fixed-point. If the initial state is not in the set of states n steps from the goal at fixed-point, it will also not be in any set of states $m > n$ steps from the goal, and the planning problem is provably unsolvable.

Our "Operator Sequencing" approach (described in chapter 5) also falls into this SAT-enabled category. This approach combines ideas from SAT-planning with one of the most recent ideas from heuristic search: Operator Counting (Pommerening, Röger, et al., 2014). Operator Sequencing uses a state-of-the-art heuristic to estimate an "operator count" that may be sequenceable into a plan, then constructs a SAT-formula to test the sequencability of this operator count. A lin-

[5] In the simplest case, a "step" corresponds to exactly 1 action, but much more complex (and effective) definitions exist that allow many actions to occur in a single step.(Wehrle and Rintanen, 2007)

ear inequality is learned and added to the heuristic, encoded as a Mixed-Integer Program, to improve future operator counts.

"Counter-Example Guided Abstraction Refinement" (CEGAR) is another conflict-directed algorithm adapted from the verification community (Clarke et al., 2003; Seipp and Helmert, 2013). CEGAR is both a complete algorithm, and a method for constructing abstraction heuristics. An abstraction maps every state in the state space to a small number of abstract states. It begins with the simplest possible abstraction which distinguishes only goal states from all other states. An optimal path through the abstraction is tested in the real search space and, if it is not a valid plan, the abstraction is refined by splitting one or more abstract states on the optimal path in two.

In contrast to adapting ideas from the SAT and verification communities, other learning algorithms have their roots in heuristic search, in particular the h^m and h^C family of critical path heuristics. These heuristics include explicit conjunctions of facts in their estimation of goal distance. As the set of conjunctions considered approaches the full state space, these heuristics approach the perfect heuristic. The h^m heuristic considers all conjunctions of up to m facts, whereas h^C considers an explicit set C of conjunctions of arbitrary size. In addition to computing a critical path with these conjunctions, the conjunctions can be compiled into a new planning problem, denoted Π^C, such that h^1 in P^C is equivalent to h^C in the original problem (Haslum et al., 2009; Haslum, Slaney, and Thiébaux, 2012a). This compilation may exponentially increase the number of operators in the planning problem,[6] but allows other algorithms to be applied with potentially improved results. Heuristic methods for choosing C are the primary focus in the literature, however one conflict-directed approach has been covered: h^{++} computes an optimal delete-relaxed plan for Π^C, initially with only unit conjunctions in C, and adds conjunctions to the plan which invalidate the previous optimal plan

[6] unless conditional effects are added in the compilation, leading to a compiled problem denoted Π^C_{ce}

(Haslum, Slaney, and Thiébaux, 2012a). By iterating this process, h^{++} will eventually compute a delete-relaxed plan which is also a plan in the real state-space.

In addition to these techniques, we have introduced a number of plan-space search algorithms whose inspiration comes from the fields of Operations Research and Mathematical Programming. These fields are concerned with many real-world resource allocation and scheduling problems, and the distinction between scheduling and planning in real-world domains is often blurry.

In particular we utilise incremental decompositions to Mixed-Integer Programming problems. These decompositions are not typically described as conflict-directed, simply because that term is not widely used in the OR or MP communities, but the widely used and supported technique of cut-generation bears a strong resemblance to lazy clause generation. These decompositions rely on the observation that while many variables and constraints may be required to ensure the correctness of a MIP model, relatively few of these are actually necessary to find a solution and prove its optimality.

Logic-Based Benders Decomposition (LBBD) is a class of cut generation decomposition typically combining Mathematical Programming with Constraint Programming or SAT. Two approaches in this thesis are based primarily on LBBD: "Operator Sequencing" in chapter 5 and "Automatic LBBD" in chapter 7.

Automatic LBBD is not strictly speaking a planning algorithm, but an approach to solving general constraint programming problems specified in the MiniZinc modelling language. LBBD is particularly well suited to alternative scheduling problems, which are a useful middle-ground between traditional scheduling problems and temporal planning. This framework is then exploited by our "Operator Scheduling" approach in section 7.6 to try to bridge the gap between CP-based scheduling and temporal planning, by defining a way to model open-ended scheduling problems in CP so that the problem can be sensibly extended incrementally and make use of important resource-constrained scheduling con-

straints like “cumulative” and “alldifferent”.

Another Mathematical Programming decomposition: “Column-generation”, in contrast does not have an obvious analogue in SAT or CP, as the approach relies on the so-called “dual solution” of a linear program. The dual solution to a resource allocation problem assigns prices to each resource in a way that encourages agents to avoid resource bottlenecks, and thus to effectively cooperate. We introduce a multiagent planning algorithm: “Fragment-Based planning” in Part I of this thesis. This approach is a hybrid that uses state-space search as a subproblem solver within a plan-space algorithm that commits to one agent’s plan at a time. At each iteration, the dual solution is used to learn better plans for some subset of the agents which avoid conflicts seen in previous iterations and improve the current relaxed solution.

Part I

Multi-Agent Planning

Introduction to Part I

We begin with our multi-agent conflict-directed algorithms. Multi-Agent planning has the distinct advantage of an obvious decompostion, allowing us to focus on conflicts between agents.

Throughout this part we will use *Golog* variants to specify problems. Multi-agent STRIPS (Brafman and Domshlak, 2008) should expose sufficient structure to allow the application of the techniques described in the next two chapters, however we use the more general *Golog* formalism to highlight that these techniques work for teams of black-box agents.

Indeed the only requirements for agents is that finding each agents' optimal plan must be decidable; and each agent is aware of the state variables that might be affected by other agents. No agent needs to know the specific capabilities, or even the existence of any other agent.

Chapter 3 is an example-driven introduction to the basic "Fragment-Based Planning" algorithm, which is generalized slightly in chapter 4. Chapter 4 focuses on the necessary and sufficient conditions for modelling a domain in a way that is appropriate for Fragment-Based Planning.

Fragment-Based planning is a plan-space search algorithm which picks the optimal set of single-agent plans by exponential reduction to set-packing: It tries to find the maximum number of agents' plans (with minimum total cost) such that no shared resources are over utilised.

Chapter 3
Fragment-Based Planning Using Column Generation

We exploit the framework of column generation to tackle complex resource constrained multi-agent planning problems by learning a set of agent plans to avoid resource bottlenecks identified by a linear program.[1]

3.1 Introduction

The motivation for this work came from my experiences of modifying an existing rail service scheduling tool to handle an apparently small change to the structure of some generated services for a bulk-freight railway serving the Australian mining industry. Modifying the 10,000 lines of C++ data structures and procedures used to generate service plans took nine months. Planning formalisms such as *Golog* and *STRIPS* are extremely general modelling techniques that make them attractive given the constantly changing needs of industrial optimisation problems. However, pure heuristic search is often insufficient to solve large-scale industrial problems with hundreds of goals and thousands of time-points. We present the Bulk Freight Rail Scheduling Problem as a simplified example of such a domain.

[1]The research presented in this chapter was published in Toby O. Davies, Adrian R. Pearce, Peter J. Stuckey, and Harald Søndergaard (2014). "Fragment-Based Planning using Column Generation". In: *International Conference on Automated Planning and Scheduling (ICAPS 14)*, pp. 83–91

Multi-agent temporal *Golog* (Kelly and Pearce, 2006) and heuristic optimising *Golog* (Blom and Pearce, 2010) have been investigated separately. The combination of these features have potential industrial applications in scheduling problems, and the ability to use domain knowledge to supplement the search for solutions is attractive, however *Golog*'s lack of powerful search algorithms has limited its applicability. In this chapter we deal with this shortcoming for an important class of planning/scheduling problems.

Resource constrained planning problems are known to be challenging to solve using current technology, even in non-temporal settings (Nakhost, Hoffmann, and Müller, 2012). The Divide and Evolve meta-heuristic has been used to tackle temporal planning problems (Schoenauer, Savéant, and Vidal, 2006), it too repeatedly solves guided sub-problems but, unlike our approach, cannot prove optimality.

One of the key techniques behind our approach is linear programming, in particular the dual, which allows us to accurately predict the cost of resource consumption. Linear programming has been used by a number of planning heuristics (Van Den Briel et al., 2007) (Coles, Fox, Long, et al., 2008) (Bonet, 2013). However these heuristics have exploited only the primal solutions to the LP, whereas we use both the primal and the dual. Additionally we use the information in a way that cannot be described as a heuristic in the usual sense.

3.2 Preliminaries

The Situation Calculus and Basic Action Theories. The *situation calculus* is a logical language specifically designed for representing and reasoning about dynamically changing worlds (Reiter, 2001). All changes to the world are the result of *actions*, which are terms in the language. We denote action variables by lower case letters a, and action terms by α, possibly with subscripts. A possible world

history is represented by a term called a *situation*. The constant S_0 is used to denote the initial situation where no actions have yet been performed. Sequences of actions are built using the function symbol *do*, such that $do(a, s)$ denotes the successor situation resulting from performing action a in situation s. Predicates and functions whose value varies from situation to situation are called *fluents*, and are denoted by symbols taking a situation term as their last argument (e.g., $Holding(x, s)$).

Within the language, one can formulate *basic action theories* that describe how the world changes as the result of the available actions (Reiter, 2001). These theories, combined with *Golog*, are more expressive than STRIPS or ADL (Röger and Nebel, 2007). Two special fluents are used to define a legal execution: $Poss(a, s)$ is used to state that action a is executable in situation s; and $Conflict(as, s)$ is used to state that the set of actions as may not be executed concurrently.

High-Level Programs. High-level non-deterministic programs can be used to specify complex goals: the goal of a *Golog* program is to find a sequence of actions generated by some path through the program. We use temporal semantics from *MIndiGolog* (Kelly and Pearce, 2006) which builds on *ConGolog* (De Giacomo, Lesperance, and Levesque, 2000), and refer to these extensions simply as *Golog*. A *Golog* program δ is defined in terms of the following structures:

α	atomic action
$\varphi?$	test for a condition
$\delta_1; \delta_2$	sequence
while φ **do** δ	while loop
$\delta_1 \vert \delta_2$	non-deterministic branch
$\pi x.\delta$	non-deterministic choice of argument x
δ^*	non-deterministic iteration
$\delta_1 \Vert \delta_2$	concurrency

In the above, α is an action term, possibly with parameters, δ is a *Golog* program, and φ a situation-suppressed formula, that is, a formula in the language with all situation arguments in fluents suppressed (e.g. $Holding(x, s)$ would have s suppressed and should be written $Holding(x)$ in ϕ). Program $\delta_1|\delta_2$ allows for the non-deterministic choice between programs δ_1 and δ_2, while $\pi x.\delta$ executes program δ for *some* non-deterministic choice of a legal binding for variable x. δ^* performs δ zero or more times. Program $\phi?$ succeeds only if ϕ holds in the current situation. Program $\delta_1 \| \delta_2$ expresses the concurrent execution of programs δ_1 and δ_2. For notational convenience we add:

$\pi(x \in X).\delta$	equivalent to $\pi x.(x \in X)?;\delta$
`foreach` x `in` vs `do` δ	equivalent to $\delta_{[x/v_1]}; \cdots ; \delta_{[x/v_n]}$
`forconc` x `in` vs `do` δ	equivalent to $\delta_{[x/v_1]} \| \| \cdots \| \| \delta_{[x/v_n]}$

Here $\delta_{[x/y]}$ denotes the program δ where each occurrence of variable x has been replaced with the value y, and v_i is the ith element of the sequence vs.

Linear and Integer Programming. An Integer Program (IP) consists of a vector of binary decision variables,[2] usually denoted $\vec{x}$, an objective linear expression and a set of linear inequalities. We will refer to these inequalities as "resources" throughout this chapter, and each decision represented by the variable x_i can be thought of as using $u_{r,i}$ units of some resources $r \in R$, each of which has an availability of a_r.

A general form, assuming a set R of inequalities, where c_i, $u_{r,i}$ and a_r are constants, is:

[2] In general decision variables can be integer but binary decision variables suffice for our purposes.

$$\begin{array}{lll} \text{Minimise:} & \sum c_i \cdot x_i & \\ \text{Subject To:} & \sum u_{r,i} \cdot x_i \quad \leq a_r & \forall r \in R \end{array}$$

Finding the optimal solution to an integer program is NP-hard, however the linear program (LP), constructed by replacing the integrality constraints $x_i \in \{0,1\}$ with a continuous equivalent $x_i \in [0,1]$, can be optimised in polynomial time. This LP is known as the linear relaxation of the IP.

A model of this form where some variables are continuous and some are integral is called a Mixed Integer Program (MIP). To limit confusion, we will denote binary variables x_i and continuous ones v_i, we assume all $x_i \in \{0,1\}$ and $v_i \in [0,1]$ throughout this chapter.

The (M)IP is then solved using a "branch-and-bound" search strategy where some x_i which is fractional in the LP optimum is selected at each node and two children are enqueued with additional constraints $x_i = 1$ in one branch and $x_i = 0$ in the other. Heuristically constructed integer solutions provide an upper bound, and the relaxations provide a lower bound.

Solving the linear relaxation implicitly also optimises the so-called dual problem. Intuitively the dual problem is another linear program that seeks to minimise the unit price of each resource in R, subject to the constraint that it must under-estimate the optimal objective of the primal. We use λ_r to denote this so-called "dual-price" of resource r. [3] An estimate of the impact of consuming u additional units of resource r on the current objective can then be computed by multiplying usage by the dual price: $u \cdot \lambda_r$.

These dual prices allow us to quickly identify bottlenecks in a system, and

[3] Typically π is used for dual variables, but to avoid confusion with the *Golog* operator we use the less common λ.

give us an upper bound on just how far out of the way we should consider going to avoid them. This leads to the important concept of reduced cost: an estimate of how much a decision can improve an incumbent solution. Informally, reduced cost is the decision's intrinsic cost c_i, less the total dual price of the resources required to make this decision.

Given an incumbent dual solution $\tilde{\boldsymbol{\lambda}}$, a decision variable x_i has a reduced-cost $\gamma(i, \tilde{\boldsymbol{\lambda}})$, defined as:

$$\gamma(i, \tilde{\boldsymbol{\lambda}}) = c_i - \sum_{r \in R} \lambda_r \cdot u_{r,i}$$

This is guaranteed to be a locally optimistic estimate, so that, in order to improve an incumbent solution, we only need to consider decisions x_i with $\gamma(i, \tilde{\boldsymbol{\lambda}}) < 0$. Due to the convexity of linear programs, repeatedly improving an incumbent solution is sufficient to eventually reach the global optimum, and the non-existence of any x_i with negative reduced cost is sufficient to prove global optimality.

Column Generation and Branch-and-Price. Most real-world integer programs have a very small number of non-zero x_i. This property, combined with the need only to consider negative reduced-cost decision variables, allows us to solve problems with otherwise intractably large decision vectors using a process known as "column generation" (Desaulniers, Desrosiers, and Solomon, 2005). The name reflects the fact that the new decision variable is an additional column in the matrix representation of the constraints.

Column generation starts with a restricted set of decision variables obtained by some problem-dependent method to yield a linear feasible initial solution. With such a solution we can compute duals for this restricted master problem (RMP) and use reduced-cost reasoning to prune huge areas of the column space.

Incomplete and suboptimal methods of constructing integer feasible solutions are referred to in the Operations Research literature simply as "Heuristics". To

avoid ambiguity we will refer to them as "MIP heuristics". These are essential for both finding a feasible initial solution, and providing a worst bound on the solution during the branch-and-bound search process. These are analogues to fast but incomplete search algorithms in planning such as enforced hill climbing.

Column generation then proceeds by repeatedly solving one or more "pricing problems" to generate new decision variables with negative reduced cost and re-solving the RMP to generate new dual prices. Iterating this process until no more negative reduced cost columns exist is guaranteed to reach a fixed point with the same objective value as the much larger original linear program. We can then use a similar "branch-and-bound" approach as in integer programming to reach an integer optimum. This process is known as "branch-and-price".

Branching rules used in practical branch-and-price solvers are often more complex than in IP branch-and-bound and are sometimes problem dependent. The branch $x_i = 0$ does not often partition the search-space evenly: there are usually exponentially many ways to use the resources consumed by x_i. Additionally, disallowing the re-generation of the same specific solution x_i by the pricing problem is not possible with an IP-based pricing-problem without fundamentally changing the structure of the pricing-problem.

Consequently, effectiveness of a branching strategy must be evaluated in terms of how effectively the dual-price of the branching constraint can be integrated into the pricing problem. This is another motivation for our hybrid IP/Planning approach, as using a planning-based pricing problem allows us to disallow specific solutions and handle non-linear, time-dependent costs and constraints.

A concrete example of branch-and-price is presented in section 3.5.

Big-M Penalty Methods. We noted earlier that to start column generation an initial (linear) feasible solution is required. There is no guarantee that finding such a feasible solution is trivial. Indeed for classical Planning problems, finding

a feasible solution is PSPACE-complete in general. No work we are aware of determines the complexity of computing a linear relaxation of a plan.

We avoid this problem by transforming our IP into an MIP where for any constraint having $a_r < 0$ we add a new continuous variable $v_r \in [0, 1]$, representing the degree to which the constraint is violated, and replace the constraint with: $\sum u_{r,i} \cdot x_i \leq |a_r| \cdot v_r - |a_r|$ and $c_v = M$ where M is a large number. This guarantees the feasibility of the trivial solution $x_i = 0$ for all i and all $v_r = 1$.

This represents a relaxation of the original IP with the property that, given sufficiently large M, the optimal solution is a feasible solution to the original problem, iff such a solution exists. This is known as a "penalty method" or "soft constraint". The process is similar to the first phase of the simplex algorithm for finding an initial feasible solution to a linear program when all decision variables are known in advance.

3.3 The Bulk Freight Rail Scheduling Problem

In the Australian mining industry, it is common for bulk commodities such as coal and iron ore to be transported to ports on railways that are principally or exclusively used for that purpose. This is in contrast to Europe and the US where freight railways often share significant infrastructure with passenger trains. Additionally, bulk freight is nearly always routed as whole trains and usually only stockpiled at the mines and ports, avoiding many complex blocking and routing problems addressed in the literature.

The Bulk Freight Rail Scheduling Problem (BFRSP) is solved by bulk freight train operators and mining companies on a daily basis. The BFRSP can be an operational or tactical problem depending on the degree of vertical integration of the supply chain: where different above-rail operators share a track network they may need to prepare an accurate schedule to be negotiated with the track

network operator or other above-rail operators; alternatively schedules may be prepared "just-in-time" to assign crew.

The term "train" is highly overloaded in this domain, as such we will avoid using it. Instead we will use the words *consist* and *service*. These denote respectively: a connected set of locomotives and wagons; and a sequence of movements performed by a consist.

The above-rail operator has a set of partially-specified services collected from its clients. We refer to such partially-specified services as "orders". Each order is a sequence of locations that must be visited in order with time-windows (e.g. a mine, then a port, then the yard). Given this, schedulers must find a path for as many services as possible on the rail network such that no two services occupy the same "block" at the same time. A block is represented by a vertex in the track-network graph.

The objective is to first maximise the number of orders delivered, then to minimise the total duration of services. The number of segments that a single service must traverse varies with the path through the network. Services can dwell for an arbitrary time on some edges to allow others to pass.

More formally, a BFRSP $B = \langle G, \tau, c, O, T \rangle$ consists of: a track network graph $G = \langle V, E \rangle$ where each block is a vertex in V; a crosstime function $\tau(\langle v, v' \rangle)$ which represents the minimum time a consist may spend at the vertex v' after traversing the edge $\langle v, v' \rangle \in E$; a capacity function $c(v)$ which represents the maximum number of services which may occupy $v \in V$ concurrently; a set of orders O, each of which is a sequence of waypoints of the form $\langle v, t_{min}, t_{max} \rangle$, where $v \in V$ and t_{min} and t_{max} represent the earliest and latest times the waypoint may be visited; and a set of consists T.

The aim of the BFRSP is to satisfy as many orders as possible, with the minimum sum of service durations. An order o is satisfied by a service s if that service visits each waypoint within the specified time window. One service must satisfy

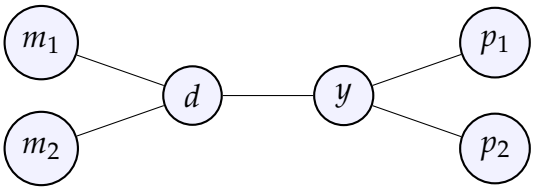


Figure 3.1: A simple track network

one order and be performed by one consist.

3.4 Modelling the BFRSP

We model the problem as a planning problem in *Golog*. The actions available to a consist c are: traverse an edge e with $\texttt{move}(c, e)$; wait for some integral number d of time points with $\texttt{wait}(c, d)$; and perform whatever action is required at a block b (e.g. loading at a mine or unloading at a port) that is a waypoint in some order o with $\texttt{visit}(c, o, b)$. We assume unit duration for these actions.

The following are effect axioms for the domain, which may be transformed into successor state axioms to address the frame problem in the standard way (Reiter, 2001). We assume the existence of some predicates to identify the first, last and next waypoints in an order: $\mathit{first}(o, b)$, $\mathit{last}(o, b)$ and $\mathit{next}(o, b, b')$

$$
\begin{aligned}
&\mathit{At}(c, b, \mathit{do}(\mathit{move}(c, \langle \_, b \rangle), \_)).\\
\neg &\mathit{At}(c, b, \mathit{do}(\mathit{move}(c, \langle \_, b' \rangle), \_)) \Leftarrow b \neq b'.\\
&\mathit{Started}(c, o, \mathit{do}(\mathit{visit}(c, o, b), \_)) \Leftarrow \mathit{first}(o, b).\\
&\mathit{Finished}(c, o, \mathit{do}(\mathit{visit}(c, o, b), \_)) \Leftarrow \mathit{last}(o, b).\\
&\mathit{Visited}(c, o, b, \mathit{do}(\mathit{visit}(c, o, b), \_)).
\end{aligned}
$$

We also need to keep track of allowable next visitable destinations for each

consist. A consist c is allowed to visit the first waypoint of an order that no consist has started, so long as c has not started any order; alternatively it may visit the next waypoint in the order it has already started.

$$
\begin{aligned}
&Dest(c, o, b, s_0) \Leftarrow first(o, b).\\
&Dest(c, o, b, do(visit(c, o', b'), s)) \Leftarrow first(o, b) \wedge last(o', b') \wedge \neg Visited(\_, o, b, s).\\
&Dest(c, o, b, do(visit(c, o, b'), s)) \Leftarrow next(o, b, b').\\
\neg &Dest(c, o, b, do(visit(\_, o, b), \_)).
\end{aligned}
$$

Finally, we assume the existence of a $time(s)$ fluent to define our time window checks.

$$InWindow(o, b, s) \Leftarrow \exists t, t'.\langle b, t, t' \rangle \in o \wedge t \leq time(s) \leq t'.$$

Obviously a consist cannot *move* on an edge that starts at a different node to the end of the last edge it traversed, and a consist can only visit a block that it is on. A waypoint (block) can only be visited once for an order and a consist cannot visit another order's waypoint until it has visited all the waypoints required for any order it has visited any waypoints of (i.e. consists cannot interleave visiting waypoints for different orders).

$$
\begin{aligned}
&Poss(move(c, \langle b, b' \rangle), s) \Leftarrow At(c, b, s) \wedge \langle b, b' \rangle \in E.\\
&Poss(visit(c, o, b), s) \Leftarrow Dest(c, o, b, s) \wedge InWindow(o, b, s).\\
&Poss(wait(c, n), s) \Leftarrow n \in \mathbb{Z}.
\end{aligned}
$$

Solving for a single service with a single waypoint is a case of finding a minimum-

cost path through G subject to the time-window constraints, where the edge cost is given by τ.

The *Golog* program to solve this problem is:

```
proc travelto(c, o, b, t0, t1):
  while ¬onblock(c, b) do
    π(e ∈ E).move(c, e);
  π(t ∈ [t0, t1].waituntil(t);
  visit(c, o, b)
```

To deliver a more complex service with more waypoints we simply `travelto` them in sequence:

```
proc deliverservice(c, o):
  foreach (b, t0, t1) in o do
    π(dt ∈ [0..t1 − time()]).wait(dt);
    travelto(c, o, b, t0, t1)
```

Up to this point, the problem is simple; existing implementations of *Golog* are able to find high-quality plans on simple networks without even considering action costs. Also most of the complex preconditions of the domain are irrelevant: it is impossible to interleave the satisfaction of different orders and no two trains can occupy the same block.

The *Poss* and *Conflict* clauses required to model these inter-order constraints are easier to model in terms of shared resources. For this reason we introduce $usage(action, resource, situation)$ and $availability(resource, situation)$ functions. These can be considered as additional clauses of the definition of *Conflict* and *Poss* in the

following way:

$$Conflict(as, s) \Leftarrow \exists r. \sum_{a \in as} usage(a, r, s) > availability(r, s)$$

$$\neg Poss(a, s) \Leftarrow \exists r. usage(a, r, s) > availability(r, s)$$

For the BFRSP domain we define these resources:

$$usage(enter(C, V), block(V, T), S) = 1 \Leftarrow time(T, S)$$

for block capacity constraints; and, in order to ensure that each order is delivered at most once:

$$usage(visit(C, V, O), waypoint(V, O), S) = 1$$

When there are a set of orders, but only one consist, the consist must still deliver the orders in some sequence, however there exist $n!$ sequences in which n orders may be satisfied. Any of these sequences is valid (as any service can be dropped), so we begin to tackle the combinatorial optimisation aspect of the BFRSP, in which the solver must pick an optimal sequence to satisfy orders.

```
proc deliverservices(c, O):
  π(o ∈ O).
    (deliverservice(c,o) | noop);
    deliverservices(c,O\{o})
```

With more than one consist, each consist must deliver some non-overlapping set of services. Choosing this set is an optimisation problem in its own right and

existing implementations of *Golog* will not find any solution.

```
proc main(C, O):
  forconc o in O do
    π(c ∈ C).deliverservice(c, o) | noop
```

This program has an important structural property that will be exploited by our algorithm: Given an assignment of orders, the joint execution is a set of fragments generated by

$$\pi(o \in O).\pi(c \in T).\texttt{deliverservice}(c,o) \mid noop$$

3.5 Fragment-Based Planning

Given the set F of all possible executions of the fragment program

$$\pi(c \in T).\texttt{deliverservice}(c,o)$$

all joint executions of `main` are a subset of F where no two fragments visit the same order or occupy the same track segment simultaneously.

Finding the optimal execution is then equivalent to finding the optimal solution to the Integer Program:

$$\begin{aligned}
&\text{Minimise:} && \sum_{f \in F} d_f \cdot x_f + \sum_{o \in O} M \cdot v_o && \\
&\text{Subject To:} && \sum_{f \in F_{bt}} x_f \leq c(b) && \forall b \in B\,, \forall t \in T \\
& && v_o + \sum_{f \in F_o} x_f = 1 && \forall o \in O
\end{aligned}$$

Here x_f is 1 iff f should be executed in the joint plan. $F_o \subseteq F$ is the set of fragments that satisfy order o, $F_{bt} \subseteq F$ is the set of fragments that are on block b at time t and d_f is the duration of fragment f.

Enumerating F is however prohibitive, and large numbers of potential fragments are uninteresting, or prohibitively costly and will never be chosen in any reasonable joint-execution, nor need to be considered in finding and proving the optimal joint execution.

To avoid enumerating F, we can use delayed column generation as described earlier. To use this approach, we need to re-compute action costs that minimise the reduced cost of the next fragment generated given an optimal solution to the restricted LP. Given dual prices $\lambda_{b,t}$ for block/time resources and $\lambda_{b,o}$ for waypoints, our fragment planner's action costs become $\lambda_{b,t}$ for $move(c, e)$, actions executed at time t, given $e \in b$; and $\lambda_{b,o}$ for $\texttt{visit}(c, o, b)$ actions.

We provide pseudo-code for the FBP algorithm below, Quine quotes are used around linear expressions such as $[\![expression \leq constant]\!]$ to denote constraints given to the LP solver to distinguish them from logical expressions.

To solve the linear relaxation of the joint planning problem, we call

$$\text{LINFBP}(\{v_o \mid o \in O\}, \{o : [\![v_o = 1]\!] \mid o \in O\}, O, \delta)$$

Note that the $[\![expr = a]\!]$ form of constraints can be considered a shorthand for two constraints $[\![expr \leq a]\!]$ and $[\![-expr \leq -a]\!]$.

We assume that the *Golog* search procedure *Do* returns the fragment f with the least reduced cost $\gamma(f, \tilde{\boldsymbol{\lambda}})$, rather than just any valid execution. Our implementation uses uniform-cost search to achieve this.

function LINFBP(*Frags*, *Res*, *Goals*, δ)

LowBound $\leftarrow 0$

UpBound $\leftarrow M \cdot \|Goals\|$

while $(1-\epsilon)\cdot UpBound > LowBound$ **do**
$\theta, \tilde{\boldsymbol{x}}, \tilde{\boldsymbol{\lambda}} \leftarrow SolveLP(Frags, Res)$
$F \leftarrow Do(\pi(g \in Goals).\delta|noop, \tilde{\boldsymbol{\lambda}})$
$Frags \leftarrow Frags \cup \{F\}$
$d\theta \leftarrow \|Goals\| \cdot \gamma(F, \tilde{\boldsymbol{\lambda}})$
$UpBound \leftarrow \theta$
$LowBound \leftarrow max(LowBound, \theta + d\theta)$
for all $r \in F$ **do**
$[\![e \leq a]\!] \leftarrow Res[r]$
$Res[r] \leftarrow [\![e + u_{F,r} \cdot x_F \leq a]\!]$
return $\theta, Frags, Res, \tilde{\boldsymbol{x}}$

We then use the LINFBP column generation implementation inside a branch-and-price search. We assume that fragments use redundant resources for the purposes of branching. In particular we rely on each fragment to use 1 unit of a resource that uniquely identifies that fragment, so that we eventually find an integral solution if one exists.

function FBP(Gs, δ)
$Fs \leftarrow \{v_g \mid g \in Gs\}$
$Rs \leftarrow \{g : [[1 \cdot v_g = 1]] \mid g \in Goals\}$
$LowBound \leftarrow 0$
$UpBound \leftarrow M \cdot \|Goals\|$
$Queue \leftarrow \{\emptyset\}$
$Fs \leftarrow Fs \cup initfrags(Gs, \delta)$
while $(1-\epsilon)\cdot UpBound > LowBound$ **do**
$BranchRs \leftarrow Pop(Queue)$
$LRs \leftarrow Rs \cup BranchRs$
$\theta, Fs, Rs, \tilde{\boldsymbol{x}} \leftarrow$ LINFBP(Fs, LRs, Gs, δ)
if any resources have fractional usage **then**

Iteration		$\tilde{\boldsymbol{\pi}}$	Fragments
1.	$\theta = 200$ $\gamma(f_1) = -91$ $\gamma(f_2) = -91$		f_1, f_2
2.	$\theta = 109$ $\gamma(f_3) = -90$ $\gamma(f_4) = -90$	$\pi_{d,0} = -91$	f_3, f_4
3.	$\theta = 19$	$\pi_{d,3} = -1$ $\pi_{p_1,o_1} = -90$ $\pi_{p_2,o_2} = -90$	

Table 3.1: LINFBP iterations. $M = 100$

if soft constraints satisfied i.e. $\theta < M$ **then**

$X \leftarrow$ some fractional resource

Branches $\leftarrow$ branch on $\lceil X \rceil$ and $\lfloor X \rfloor$

Queue $\leftarrow$ *Queue* $\cup$ *Branches*

UpBound $\leftarrow$ *SolveIP*(Fs, LRs)

LowBound $\leftarrow$ minimum θ in *Queue*

This process can be modified to incrementally return each solution to *SolveIP*(Fs, Rs) as it is computed, and make this an effective anytime algorithm.

We illustrate the iterations of the LINFBP algorithm in Table 3.1 using a goal non-satisfaction penalty (Big-M) of 100. We solve a BFRSP on the network in figure 3.1 with two orders:

$$o_1 = [\langle m_1, 0, 20\rangle, \langle p_1, 0, 20\rangle, \langle y, 0, 20\rangle]$$

and

$$o_2 = [\langle m_2, 0, 20\rangle, \langle p_2, 0, 20\rangle, \langle y, 0, 20\rangle]$$

We omit the first consist argument to all actions and the duals of the goal satisfaction constraints for brevity.

Initially, the dual vector has no non-zero elements other than the goal non-

satisfaction constraints. The lowest reduced-cost plans for the two sub-goals are then simply the shortest paths from y to one of the mines m_1 or m_2, then one of the ports p_1 or p_2, then to the yard. That is,

$$f_1 = \begin{bmatrix} move(\langle y, d\rangle); move(\langle d, m_1\rangle); visit(m_1, o_1); \\ move(\langle m_1, d\rangle); move(\langle d, y\rangle); move(\langle y, p_1\rangle); \\ visit(p_1, o_1); move(\langle p_1, y\rangle); visit(y, o_1) \end{bmatrix}$$

$$f_2 = \begin{bmatrix} move(\langle y, d\rangle); move(\langle d, m_2\rangle); visit(m_2, o_2); \\ move(\langle m_2, d\rangle); move(\langle d, y\rangle); move(\langle y, p_2\rangle); \\ visit(p_2, o_2); move(\langle p_2, y\rangle); visit(y, o_2) \end{bmatrix}$$

Given these two new columns, the dual vector detects one of the bottlenecks at block d at time 0, overestimating that over-use of this resource costs 91 units. Using this information the new minimum cost way to achieve each goal now avoids this shared resource by initially waiting, generating fragments $f_3 = [wait(1)] +\!+ f_1$ and $f_4 = [wait(1)] +\!+ f_2$.

With these two additional fragments, the dual vector shows us that the unloading *visit* actions at p_1 and p_2 are a bottleneck (as we now have two ways to satisfy each goal). Obviously these bottlenecks cannot be avoided, however they have a cost of only 90, so if there is a fragment with cost < 10 which also does not use block d at time 3, there could exist a fragment with negative reduced cost. The search shows that no such fragment exists, and this proves (linear) optimality.

Within the FBP algorithm we then solve the IP with the 4 fragments f_1 to f_4, we find one of the primal integral solutions $x_1 = 1$ and $x_3 = 1$ and all other variables 0. Since this solution has the same objective of 19 as the linear relaxation, we have proved this solution optimal.

3.6 Fragment-Based Planning in other domains

The resources we use in the BFRSP are only consumed by actions, and have a finite, non-renewable availability. Additionally, fragments can only have negative interactions between one another. The choice of one fragment can only prevent the choice of another. However we would like to be able to choose fragments such that one may satisfy an open precondition of another.

We handle this case in a similar way to goal-satisfaction constraints by generalising our resources so that they can be generated in addition to being consumed. While we did not call our goal-satisfaction constraints resources, it can be considered that the fragments in the BFRSP generated 1 unit (i.e. consumed -1 units) of a "goal-satisfied" resource, which had a requirement of 1 unit (i.e. an availability of -1 unit).

If we consider the case where a set of fragments F_{pre} satisfies a precondition of F_{post}, then a constraint of the form

$$\sum F_{post} - M \cdot \sum F_{pre} \leq 0$$

is sufficient to guarantee that fragments from F_{post} can only be chosen if the precondition has been satisfied. If F_{post} fragments immediately invalidate this precondition then a constraint of the form

$$\sum F_{post} - \sum F_{pre} \leq 0$$

is more appropriate, and forces at most a single f_{post} to be chosen per time the precondition is satisfied. In either case, as most resources (including preconditions) will be time-indexed (like the block constraints in BFRSP) in many domains there will likely be constraints forcing $\sum F_{post} \leq 1$.

Which form to choose is an exercise for the modeller, and an algorithm for turning a basic action theory into resource-based constraints is beyond the scope

of this chapter.

Both of the forms fall into our existing resource framework, and can be handled without modification to the FBP procedure. However the *Poss* fluent now needs to be relaxed to allow fragments to assume that a precondition has been satisfied by another fragment, at a cost derived from the π_{pre} dual variables.

If we know each action's resource consumption from the *usage* function described above, and the usage is independent of the actions performed in other fragments, then the usage of a resource r by a fragment f is the sum over all actions a performed in that fragment of $usage(a, time(f), f)$.

Given this, a *Golog* program of the form `forallconc` x `in` G `do` δ, can generate an equivalent FBP model:

$$\begin{aligned}
\text{Minimise:} \quad & \sum_{f \in F} c_f \cdot x_f + \sum_{g \in G} M \cdot v_g & \\
\text{Subject To:} \quad & \sum_{f \in F} u_{fr} \cdot x_f \leq a_r & \forall r \in R \\
& v_g + \sum_{f \in F_i} x_f = 1 & \forall g \in G
\end{aligned}$$

F_g is the set of all legal fragments generated by the *Golog* program $\delta_{[x/g]}$. F is the union for all $g \in G$ of F_g . R is the set of all resources used by any fragment in F. c_f is the cost of executing the actions contained in fragment f. $x_f \in \{0, 1\}$ is 1 iff f should be executed in the joint plan. u_{fr} is the net usage of resource r by fragment f. a_r is the total availability of resource r. λ_r will denote the optimal dual prices of each resource r in the solution of this problem.

We refer to the set G as the "fragmentation dimension" and δ as the "fragment program".

If these resources totally describe the potential interactions between the different sub-goals $\delta_{[x/g]}$, then the optimal solution to this FBP model is the optimal joint execution.

We present results in the next section for a multi-agent variant of blocks-world. We use the set of blocks as the fragmentation dimension, and pseudo-code for the fragment program in this example is:

```
proc handleblock(hand, block):
  wait(1)*;
  (pickup(hand, block); drop(hand, table)|noop);
  wait(1)*;
  (pickup(hand, block); drop(hand, dest(block))|noop);
  On(block, dest(block))?
```

Given this fragment program, we can see that other fragments will be required to satisfy the *Clear* fluent required by `pickup` and `drop`. Consequently $Clear_{b,t}$ becomes a generatable resource for each block at each time point, with initial availability of 1 if both $t = 0$ and b is initially clear; and 0 otherwise.

Performing the action `pickup`(h, b) at time t consumes one unit of $Clear_{b,t}$ and generates one unit of $Clear_{b',t+1}$, assuming b was on b'.

Similarly, $Occupied_{h,t}$ is a (non-generatable) resource with availability 1 which is consumed by picking up a block with hand h or waiting when h is holding a block.

The actions we have described thus far produce *Clear* resources at specific time points, however the clearness of a block persists if it is not picked up and nothing is put down on it. To handle this we introduce an explicit zero-duration `persist`(r) action, which consumes one unit of r_t and generates one unit of $r_{time()}$ for some non-deterministically chosen $t < time()$. These `persist`$(Clear_b)$ actions are inserted immediately before the pickup and drop actions in `handleblock` above.

We discuss some of the necessary and sufficient conditions for the correctness

of this approach in chapter 4.

3.7 Experiments

We compare our FBP implementation in these two domains with: MIndiGolog (Kelly and Pearce, 2006), a temporal Golog interpreter; POPF2 (Coles, Coles, et al., 2010) and YAHSP2 (Vidal, 2011) two heuristic satisficing planners; CPT4 (Vidal and Geffner, 2006), a temporal planner based on constraint programming; and CPX, a constraint programming solver using Lazy Clause Generation (Ohrimenko, Stuckey, and Codish, 2009). All experiments were performed on a 2.4 GHz Intel Core i3 with 4GB RAM running Ubuntu 12.04. Our implementation used Gurobi 5.1, CPython 2.7.3. We used the binary version of `cpx` included with MiniZinc 1.6. All of the temporal planners were the versions used in the 2011 IPC.

We present results from a simple variant of the BFRSP with a fixed number of waypoints per order: a mine, a port and then the train yard. Results in Tables 3.2 and 3.3 represent results on the track network depicted in Figure 3.1 and a high level network of an Australian mining company. We assume the capacity at all nodes except y is 1, and the y has unbounded capacity, and there are as many consists as orders. The "m/p" column is the number of mines each port makes orders for. All models of the problem were given symmetry-breaking constraints regarding assignment of consists to orders. We report "time to first solution" in these experiments, meaning the time taken to generate a schedule that delivered all orders.

Table 3.2 shows that the FBP approach scales to problems more than an order of magnitude larger than either a MiniZinc model based on scheduling constraints solved with the lazy clause generation solver `cpx`, or a temporal PDDL model solved with `popf2`. The constraint-programming technique of Lazy Clause

Generation is state-of-the-art for many scheduling problems, and `cpx` is competitive with our approach until memory limits were reached.

There is a significant overhead of the FBP algorithm in the smaller models, this is likely explained by our pure python implementation when compared with the highly engineered, compiled implementations of both `popf2` and `cpx`. Unscientifically, interpreted python can expect a 30× to 100× slowdown compared to an equivalent algorithm in C or C++.

The temporal planner `popf2` was chosen because it was the best performing of the temporal satisficing planners in IPC 2011 with support for numeric fluents or initial timed literals one of which is required to model time-windows.

`popf2` performs very poorly when there is more than one order to any mine or port. We presume this is because standard planning heuristics cannot effectively detect the bottleneck in the track network at block d in Figure 3.1.

To test this theory, we relax the complicating time-window constraints, which allows us to test other temporal planners on this domain. Table 3.3 shows that, as contention for block d increases, solution times in the heuristic search planners increase very sharply. This leads to the somewhat surprising result that a decomposition approach outperforms heuristic search as interaction increases. We also outperform the constraint programming approach used in `cpt4`, note that this is an optimal planner rather than the anytime algorithms implemented in `yahsp2` and `popf2`.

This result is not seen in the blocks-world domain where both enforced hill climbing and WA* find solutions very quickly. However we still see significant speedups in proof of optimality versus both `popf2` and `yahsp2` even in this quite sequential domain. It should be noted that neither are optimal planners, but as both are complete anytime algorithms, like FBP, we believe this is still a meaningful comparison. FBP also proves optimality faster than the optimal temporal planner `cpt` on the largest instance.

This is obviously not a totally fair comparison, as the *Golog* model can encode more domain knowledge, however we believe this is a desirable property in a modelling language, and closer to a real-world optimisation setting, where modellers can and will provide as much useful information to any solver they use as is feasible. Additionally, existing *Golog* implementations do not perform well on either of these problems, and PDDL planners are among the most well studied alternatives and provide the fairest comparison that can be considered state-of-the-art.

$\|V\|$	$\|O\|$	m/p	*Golog*	popf2	cpx	FBP
6	2	1	0.4	**0.3**	0.7	1.6
6	4	1	—	—	**2.3**	3.3
6	4	2	—	—	**1.7**	2.0
6	8	2	—	—	**7.5**	7.6
6	16	2	—	—	37.9	**29.7**
6	32	2	—	—	—	**50.3**
6	64	2	—	—	—	**150.3**
6	128	2	—	—	—	**418.8**
6	256	2	—	—	—	**589.7**
16	44	11	—	—	—	**315.0**
16	88	11	—	—	—	**363.0**

Table 3.2: Time to first solution for increasing track network vertices, $|V|$, and orders, $|O|$, in seconds (1800s time limit, 4GB memory)

$\|V\|$	$\|O\|$	m/p	popf2	yahsp2	cpt	FBP*
6	2	1	0.3	**0.0**	**0.0**	1.6
6	4	1	8.4	**0.0**	0.8	3.3
6	4	2	—	**0.0**	0.7	2.0
6	8	2	—	**0.3**	—	7.6
6	16	2	—	42.5	—	**29.7**
6	32	2	—	—	—	**50.3**

Table 3.3: BFRSP without time windows: Time to first solution in seconds (time limit 1800s, 4GB memory, FBP results are from the non-relaxed problem)

Blocks	popf		yahsp2		cpt		FBP	
3	0.0	3.5	0.0	0.0	0.0	0.0	0.3	0.5
4	0.0	53.0	0.0	0.0	0.0	0.0	0.3	0.7
5	0.0	—	0.0	0.1	0.0	0.1	0.2	0.6
6	0.1	—	0.0	6.4	0.1	0.1	0.5	0.8
7	0.9	—	0.1	—	0.2	0.2	0.6	0.8
8	0.1	—	0.0	—	0.4	0.4	1.0	1.9
9	17.9	—	0.0	—	1.7	1.7	5.0	6.2
10	0.3	—	0.0	—	1.7	1.7	1.3	2.0
11	1.4	—	0.0	—	5.2	5.2	2.0	3.8
12	—	—	0.0	—	12.6	12.6	4.9	6.8

Table 3.4: Blocks-world: Time to first / optimal solutions in seconds (time limit 120s, 4GB memory)

3.8 Conclusions and Further Work

We see from our experimental results that our Fragment-Based Planning approach scales to an important class of industrial problems while sacrificing little of the flexibility of the underlying planning formalism, and *Golog* language.

Our approach fares significantly better on the BFRSP than on blocks world. We believe this is because this domain combines the strength of the two core technologies of state-based search and MIP: the MIP detects global, largely sequence-independent bottlenecks and guides the overall search; and state-based search handles the sequential aspects of shortest path finding that might otherwise require a large number of variables to encode as an LP. Whereas, in blocks world, the LP detects the desired sequence of block handling, and the search solves the relatively trivial problem of how much waiting is required to ensure the blocks are picked up and put down at the optimal time. We suspect that this is a bad model for FBP, as it splits the work unevenly between master and sub-problems, and relies on the LP to detect impossible sequences, something that even uninformed search should be better at. More work is required to explore what fluents are best to relax into linear constraints; what forms those constraints should take;

the choice of fragmentation dimension; and the fragment program. Nonetheless, this approach has proven effective in this sequential domain.

Our implementation lacks a number of significant engineering improvements that could be applied, most significantly: better MIP heuristics; and better branching strategies. We use a simple variant of the general diving MIP heuristics described in (Joncour et al., 2010), though problem specific MIP heuristics could yield better performance, such as (Jampani and Mason, 2008).

Additionally we use very simplistic branching rules: we branch only on individual fragments being included, rather than e.g. pairs of resources as in Ryan-Foster branching (Ryan and Foster, 1981). This leads to a very unbalanced branch-and-price tree and can lead to exponentially increased runtimes if the problem is not proven optimal at the root. Note that while the FBP algorithm we describe allows Ryan-Foster branching, our implementation does not branch on such "good" redundant resources.

Cost-optimal planning in *Golog* is required to solve the pricing problem in FBP. This is not a problem considered in the existing literature, and FBP might benefit from analogues to classical planning heuristics. Alternatively, as the pricing problem has a cost bound, bounded cost search algorithms for *Golog* might be beneficial. This is still an open area of research even for *STRIPS* planning (Haslum, 2013).

Existing fast but incomplete algorithms such as enforced hill climbing or causal chains (Lipovetzky and Geffner, 2009) might compute a better starting set of fragments and/or inspire FBP-specific MIP heuristics.

Many of these enhancements are well studied and implemented in *STRIPS* planners, and only the *Do* call in the FBP algorithm is *Golog*-specific. If similar explicit relaxations and fragment goals can be provided, this could be replaced by a call to an optimal planner, with additional `assume-[`*`fact`*`](?time, ...)` actions for each fact with the action cost derived from the dual prices.

The effect of these time-dependant costs on the performance of *STRIPS* planners would also need to be investigated. Few resources will normally have non-zero dual prices, which could help limit the branching factor.

How to select the fragment goals is less obvious. Planning with preferences could be used to choose some subset of goals to achieve, however it is not obvious how to ensure this planning problem is easier than the original.

Additionally, for FBP to work directly with PDDL work is required to automatically identify: a "fragmentation dimension"; "fragment generator"; and the fluents to relax.

Chapter 4
Optimisation and Relaxation in the Situation Calculus

In the previous chapter we introduced Fragment-Based Planning including some example domains where it can be applied. In this chapter we slightly generalise that algorithm and explore the necessary and sufficient conditions for agents to cooperate through the FragPlan framework.[1]

4.1 Introduction

Reasoning about quality is essential in many domains, as agents must often make economical use of one or more resources, be it money, fuel, time, or some other domain-specific resource.

We consider multiagent planning problems expressed in *Golog*, an agent language based on the situation calculus. *Golog* is Turing complete—the use of a *Golog* interpreter comes without any guarantee of termination. We can, however, identify a restricted class of problems (or associated *Golog* programs) that have a "bounded benefit" property. We show that budget-limited planning is decidable for this class. We also introduce relaxations that let us reason about how far from optimal a candidate solution is. Relaxations can be viewed as specialised

[1]The research presented in this chapter was published in Toby O. Davies, Adrian R. Pearce, Peter J. Stuckey, and Harald Søndergaard (2015). "Optimisation and Relaxation in the Situation Calculus". In: *Autonomous Agents and Multiagent Systems (AAMAS 15)*, pp. 1141–1149

algorithms to prove statements of the form "no solution is more than a multiple of ϵ better than solution x" without the need to enumerate the solution space. Technically our approach is close to the use of Lagrangian relaxation techniques in operations research.

Delete relaxation has previously been applied to *Golog*, to generate heuristically improved execution (Blom and Pearce, 2010). In contrast with that work, we focus on *precondition relaxation*, as this is particularly appropriate for multi-agent teams where each agent ideally helps others achieve the preconditions they face. The kind of relaxation we have in mind is much more sophisticated than an "ignore precondition" relaxation. Our precondition relaxations can avoid combinatorial explosions by ignoring the multiple ways concurrent action sequences can be interleaved. To guide search for optimal solutions we apply costs to assuming, and bonuses to causing, the relaxed preconditions. Thus preconditions are treated as a type of shared resource that can be traded between agents at a cost. This is key to the performance improvements we achieve.

Utilisation of shared resources has previously been treated in the situation calculus, but in the context of an explicit interleaved semantics of concurrency (De Giacomo, Lespérance, and Levesque, 2000). Our approach can frequently minimise—sometimes even ignore—explicit inter-agent action interleaving, heuristically finding high-quality joint executions directly from high-level specifications.

Contribution The relax-&-merge algorithm described in this chapter generalises the fragment-based planning approach of chapter 3 which shows orders of magnitude improvements over state-of-the-art temporal planners. We define the necessary conditions for modelling domains in this generalised formalism. We propose a new kind of relaxation for planning problems; "precondition relaxations" which adapt Lagrangian relaxation to dynamic logic-based optimisation problems. The technique is of particular interest in a multi-agent setting, because it

offers a blackbox approach to planning via collaborative search: An agent is not required to know about other agents' programs or effect axioms for "private" fluents; all that is required is the ability to query those agents for their optimal plans under a given penalty function. We show that well-chosen relaxations can provide large increases in the speed of planning, scaling linearly with the number of interacting agents. Our work allows optimisation using a wider range of search techniques in the situation calculus than previously possible.

Outline We recapitulate *Golog* and relaxation broadly, in sections 4.2 and 4.3, respectively. section 4.4 introduces bounded-benefit programs and section 4.5 introduces precondition relaxation. section 4.6 shows how to combine individual agents' relaxed plans and section 4.7 shows how to use the result to construct a feasible joint execution.

4.2 Preliminaries

We assume familiarity with the situation calculus and reasoning based on regression, at the level of Reiter (Reiter, 2001), from which we also (mostly) borrow notation and terminology. We use $\mathcal{R}$ for the regression operator; $\preceq$ for the prehistory relation; Φ_f as the regressable successor-state axiom for a fluent f; and ϕ_f^+ and ϕ_f^- for the positive and negative effect axioms of a fluent f respectively.

We use a fragment of *ConGolog* (De Giacomo, Lespérance, and Levesque, 2000), which includes most constructs of the language, except for (recursive) procedures. Hereafter we will simply refer to *Golog*, *ConGolog* and its extensions simply as *Golog*:

α	atomic action
$\varphi?$	test for a condition
$\delta_1; \delta_2$	sequence

if φ **then** δ_1 **else** δ_2	conditional
while φ **do** δ	while loop
$\delta_1 \vert \delta_2$	non-deterministic branch
$\pi x.\delta$	non-deterministic choice of argument
δ^*	non-deterministic iteration
$\delta_1 \Vert \delta_2$	concurrency

In the above, α is an action term, possibly with parameters, and φ is a situation-suppressed formula, that is, a formula in the language with all situation arguments in fluents suppressed. We denote by $\varphi[s]$ the situation calculus formula obtained from φ by restoring the situation argument s into all fluents in φ.

Program $\delta_1 \vert \delta_2$ allows for the non-deterministic choice between programs δ_1 and δ_2, while $\pi x.\delta$ executes program δ for *some* non-deterministic choice of a legal binding for variable x. δ^* performs δ zero or more times. Program $\delta_1 \Vert \delta_2$ expresses the concurrent execution (interpreted as interleaving) of programs δ_1 and δ_2. We assume without loss of generality that each occurrence of the construct $\pi x.\delta$ in a program uses a unique fresh variable x.

Formally, the semantics of *Golog* is specified in terms of single-step transitions, using the following two predicates (De Giacomo, Lespérance, and Levesque, 2000): *(i)* $Trans(\delta, s, \delta', s')$, which holds if one step of program δ in situation s may lead to situation s' with δ' remaining to be executed; and *(ii)* $Final(\delta, s)$, which holds if program δ may legally terminate in situation s.

The definitions of *Trans* and *Final* we use are as in (Sardina and De Giacomo, 2009); these are standard (De Giacomo, Lespérance, and Levesque, 2000), except that, following (Classen and Lakemeyer, 2008), the test construct $\varphi?$ does not yield any transition, but is final when satisfied. Thus, it is a *synchronous* version of the original test construct (it does not allow interleaving). Note that the definition of $Trans(\delta, s, \delta', s')$ has only one successful non-recursive case, where s' is exactly one action longer than s; any successful transition adds exactly one action to the

current situation.

Trans and *Final* are used to define the single-step semantics of *Golog*; they are used to look ahead, to solve the planning problem. We define $Trans^*$ to be the transitive reflexive closure[2] of *Trans*. $Trans^*$ is used to define the reachable states: we wish to limit the search to states that are both reachable, and for which any residual program is *Final*. For this purpose we define

$$Do(\delta, s, s') \equiv \exists \delta'. Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s')$$

4.3 Reasoning about Optimality

A key technique used in operations research to prove solution quality is *relaxation*. Given a problem with a set X of solutions and cost function to be minimised C, a relaxation is a new problem with solutions X' and cost function C' such that $X \subseteq X'$ and for any solution $x \in X$, the relaxed cost is a lower bound on the real cost, that is, $C'(x) \leq C(x)$.

A useful relaxation is one that is easier to solve to optimality, compared to the original problem, and which additionally provides a tractable way to establish how far from optimal a candidate solution is. Let x be a feasible solution to the original problem and let x'^* be an optimal solution to the relaxed problem. The optimality gap is then defined as $\eta = C(x)/C'(x'^*) - 1$. When the gap is 0, the solution x is optimal. Where multiple relaxations are available, the tightest can be used to compute η. This approach is used with Lagrangian relaxation of integer programs.

In many optimisation problems it is easier to optimise a variant of the problem with fewer constraints. For example the resource constrained shortest path problem (Mehlhorn and Ziegelmann, 2000) is NP-complete, whereas there are

[2] *Trans* is a binary relation but we follow convention, writing $Trans(\delta, s, \delta', s')$ rather than $Trans((\delta, s), (\delta', s'))$.

well-known polynomial-time algorithms, such as Dijkstra's, for the classical, unconstrained shortest path problem.

Lagrangian relaxation softens complicating constraints and incorporates them into the objective function (Lemaréchal, 2001; Fisher, 2004). The idea is to capitalise on algorithms designed for the easier problem while penalising violations of the complicating constraints. Careful variation of the penalties (the so-called Lagrange multipliers) allows solutions to the relaxed problem to be guided towards feasible areas of the original problem.

Traditionally Lagrangian relaxation is applied to Integer Programming (IP) models. A major contribution of this chapter is the extension of Lagrangian relaxation to logic-based optimisation in dynamical systems.

Consider a set R of inequalities to relax in an IP model:

$$\begin{array}{lll} \text{Minimise:} & z(\tilde{\boldsymbol{x}}) & \\ \text{Subject To:} & c_r(\tilde{\boldsymbol{x}}) \leq 0 & \forall r \in R \\ & c_n(\tilde{\boldsymbol{x}}) \leq 0 & \forall n \in N \\ & \tilde{\boldsymbol{x}} \in \mathcal{Z}^n & \end{array}$$

We transform it into a relaxed problem:

$$\begin{array}{lll} \text{Minimise:} & z(\tilde{\boldsymbol{x}}) + \sum_{r \in R} \lambda_r \cdot c_r(\tilde{\boldsymbol{x}}) & \\ \text{Subject To:} & c_n(\tilde{\boldsymbol{x}}) \leq 0 & \forall n \in N \\ & \tilde{\boldsymbol{x}} \in \mathcal{Z}^n & \end{array}$$

Note that the effect is to increase the objective when constraints are violated, and decrease it when constraints are strictly satisfied. To solve the original problem, the relaxed problem is optimised and for each violated constraint r, the penalty λ_r is increased. The relaxed problem is then re-optimised, and so the process

is iterated. In general, branching is also required to find solutions to discrete problems.

4.4 Cost-Aware Search

Consider the problem of finding a path for agent a from $\langle 0, 0\rangle$ to $\langle x, y\rangle$ on an infinite Manhattan grid. The agent is allowed to move in each of the cardinal compass directions N, S, E, and W.

A naive *Golog* program to solve this problem is:

```
proc travelto(a, x, y):
  while ¬At(a,x,y) do
    π(d ∈ [N,S,E,W]).move(a,d);
```

A satisfying solution to this problem is uninteresting as there are infinitely many and an optimal path is trivial to compute. However a simple depth first search for solutions to this program might not terminate. To guarantee termination, a significantly more complex and less flexible program may restrict the search to only move towards $\langle x, y\rangle$. Such a program will find an optimal path between two points, but the approach will fail in general, if any obstacles are introduced into the grid.

We introduce a restricted class of *Golog* program and cost functions that allow the simpler and more general *Golog* program to always terminate. Algorithms from classical planning then allow us to compute the optimal solution. We refer to the restrained programs as "bounded-benefit".

Definition 4.1 (Bounded Benefit)**.** *A bounded-benefit program is a Golog program δ, for which there exists some plan length l_0 and $\epsilon > 0$ such that some lower bound $lb(l)$ on the cost of any reachable situation of length l satisfies $\forall n \in \mathbb{N} : lb(l_0 + n) \geq lb(l_0) + n\epsilon$,*

and $lb(l_0)$ is finite.

That is, there are a finite number of beneficial actions that can be performed, and all other actions increase cost by at least ϵ. Note that this is a property of the program and cost function, and may not depend on the precondition axioms of the domain.

Bounded benefit programs are related to the problems addressed by classical planning, but are slightly more general. In particular, planning algorithms assume monotonically increasing costs where $l_0 = 0$ and $lb(0) = 0$. Bounded benefit programs can be converted into this form when $lb(l_0)$ is known (rather than merely guaranteed to exist) by defining a new cost function similar to removal of soft-goals in classical planning (Keyder and Geffner, 2009). Optimal algorithms for this class of problems are well studied in the automated planning literature, and include both "uninformed" search, such as uniform cost search, and "informed" search such as A^*, which is reliant on a heuristic or relaxation.

The simplest class of bounded-benefit programs results when all actions increase cost. In the Manhattan grid example, if all actions have unit cost, every program has bounded-benefit. A less obvious, but interesting class of bounded-benefit programs results when the cost function defines a finite set of "soft goals" that decrease the objective when achieved. For example, in the Manhattan grid example, some grid points may be points of interest for which a reward is available as they are visited the first time (similar to the prize-collecting travelling salesman problem (Balas, 1989)). We use this kind of soft-goal in later sections to guide agents to achieve preconditions for others.

We have introduced bounded-benefit programs in response to the undecidability of planning in *Golog*. In aid of our proof of decidability we introduce the following definitions:

Definition 4.2 (Agent State)**.** *An agent state is a pair $\langle s, \delta \rangle$ comprising a situation s and a residual program δ.*

Definition 4.3 (Reachable State)**.** *An agent state $\langle s', \delta' \rangle$ is reachable from $\langle s, \delta \rangle$ if it satisfies $Trans^*(\delta, s, \delta', s')$.*

Definition 4.4 (Applicable Transitions)**.** *The set of applicable transitions $T(s, \delta)$ is the set of agent states reachable by using exactly one action from $\langle s, \delta \rangle$*

$$T(s, \delta) = \{\langle s', \delta' \rangle \mid Trans(\delta, s, \delta', s')\}$$

Definition 4.5 (Branching Factor)**.** *A Golog program δ in some domain $\mathcal{D}$ has branching factor $b = max(|T(s', \delta')|)$ over all reachable states $\langle s', \delta' \rangle$ of finite length.*

To have a finite branching factor merely requires that the set of possible actions in any given situation is bounded. This is not an unreasonable restriction, and would be required for a Prolog-based *Golog* implementation.

Theorem 4.1. *Let $m \in \mathbb{N}$ and let δ be a bounded-benefit program with finite branching. The query $\mathcal{D} \models Do(\delta, S_0, s) \wedge (\forall s' : s' \preceq s \Rightarrow cost(s') \leq m)$ is decidable.*

Proof. For any finite length l, the situations reachable from S_0 in at most l steps can be enumerated (there are at most b^l solutions), for example, by a breadth-first search.

By Definition 4.1 we can assume the existence of a length l_0 beyond which cost grows linearly, at least. The length of a reachable situation s with $cost(s) \leq m$ is then bounded by $l_0 + (m - lb(l_0))/\epsilon$. □

More practically, we can perform any search algorithm with an additional test for the condition $cost(s) \leq m$ when nodes are expanded, and guarantee that either we exhaust the successors of s, or the cost of the successors eventually exceeds the budget m, by defining a budget-limited *Trans*:

$$Trans_m(\delta, s, \delta', s') \equiv Trans(\delta, s, \delta', s') \wedge Cost(s') \leq m.$$

Importantly this approach does not require a computable lower bound, merely the guarantee that one exists.

4.5 Relaxing Preconditions

Relaxing delete effects (by ignoring them) is common in automated planning. The result is "easier" than the original problem because the application of an action without delete effects monotonically increases the set of true facts. Ignoring preconditions is the regression analogue: the regression of a goal formula through an action with no preconditions monotonically decreases the set of open preconditions, guaranteeing the regressed formula will eventually become true in the initial situation, if any such action sequence exists.

Instead of simply ignoring preconditions, we attach costs to assumptions. In a collaborative setting, an agent has a choice between the pursuit of achieving its own precondition, or relying on others to do this. Whichever is easier depends on circumstances but the choice can be informed by using cost to express a benefit to achieving another agent's preconditions.

For a very simple example, consider Figure 4.1 which depicts an instance of a "cooperative navigation" problem. (We will vary the initial conditions throughout this chapter, but Figure 4.1's graph will remain unchanged in subsequent examples.) The dashed edge denotes an edge that is in a "locked" state; it can be unlocked only by certain agents, and only if they are at appropriate locations. For now assume there are two agents, a and b, with the joint goal for a to reach z. The edge from y to z can only be unlocked by b if it is at y. We assume unit cost to unlock an edge and to traverse any edge. Note that the shortest path for one agent to travel from x to z depends on other agents enabling it.

Primitive actions in this domain are $move(Agt, From, To)$ and $\texttt{unlock}(Agt, From, To)$. Fluent $At(Agt) = Loc$ gives an agent's current location; $Unlocked(From, To)$ rep-

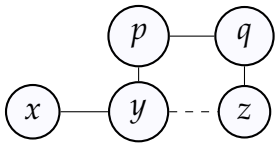


Figure 4.1: Cooperative navigation with unlockable edges. Initially the edge from y to z is locked. The shortest path from x to z is dependent on the location of other agents capable of unlocking the edge.

resents a usable *From-To* edge while $\mathit{HasKey}(\mathit{Agt}, \mathit{From}, \mathit{To})$ represents a potential *From-To* edge.

The cost function in this domain is equivalent to plan length. Note that any program has bounded-benefit with this cost function: $l_0 = 0$ and $\epsilon = 1$.

First consider the initial situation where agent a is at x, agent b is at y, and b has a key to unlock $y \rightarrow z$. The minimum cost joint execution is for b to unlock $y \rightarrow z$, and a to travel from x to y to z. If however b is initially at z, the minimum cost path is for x to travel via p and q. Note carefully that a sum of per-agent relaxation will not be admissible here in general: a's optimal path over-estimates the true cost.

Preconditions persist in general, and the best time to allow another agent to achieve our precondition is not necessarily the situation immediately preceding the action requiring that precondition. There are two options as to where we can allow this necessary choice: where we assume a fluent becomes true; and when we allow a fluent we cause to be used by another agent.

We choose to use the first of these, as there will always be a bounded choice of situations in which we can assume a fluent. We then force agents to take the benefit associated with causing a fluent at the point it was actually caused.

That is, in $do([\mathit{move}(a, x, y), \mathit{move}(a, y, z)], S_0)$ agent a can assume that $y \rightarrow z$ was unlocked in S_0 or after a's first action, and thus may choose either of these two penalties to apply. However in $do([\mathit{move}(b, p, y), \mathtt{unlock}(b, y, z)], S_0)$ agent b has no choice and must take the bonus available after the first action.

In the cooperative navigation domain nothing negates the preconditions we are relaxing. However, in general an agent could negate an assumed precondition before the point at which it was required. To avoid this, we introduce some analysis functions that allow us to determine the situations from which an assumed fluent would have persisted. We assume that fluents have successor state axioms transformed from effect axioms ϕ^+ and ϕ^- in the standard way (Reiter, 2001). $\mathit{lasteff}(f, s) = s'$ computes the last situation which would have had either a positive or negative effect on f:

$$\begin{array}{lcl} \mathit{lasteff}(f, S_0) & = & S_0 \\ \mathit{lasteff}(f, do(\alpha, s)) & = & \begin{cases} do(\alpha, s) & \textbf{if } \phi_f^{\pm}(\alpha, s) \\ \mathit{lasteff}(f, s) & \textbf{otherwise} \end{cases} \end{array}$$

Here $\phi_f^{\pm}(\alpha, s) \equiv \phi_f^+(\alpha, s) \vee \phi_f^-(\alpha, s)$. For example, in the situation $s = do(move(a, x, y), S_0)$:

$$\begin{array}{lcl} \mathit{lasteff}(At(a), s) & = & s \\ \mathit{lasteff}(Unlocked(y, z), s) & = & S_0 \end{array}$$

We then restrict situations in which f can be assumed before situation s, to situations s' with $\mathit{lasteff}(f, s) \preceq s' \preceq s$.

To transform a regressable query in a basic action theory in such a way that we can apply penalties, we need to detect which assumptions Δ are made. For this purpose we define Precondition Relaxed Regression $\mathcal{PRR}_R^{\Delta}$ in terms of a set of relaxable fluents R. This can be considered a kind of abductive reasoning, but differs in that any relaxable fluent $F(s)$ must be explicitly assumed if it is ever true, even if it is caused by some action already present in s.

In the cooperative navigation example, $Unlocked(y, z)$ must be assumed (recorded in Δ) to ensure that agent a can find the optimal plan. That is, a must assume that the (y, z) edge becomes unlocked before attempting $move(a, y, z)$.

Precondition relaxed regression is defined like usual regression, except for the cases for atomic fluents:

$$\mathcal{PRR}^{\Delta}_{R}(f(S_0)) = \\ (\llbracket f(S_0) \rrbracket \notin R \wedge f(S_0)) \vee (\llbracket f(S_0) \rrbracket \in R \wedge \llbracket f(S_0) \rrbracket \in \Delta)$$

$$\mathcal{PRR}^{\Delta}_{R}(f(s)) = \\ (s = do(\alpha, s')) \wedge \llbracket f(s) \rrbracket \notin R \wedge \mathcal{PRR}^{\Delta}_{R}(\Phi_f(\alpha, s')) \vee \\ \llbracket f(s) \rrbracket \in R \wedge \exists s''.(\mathit{lasteff}(f, s) \preceq s'' \preceq s \wedge \llbracket f(s'') \rrbracket \in \Delta)$$

Here Φ_f is the successor state axiom for f. The brackets $\llbracket \ldots \rrbracket$ have no semantic significance; we use them simply as a reminder that Δ and R store *representations* $\llbracket f(s) \rrbracket$ of fluents f to evaluate at situations s. In the definition, any relaxable fluent $f \in R$ required to be true to satisfy the formula is in Δ, evaluated at some point between when it is required and the last effect on f.

We assume that any regressed formula is in negation normal form, that is, only atomic fluents are negated, not whole expressions. Additionally we assume that the negation of a fluent is represented as a separate fluent with opposite effect axioms, i.e., $\neg f(s)$ is replaced with $f'(s)$ where f' has effect axioms $\phi^{+}_{f'} = \phi^{-}_{f}$ and $\phi^{-}_{f'} = \phi^{+}_{f}$. This is simply syntactic sugar that simplifies our formulae and proofs.

We can now use the assumptions, Δ, computed in the precondition relaxed problem to compute penalties (Lagrange multipliers). We define an *AssumptionCost* function that computes the total cost of those assumptions:

$$\mathit{AssumptionCost}(\lambda, \Delta) = \sum_{f(s) \in \Delta} \lambda(f, s)$$

We also define a *Bonuses* function that calculates the benefit of causing a fact, by

summing those relaxed fluents that actually hold in s:[3]

$$Bonuses(R, \lambda, s) = -\sum_{f(s) \in R} \lambda(f, s) \cdot \langle \Phi_f[s] \rangle \cdot \langle lasteff(f, s) = s \rangle$$

We combine these components with the original cost to define a penalised cost function:

$$Cost_\lambda(R, \Delta, s) = Cost(s) + AssumptionCost(\lambda, \Delta) - Bonuses(R, \lambda, s)$$

We illustrate this concept on a handful of situations in the cooperative navigation domain using the following penalties:

$$\begin{aligned} \lambda_1(Unlocked(x, y), s) &= 9 && \Leftarrow time(s) = 0 \\ \lambda_1(Unlocked(x, y), s) &= 1 && \Leftarrow time(s) = 1 \\ \lambda_1(\_, s) &= 0 && \textbf{otherwise} \end{aligned}$$

The assumption necessary to perform

$$do([move(a, x, y), move(a, y, z)], S_0))$$

is just $\Delta = \{Unlocked(x, y, do(move(a, x, y), S_0))\}$. Using it, we compute the penalties: $AssumptionCost(\lambda_1, \Delta) = 1$ and no bonuses. Importantly,

$$do([move(b, p, y), \texttt{unlock}(b, y, z), move(b, y, z)], S_0)$$

still requires the assumption

$$Unlocked(x, y, do([move(b, p, y), \texttt{unlock}(b, y, z)], S_0))$$

[3] We use $\langle$ and $\rangle$ as Iverson brackets. $\langle P \rangle$ denotes a 0-1 variable which takes the value 1 iff condition P holds.

in spite of the fact that the action sequence causes this assumed fluent. This ensures that all relaxable fluents are treated uniformly across all agents.

In many domains, the only rational choice is to assume the fluent in the least penalised situation where that fluent would persist to s:

$$\min_{\lambda(f,s)} (s' \mid \mathit{lasteff}(f,s) \preceq s' \preceq s)$$

This observation is important in reducing the computational burden of the planning problem. The obvious exception to this is when a fluent has already been assumed in order to enable an earlier action, in which case there may be no need to assume the same fluent twice. E.g., if an agent takes the route $y \rightarrow z \rightarrow q \rightarrow p \rightarrow y \rightarrow z$, the agent need only assume that $y \rightarrow z$ is unlocked once, before starting the circuit.

Definition 4.6 (Precondition-relaxed planning)**.** *The precondition relaxed planning problem with a set R of relaxed fluents, $\Delta \subseteq R$ of fluents assumed within the execution s, and a penalty function λ, is defined*

$$PRDo_\lambda(R, \Delta, \delta, S_0, s) \equiv \mathcal{PRR}_R^\Delta(Do(\delta, S_0, s))$$

Lemma 4.1. *For any choice of relaxed fluents R, any legal execution is legal in the precondition-relaxed problem, i.e.,*

$$(\mathcal{D} \models Do(\delta, S_0, s)) \Rightarrow \forall R.\exists \Delta.(\mathcal{D} \models PRDo(R, \Delta, \delta, S_0, s)),$$

Proof. $\Delta \subseteq \{f(s) \in R \mid \mathcal{R}[f(s)]\}$, the set of assumptions required is at most the set of relaxed fluents that actually hold in the original problem. □

Lemma 4.2. *For any choice of relaxed fluents R and penalty function λ satisfying $\forall f, s.\lambda(f,s) \geq 0$ the penalised cost of any solution s^* to the original problem is no more than the non-penalised cost.*

Proof sketch. By contradiction.

Since s^* is a solution to the non-relaxed problem, for each f which is a precondition of any action in s^*, it must hold that $\forall s \preceq s^* : f(\mathit{lasteff}(f, s))$, as any subsequent effect would either negate f, causing s^* not to be a solution, or it would cause f in which case that subsequent situation would be the cause of f.

Assume the penalised cost is greater than the non-penalised cost. Then there exists some precondition f assumed in some situation $s \preceq s^*$ where the cost of assuming f exceeds the bonus for causing f (because the restriction on λ means there cannot be a penalty for causing f):

$$\mathit{min}(\lambda(f, s') \mid \mathit{lasteff}(f, s) \preceq s' \prec s) > \lambda(f, \mathit{lasteff}(f, s))$$

But it is impossible that the minimum of a set that includes the value $\lambda(f, \mathit{lasteff}(f, s))$ exceeds that value, hence we have a contradiction. □

Theorem 4.2. *For any choice of relaxed fluents R and penalty function λ satisfying $\forall f, s.\lambda(f, s) \geq 0$,*

$$\min_{\mathit{Cost}_\lambda(\Delta, S)} \{S \mid \mathcal{D} \models \mathit{PRDo}(R, \Delta, \delta, S_0, S)\}$$

is a relaxation of

$$\min_{\mathit{Cost}(S)} \{S \mid \mathcal{D} \models \mathit{Do}(\delta, S_0, S)\}$$

Proof. Directly from Lemmas 4.1 and 4.2. □

For our relaxation to remain decidable, we place a slightly different restriction on the penalty function: it should have finitely-many ways that actions can have negative cost. An action can have negative cost if any precondition or postcondition has a non-zero penalty.

Theorem 4.3. *Let δ be a bounded-benefit program and R a set of fluents to relax such that the branching factor of δ remains bounded. For any choice of relaxed fluents R (where the branching factor remains bounded) and penalty function λ and bounded-benefit program*

δ. If the penalty function λ has a finite number of solutions to $\lambda(f,s) \neq 0$, and a finite total magnitude $M = \sum|\lambda(f,s)|$, then the precondition relaxed problem is itself a bounded-benefit program.

Proof sketch. By assumption there is a bounded total benefit that can be gained, given an initial lower bound. Defining $lb'(l) = lb(l) - M$ gives a lower bound for the relaxed problem with the necessary properties. □

When lb and M are computable and known, we can use lb' to apply informed search algorithms in sub-problems.

Note that when R satisfies the requirements of Theorem 4.3, an infinite number of penalty functions can be systematically generated. Importantly, to estimate the joint cost, one need not know any agent's program, merely be able to query agents for their optimal relaxed plan given a set of relaxed fluents and penalty function.

4.6 Multi-Agent Relaxations

We now have a mechanism to generate per-agent relaxed plans from penalties. The next logical step is to merge these into a relaxed joint execution. There are several approaches to merging agent plans, the most general approach is to treat agent i's plan as a literal program: $do([\alpha_1, \ldots, \alpha_n], S_0)$ becomes $\delta_i \equiv \alpha_1; \cdots; \alpha_n$; then to merge a pair of such literal programs for agents i and j by taking the minimum cost plan generated by $Do(\delta_i \| \delta_j, S_0, s)$. Other merge operators cannot have solutions that are not also solutions to this general merge operator.

We assume that the actions in s originating from s_1 are distinguishable from those in s_2 using a function $agent(\alpha)$ that returns the agent responsible for this action. We can then define a predicate to un-merge a joint execution into a per-

agent equivalent:

$$\begin{aligned} m^{-1}(agt, S_0, s') =& S_0 \\ m^{-1}(agt, do(\alpha, s), s') =& (s' = do(\alpha, s) \wedge agent(\alpha) = agt) \\ & \vee (s' = s \wedge agent(\alpha) \neq agt) \end{aligned}$$

This is an inverse of any merge operator, regardless of its definition. This also allows us to determine the agent causing a fluent in a joint execution:

$$CausedBy(f, s, agt) \equiv lasteff(f, do(\alpha, s)) = do(\alpha', s') \wedge agent(\alpha') = agt$$

A more practical approach to merging, which we use in our implementation, is temporal merging based on the temporal semantics of *MIndiGolog* (Kelly and Pearce, 2006). Each agent's plan maintains a per-agent notion of time, and the merged plan must ensure that the actions in the resulting plan are a topological sort of the actions in the agent plans that is consistent with the timestamp for that action. that is, if two agents, i and j perform actions α_i and α_j then, for each s and s' representing a prefix of the merged joint execution,

$$do(\alpha_i, s) \preceq do(\alpha_j, s') \Rightarrow time_i(do(\alpha_i, s)) \leq time_j(do(\alpha_j, s')).$$

must hold for the merge to be temporally consistent.

In the cooperative navigation domain, under the assumption that each action takes one unit of time, a temporal consistent merge of $do([move(a, x, y), move(a, y, z)], S_0)$

and $do([\texttt{unlock}(b, x, y), move(b, x, p), move(b, p, q)], S_0)$ would be

$$do([move(a, x, y), \texttt{unlock}(b, x, y), \\ move(a, y, z), move(b, x, p), move(b, p, q)], \\ S_0)$$

but

$$do([move(a, x, y), \texttt{unlock}(b, x, y), \\ move(b, x, p), move(b, p, q), move(a, y, z)], \\ S_0)$$

would not be temporally consistent. Namely, $move(a, y, z)$, performed at time 2 in the per-agent plans, is performed after $move(b, x, p)$ in the merged plan, but this second action was performed at time 1 in the per-agent plan.

This approach massively reduces the computational overhead of merging plans, and also allows us to apply penalty functions more consistently between agents by taking time into account. For example, in the multi-agent navigation domain from Figure 4.1, the penalty for assuming $x \to y$ is unlocked at time 0 is high, as this is un-achievable.

In computing a multi-agent relaxation, we would like to totally avoid merging plans to compute a lower bound. To this end we define a class of (penalised) cost functions that allow us to simply sum the per-agent costs. We refer to such cost functions as "merge-consistent".

Definition 4.7 (Merge-consistent cost)**.** *The penalty function $\lambda(f, s)$ and the cost function $Cost_\lambda$ are consistent with a merge operation $m(\delta_1, \delta_2)$ under a relaxed set R*

of fluents if for all relaxed plans s_1 and s_2, with assumption set $\Delta = \Delta_1 \cup \Delta_2 \subset R$,

$$Cost_\lambda(R, \Delta, m(s_1, s_2)) = Cost_\lambda(R, \Delta_1, s_1) + Cost_\lambda(R, \Delta_2, s_2)$$

Merge-consistent penalties are quite easy to develop in practice within temporal domains, when penalties and costs are consistently applied at the same time points. Unfortunately this approach alone is not always sufficient to guarantee that the per-agent relaxed optima (which are the only solutions we want to consider ideally) can be merged to produce the optimal joint execution.

Referring again to Figure 4.1, now assume that a needs to plan a path from x to z and b one from p to z. As usual b may unlock the locked edge, and $Unlocked(y, z)$ is relaxed in all situations except S_0. The optimal solution is for both agents to travel via $y \rightarrow z$. We see the two non-penalised per-agent optima are $do([move(a, x, y), move(a, y, z)], S_0)$ and $do([move(b, p, q), move(b, q, z)], S_0)$.

No penalty function we can give to b can give any incentive to travel via $y \rightarrow z$, as b will necessarily pay the same penalty for assumption as it gets in bonuses, unless b performs some intermediate action, which would be sub-optimal. The reason for this is that the precondition can benefit multiple agents simultaneously. Hence we duplicate $Unlocked(y, z)$ as $Unlocked_a(y, z)$ and $Unlocked_b(y, z)$ with identical successor state axioms. Each agent then requires only its own copy of this fluent to perform the action, and this allows the bonuses to exceed the penalties. Importantly, this transformation also guarantees that the Δs for each agent do not overlap, and therefore no assumption costs will be double counted when a single assumption could be used in the relaxed joint program $\delta_1 \| \delta_2$.

Definition 4.8 (Shared Relaxation). *A shared relaxation is a domain $\mathcal{D}$, a set of fluents R and merge operator m such that*

$$\begin{aligned}\mathcal{D} \models Do(\delta_1 \| \delta_2, S_0, s) \Rightarrow \\ \exists \Delta_1, \Delta_2 : PRDo(R, \Delta_1, \delta_1, S_0, s_1) \\ \wedge\, PRDo(R, \Delta_2, \delta_2, S_0, s_2) \wedge m(s_1, s_2) = s \\ \wedge\, \forall f \neg \exists s' \preceq s : \\ f(m^{-1}(1, s')) \in \Delta_1 \wedge f(m^{-1}(2, s')) \in \Delta_2\end{aligned}$$

That is, it is a relaxed domain where sufficiently many fluents are relaxed to ensure that each agent can operate independently in the relaxed domain and thus all legal executions in the original can be generated by merging relaxed single-agent plans, with no assumptions shared between agents. This non-overlap restriction guarantees that penalties cannot be "double counted".

Theorem 4.4. *Let λ be a penalty function, merge-consistent with a merge operator m under a shared relaxation R. Let A be a set of agents numbered 1 to n, let Δ_i be the set of assumptions made by agent i, and let s_i be agent i's relaxed plan. For all joint executions s which are legal in the original domain:*

$$\exists s_1 \dots s_n : s = m(s_1, \dots, s_n) \wedge \sum_{i \in A} Cost_\lambda(R, \Delta_i, s_i) \leq Cost(s)$$

Proof sketch. By Definition 4.8, $s_1 \dots s_n$ exist. By Definition 4.7, $\sum_{i \in A} Cost_\lambda(R, \Delta_i, s_i)$ is equivalent to $Cost_\lambda(R, \Delta, s)$. Namely, as no two Δ_i overlap, no penalties can be double counted. So by Lemma 4.2, $Cost_\lambda(R, \Delta, s) \leq Cost(s)$. □

Obviously, to be practically useful such a relaxation must be computationally easier than the original problem. Choosing such a set is an exercise for the modeller. Table 4.1 compares the precondition-relaxed runtime with a non-relaxed

implementation in *MIndiGolog* and observe speedups in excess of 9000× in some cases. Importantly, we can see both from Definition 4.7 and our experimental results that the complexity of computing the relaxation for n agents is in $O(n)$ assuming the penalty function and remaining domain are unchanged.

4.7 Constructing Joint Executions

We now have methods to guide agents towards behaviours that benefit the overall objective in the precondition relaxed problem. We would like to use a similar approach to achieve the same aim in the original problem.

We can also guide other agents away from hard to satisfy assumptions, and towards causing helpful preconditions. By violating the non-negativity assumption in Lemma 4.2, we can also do the converse: guide agents towards assumptions, and away from causing interfering effects.

Theorem 4.5. *Given a feasible joint execution s_j such that $Do(S_0, \delta_1 \| \delta_2, s_j) \wedge s_j = m(s_1, s_2)$, there exist relaxed precondition sets R_1 and R_2 such that $PRDo(R_1, \Delta_1, S_0, \delta_1, s_1) \wedge PRDo(R_2, \Delta_2, S_0, \delta_2, s_2)$.*

Proof sketch. By Lemma 4.1, $PRDo(R, \Delta, S_0, \delta_1 \| \delta_2, s)$ must hold for any R. If we then constrain R_i to be a superset of each precondition f of any action whose last effect was caused by a different action, that is, for all agents *agt* the following should hold for each fluent f true in any situation $s \preceq s_j$:

$$CausedBy(f, s, agt') \wedge f(s) \wedge agt' \neq agt \Rightarrow m^{-1}(agt, s, s') \wedge [\![f(s')]\!] \in R_i \wedge [\![f(s)]\!] \in R$$

Each R_i is sufficient to allow each agent to perform the action sequence required using $\mathcal{PRR}$. Any superset of R_i allows at least the same solution set. □

To construct a feasible joint execution we use a form of Lagrangian relaxation. We solve the relaxed problem for each agent and then check whether there is a feasible interleaving. If not, we will discover some relaxed fluents that are assumed but not caused by any agent. The penalty for these fluents will then be increased and the process iterated.

Consider the situation in the cooperative navigation domain, where a is at x and b is at y initially and can unlock $y \rightarrow z$. If we relax the $\mathit{Unlocked}(y, z)$ fluent with penalty 0, a's optimal relaxed plan is $do([\mathit{move}(a, x, y), \mathit{move}(a, y, z)], S_0)$, which is exactly the plan a should execute. However, with the same penalty, b's optimal plan is to do nothing. We observe that these action sequences cannot be interleaved to form a legal joint execution. The reason for this is that there is a relaxed fluent assumed by a that is not generated by any agent. If instead we relax the same fluent with a penalty of 2, a will perform the same action sequence (but incur 2 additional cost units of penalty), and b will perform $do(\texttt{unlock}(b, y, z), S_0)$, gaining 2 units in bonuses. Importantly, agent a has no knowledge whatsoever of agent b, and vice-versa. Even the central planner setting the penalties has no specific a priori knowledge of b's capability to unlock $y \rightarrow z$, only by giving b sufficient incentive to achieve this precondition for a is this capability discovered. All that each agent knows is that there may exist an agent capable of causing $\mathit{Unlocked}(y, z)$, and an agent that may rely on the same fluent.

Table 4.1 shows results from the relaxations used in the agent-based rail scheduling application described in chapter 3. In this application we schedule a set of train services on a shared rail network subject to mutual exclusion constraints (two trains cannot simultaneously occupy the same track section). In the results we present, these mutual exclusion constraints are relaxed, and we apply a temporal penalty function and merge operator. As this represents a scheduling problem, we compare our approach to a state-of-the-art solver for scheduling problems, `cpx` (Schutt et al., 2013).

Agents	*Golog*	cpx	Relaxed*	Relax-&-Merge
2	0.4	0.7	0.1	1.6
4	—	1.7	0.2	2.0
8	—	7.5	1.1	7.6
16	—	37.9	4.6	29.7
32	—	—	12.5	50.3
64	—	—	25.3	150.3

Table 4.1: Time to first solution in seconds of the Bulk Freight Rail Scheduling Problem described in chapter 3. (— denotes runtime exceeds 1800s, or memory usage exceeds 4GB). (* time to *optimal* solution of the precondition relaxed problem)

The penalties were determined by the optimal dual solution to a linear program modelling the mutual exclusion constraints. The per-agent relaxed plans were then incrementally added to a pool, the linear program was re-solved, and the process was iterated. This process is an application of Branch-and-Price (Barnhart et al., 1998)—for details, see chapter 3. The results illustrate the speedup that can be gained by relaxing the right preconditions, and that relax-&-merge can be used to find feasible joint executions significantly faster than existing approaches.

Table 4.1 shows the effectiveness of using the relax-&-merge approach to constructing feasible joint executions. It also shows the average time to solve a full joint relaxation extrapolated from the average time to solve a single agent's relaxation. The time is extrapolated because the relaxation was stopped early whenever an agent plan that changed the violated constraints was generated. In the table we compare a special case of relax-&-merge (fragment-based planning chapter 3) with (column 2) *MIndiGolog* (Kelly and Pearce, 2006), a Prolog-based multi-agent Golog interpreter; and (column 3) `cpx`, a constraint programming solver using Lazy Clause Generation (Ohrimenko, Stuckey, and Codish, 2009). The implementation from chapter 3 was instrumented to find the proportion of time spent on relaxation (column 4). All experiments were performed on a 2.4 GHz

Intel Core i3 with 4GB RAM running Ubuntu 12.04. Our implementation used Gurobi 5.1, CPython 2.7.3. We used the binary version of `cpx` included with MiniZinc 1.6.

4.8 Related Work

Baier et al. (Baier et al., 2008) have considered the compilation of a restricted class of Golog programs using an intermediate language for capturing temporal logic preferences. This could allow PDDL3-compliant classical planners to tackle problems of a similar nature, utilising a range of well studied tractable relaxations. However, PDDL is less expressive than *Golog* in general (Röger and Nebel, 2007). Moreover, the application of classical approaches to multi-agent planning problems requires knowledge of each agent's goals and transition function. In contrast, the approach proposed in this chapter requires only that agents can generate optimal plans given a penalty function.

Delete relaxation is the most common of these relaxations and has been applied to *Golog* to generate a heuristically good execution (Blom and Pearce, 2010). However (Blom and Pearce, 2010) does not use the delete relaxation to generate lower bounds as no attempt is made to approximate the optimal solution to the relaxed problem, this could limit the accuracy of information derived from this heuristic. The alternative search strategies that are applicable to the bounded-benefit programs we introduce in this chapter could allow this issue to be addressed.

In contrast to delete relaxation, our approach introduces *precondition* relaxation and demonstrates its applicability to both computing relaxations in multi-agent problems, and constructing feasible joint executions.

Abduction has been used extensively in planning, however only one published approach the authors are aware of uses abduction to synthesise plans. In

this approach planning is achieved using predicates distinguished as *abducible*, and is formalised in the event calculus (Shanahan, 2000), however this work does not consider optimisation or multi-agent applications. The use of costs in conjunction with abduction has also been used in multi-agent reasoning for plan recognition (Appelt and Pollack, 1992).

4.9 Conclusions

We have introduced a notion of cost to the situation calculus, and described a logical framework to prove the relative quality of solutions to planning problems in *Golog* without the need to enumerate the solution space. These enable more control of the search algorithm, and with further work could allow the integration of informed search techniques from automated planning.

Our main contribution is precondition relaxations, a class of relaxations that are particularly applicable to multi-agent domains. Our experimental results show that relaxations can be chosen which yield dramatic speedups and scale linearly with increasing numbers of interacting agents. These relaxations can be usefully and systematically varied, so as to not only improve the lower bounds they generate, but also to generate feasible joint executions from relaxed per-agent plans.

We have explained the restrictions on domains where these relaxations can be applied and described some simple transformations that can be applied, to ensure these properties hold. Additionally we describe temporal merging which represents additional constraints that can be added to a domain such that the overhead of merging per-agent plans can be effectively reduced.

Importantly, these relaxations and approaches to search can be applied without knowledge of any agent's program, so long as those agents can be queried for their optimal plan under a given penalty function.

Part II

Cost-Optimal Classical Planning

Introduction to Part II

Cost-optimal classical planning has been dominated by heuristic search algorithms in recent decades, and the A* algorithm and its symbolic analogues in particular. We introduce two algorithms in this part: a novel plan-space search approach which generalizes the concept of landmarks; and a depth-first approach which adapts recent ideas from satisficing planning to the optimal case.

The algorithms in this part of the thesis can be seen as attempting to incrementally learn the perfect heuristic h^*, learning from two quite different notions of conflict.

Chapter 5 introduces "operator sequencing" which iteratively computes an optimal set of operators which may be sequencable into a plan, the sum of the costs of these operators being an admissible heuristic estimate. If that set is not sequencable, a "generalized landmark" is computed, forcing the next set of operators to contain at least one copy of an operator that was missing from the previous iteration.

Chapter 6 presents "Conflict-Directed Heuristic Learning", the only completely state-based algorithm in this thesis. This approach generalizes IDA*, and integrates heuristic evaluation, successor generation and dominated path detection in a single satisfiability problem using our novel "clausal heuristics" that can be used to incrementally learn the perfect heuristic.

Chapter 5
Sequencing Operator Counts

In this chapter we introduce "Generalised Landmarks", and integrate them into the recently developed framework of operator-counting heuristics. We use a SAT-MIP hybrid approach to incrementally learn Generalised Landmarks that encode the perfect heuristic, h^*, stopping when it finds a plan having the same set of operators as the admissible operator counting heuristic.[1]

5.1 Introduction

We investigate the problem of sequencing operator counts obtained from an operator counting heuristic. The algorithm will find a feasible sequence, if it exists, or obtain an explanation why there is no plan that uses only the operators counted. We refer to these explanations as generalised disjunctive action landmarks.

Disjunctive action landmarks are a core feature of many admissible heuristics (Helmert and Domshlak, 2009; Bonet and Helmert, 2010; Haslum, Slaney, and Thiébaux, 2012b; Imai and Fukunaga, 2014). Admissible heuristics based on these landmarks count the occurrence of any operator at most once. Most are

[1]The research presented in this chapter was published in Toby O. Davies, Adrian R. Pearce, Peter J. Stuckey, and Nir Lipovetzky (2015). "Sequencing Operator Counts". In: *International Conference on Automated Planning and Scheduling (ICAPS 15)*, pp. 61–69, and reproduced in abridged form as Toby O. Davies, Adrian R. Pearce, Peter J. Stuckey, and Nir Lipovetzky (2016). "Sequencing Operator Counts". In: *International Joint Conference on Artificial Intelligence (IJCAI 16)*, pp. 4140–4144.

dominated by the optimal delete relaxation h^+ (Helmert and Domshlak, 2009).

We generalise this notion of disjunctive action landmarks to count operators multiple times, and show that admissible heuristics using generalised landmarks are capable of defining the perfect heuristic h^*. As disjunctive action landmarks are the only kind of landmark we consider in this chapter, we will refer to them simply as "landmarks".

We present a complete, incremental algorithm for generating generalised landmarks, prove that generalised landmarks can encode h^*, and experimentally verify that this algorithm computes h^*. We show that even if we compute h^*, the corresponding operator count does not necessarily represent a valid plan. Our approach can be used both as an incremental lower bound function and as an optimal planner, much like h^{++} (Haslum, Slaney, and Thiébaux, 2012a), as our approach does not terminate until it finds a proof that it has computed h^*, i.e. finds a plan with optimal cost.

We explain this approach to planning in terms of Logic-Based Benders Decomposition (LBBD). LBBD partitions an optimisation problem in terms of a Mixed Integer Programming master problem, and one or more combinatorial sub-problems used to explain flaws in the master problem.

This approach to planning is particularly promising for two reasons. Firstly, it introduces a principled interaction between operator-counting heuristics and SAT. This interaction can be applied to any explanation-based combinatorial search approach including SAT Modulo Theories (SMT) (Nieuwenhuis, Oliveras, and Tinelli, 2006) and constraint programming using Lazy Clause Generation (LCG) (Ohrimenko, Stuckey, and Codish, 2009). Constraints or theories capable of generating clausal explanations can be added to the SAT model we present, potentially allowing direct integration of cost-optimal planning with SMT and state-of-the-art scheduling approaches based on LCG. Planning Modulo Theories problems (Gregory et al., 2012) could therefore potentially be tackled using the exten-

sive range of *existing* theories and constraints *already implemented* by the SMT and constraint programming communities.

Secondly, this approach decomposes the planning problem into problems for which there exist well-suited optimisation technologies: Mixed Integer Programming handling the linear counting constraints; and Conflict-Directed Clause Learning for the problem of operator sequencing given operator counts. This allows planning to take advantage of the ever improving performance of both of these widespread and industrially applied technologies.

5.2 Preliminaries

SAS^+ planning A SAS^+ planning task is a tuple $\langle V, O, s_0, s_*, c\rangle$ where V is a set of finite domain state variables, O is a set of operators, s_0 is a full assignment of each variable to one of its values representing the initial state, and s_* is a partial assignment of some subset of V representing the goal states. Finally c is a function $O \to \mathbb{N}^+$ that assigns a non-negative cost to each operator.

Each variable $X \in V$ has a domain $D(X)$, we sometimes abuse notation and write $X = x \in V$ which should be read $X \in V \wedge x \in D(X)$. Each operator o has a set of preconditions $pre(o)$ which is a partial assignment representing the preconditions of that operator, and a set of postconditions $post(o)$ which is a partial assignment representing the effects of the operator. Producers, $prod(X{=}x) = \{o \mid X{=}x' \in pre(o) \wedge X{=}x \in post(o) \wedge x' \neq x\}$ are the operators which cause $X{=}x$ to become true.[2]

A state s in the search space is a full assignment of every variable to a value. State s is said to satisfy a partial assignment F if all assignments in F are also in s, i.e. $X{=}x \in F \Rightarrow X{=}x \in s$. A state is said to be a goal state if it satisfies the partial

[2]Note for readers familiar with the planning literature, for simplicity, we do not distinguish between preconditions and prevail conditions in this thesis.

assignment s_*.

An operator $o \in O$ is applicable in s if s satisfies the partial assignment $pre(o)$. If o is applicable in state s, applying o yields a new state s' which is the same as s except that all assignments $X{=}x \in post(o)$ replace any assignment to X.

A plan π is a sequence of operators $o_1, \cdots o_n$ such that o_1 is applicable in s_0, each subsequent operator is applicable in the state resulting from applying the previous operators in sequence, and the final state satisfies s_*. An optimal plan has the minimum sum of operator costs of all plans, a SAS^+ planning task may have many optimal plans.

Mixed Integer Programming A Mixed Integer Program (MIP) is a representation of a combinatorial optimisation problem in terms of linear constraints over some finite set of integer and continuous variables. Finding a solution to a MIP is an NP-complete problem, however its linear relaxation, (which replaces all integer variables with continuous ones) can be optimised in polynomial time.

In recent years many admissible planning heuristics have been proposed that use linear programs (Van Den Briel et al., 2007; Coles, Fox, Long, et al., 2008; Bonet, 2013; Pommerening, Röger, et al., 2014; Bonet and Briel, 2014).

Of particular interest is the family of operator-counting heuristics. Operator-counting uses a linear programming framework with a common set of variables Y_o representing the count of occurrences of each operator o in some relaxed representation of a plan. One or more component heuristics can be encoded as linear constraints on these variables such that the combined operator-counting heuristic dominates each of the component heuristics, often strictly (Pommerening, Röger, et al., 2014). Pommerening, Röger, et al. describe how to encode a variety of heuristics within this framework, including optimal cost partitioning (Katz and Domshlak, 2008) of LM-Cut landmarks (Helmert and Domshlak, 2009).

As presented in the original paper, operator-counting heuristics may be linear

programs, by requiring operator count variables to be integer, we obtain a MIP that is at least as tight as the LP. As we wish to obtain perfect operator counts, branching on operators will be essential in general, and in the remainder of this chapter we will assume that any operator-counting heuristics are in fact MIPs, and we will be explicit when we solve only the linear relaxation.

Operator Counts The solution to an operator-counting heuristic assigns a count to each operator o whenever the MIP is optimised. To distinguish the count assigned to each operator by a solution to an operator-counting heuristic from the variable Y_o, we refer to a solution to the heuristic as an operator-count $\mathcal{C}$.

We primarily treat an operator-count as a function from operators to their count assigned in the last solution of the MIP. The values of each Y_o will change throughout any solving process, conversely an operator count $\mathcal{C}$ should be considered an immutable copy of the assignments. We also refer to operator counts as multisets: a count $\mathcal{C}$ is said to be a superset of another count $\mathcal{C}'$ if $\forall o \in O : \mathcal{C}(o) \geq \mathcal{C}'(o)$.

An operator-count $\mathcal{C}$ is said to be a projection of a plan π if for each distinct operator o, there exist exactly $\mathcal{C}(o)$ copies of that operator in π. We say an operator count is *perfect* if it is the projection of an optimal plan.

Delete Relaxation The delete relaxation changes the SAS^+ transition function such that applying an operator o does not replace previous assignments to variables, but accumulates them.

Imai and Fukunaga (2014) introduce a new MIP encoding of the optimal delete relaxation heuristic h^+. Their model uses 0-1 variables $\mathcal{U}(o)$ for each operator o to represent the fact that at least one o appears in the delete-relaxed plan. They also present an extension to h^+ which they call "counting constraints" which are roughly equivalent to the lower-bound constraints in single-variable flow mod-

els (Van Den Briel et al., 2007). These constraints utilise additional integer variables $\mathcal{N}(o)$ which count occurrences of an operator o. For consistency with Pommerening, Röger, et al. (2014), we will denote $\mathcal{N}(o)$ as Y_o.

Incremental lower bounding Incremental lower bounding is a general technique for obtaining high-quality lower bounds, which can be useful in proving the quality of an existing plan. Incremental lower bounding was most prominently used in planning by Haslum, Slaney, and Thiébaux (2012), however the technique is used throughout the various optimisation communities, referred to as "dual" techniques, reflecting the dual (lower) bound obtained from a linear program. Haslum, Slaney, and Thiébaux (2012) describe a distinctive property of incremental lower bounding techniques as "informed by flaws in the current [optimum]".

5.3 Generalised Landmarks

The constraints in the h^+ model of Imai and Fukunaga (2014) rely on operator variables being binary. In general this does not hold with operator counting heuristics. In particular, flow-based heuristics (Van Den Briel et al., 2007; Bonet, 2013; Bonet and Briel, 2014) can count arbitrarily many executions of an operator. Imai and Fukunaga use $\mathcal{N}(o)$ variables to handle this in their "counting constraint" extension. Changing these $\mathcal{N}(o)$ variables to Y_o leads us to a simple but interesting alternative notation for their core $\mathcal{U}(o)$ variables, which we denote $[Y_o \geq 1]$.[3] We generalise this notion by separating variables representing the number of times an operator o occurs, Y_o, from variables representing if an operator occurs at least k times, $[Y_o \geq k]$, which we refer to as bounds literals.

[3] Iverson brackets denote binary variables of the form $[P]$ that take the value 1 iff the condition P holds.

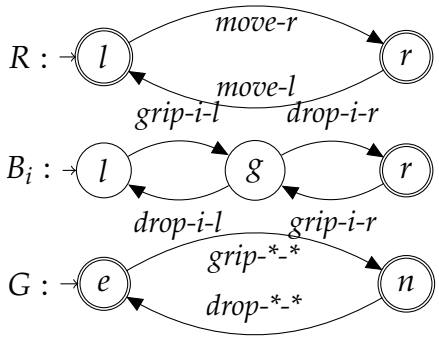


Figure 5.1: Domain Transition Graphs in gripper.

Bounds literals can be used to form linear constraints of the form $[Y_{o_1} \geq k_1] + \cdots + [Y_{o_n} \geq k_n] \geq 1$ which we call *generalised landmark constraints*. Note that only one bounds literal need occur per operator within the same landmark. If the same operator o had two literals $[Y_o \geq k_1]$ and $[Y_o \geq k_2]$ in the same landmark with $k_2 > k_1$, then $[Y_o \geq k_2]$ can be omitted without changing the solutions, as $[Y_o \geq k_1] \geq [Y_o \geq k_2]$.

Definition 5.1. *A generalised landmark constraint is a linear inequality of the form:*

$$\sum_{i \in L} [Y_{o_i} \geq k_i] \geq 1$$

for some $L \subseteq O$.

We call these generalised landmarks because any traditional disjunctive action landmark can be encoded as a generalised landmark by setting all $k_i = 1$.

Consider an instance of the simplified gripper domain shown in Figure 6.2 with 2 balls. The goal is to move both balls from the left room to the right, using a robot with a single "gripper" which can hold only one ball at a time. The robot starts in the left room, and can only pick up and drop balls in the room it is currently occupying.

The operator count obtained by h^+ on this domain counts the following.

$$\mathcal{C}(o) = 1 \quad \textbf{if } o \in \{\textit{move-l}, \textit{move-r}, \textit{grip-1-l}, \textit{grip-2-l}, \textit{drop-1-l}, \textit{drop-2-l}\}$$
$$\mathcal{C}(o) = 0 \quad \textbf{otherwise}$$

Note that there is only one occurrence of the *move-r* operator, however all feasible plans must contain two of this operator. We can add the constraint

$$[Y_{\textit{move-r}} \geq 2] \geq 1$$

to explain this requirement. With the addition of this constraint the MIP returns the optimal operator count for this instance.

If there existed an alternative path for the robot R to move between rooms the constraints are more complex. Consider an additional operator *move-r'* identical to *move-r*. The constraint we described above would no longer be valid: it is possible to solve the planning problem with one of each of the two identical operators.

However adding both of the following constraints:

$$[Y_{\textit{move-r}} \geq 2] + [Y_{\textit{move-r'}} \geq 1] \geq 1$$

and

$$[Y_{\textit{move-r}} \geq 1] + [Y_{\textit{move-r'}} \geq 2] \geq 1$$

captures the requirement that two of the move operators must occur in any feasible plan. These can be read as "either *move-r* occurs at least twice, or *move-r'* occurs at least once"; and "either *move-r* occurs at least once, or *move-r'* occurs at least twice." The conjunction of these implies that a total of at least two of these operators must occur.

To enforce the correct behaviour of bounds literals we need to add the follow-

ing *domain constraints*[4] to our model:

$$[Y_o \geq k] \leq [Y_o \geq k-1] \qquad \text{if } k > 0 \tag{5.1}$$

$$Y_o \geq \sum_{i=1}^{\infty} [Y_o \geq i] \tag{5.2}$$

$$Y_o \leq M[Y_o \geq k] + k - 1 \tag{5.3}$$

Where M is a sufficiently large number such that no feasible plan could contain more than M of any individual operator. In practice this number need only be as large as the longest plan the solver could feasibly solve. Constraint 5.1 ensures that a bound can't hold unless the next smallest bound also holds; 5.2 ensures that if k bounds literals are set, then at least k operators must occur; and finally 5.3 ensures that if k or more operators occur, the bounds literal $[Y_o \geq k]$ must be set.

Note that the constraint

$$[Y_{\textit{move-r}} \geq 1] + [Y_{\textit{move-r'}} \geq 1] \geq 1$$

is semantically equivalent to the traditional landmark constraint

$$Y_{\textit{move-r}} + Y_{\textit{move-r'}} \geq 1$$

however the former has a tighter linear relaxation since $Y_o \geq [Y_o \geq 1]$ always holds for any o. For example, the linear relaxation could assign $[Y_o \geq 1] = 0.5$, $[Y_o \geq 2] = 0.5$, $Y_o = 1$. In this case the constraint $[Y_o \geq 1] \geq 1$ is violated, but $Y_o \geq 1$ is not. Consequently if any operator-counting heuristic uses bounds literals, it is always preferable to encode landmark constraints using the bounds

[4] *Domain constraints* reflect the fact that Y_o variables are finite domain variables, and the bounds literals we use are closely related to bounds literals used in lazy clause generation (Ohrimenko, Stuckey, and Codish, 2009), where the same term is used.

literals.

Lemma 5.1. *A SAS^+ problem's cost function c can be replaced by $c'(o) = c(o) + \epsilon$ where $\epsilon > 0$ leaving at least one identical optimal plan.*

Proof. There exists an upper bound on optimal plan length l. Either all actions are uniform cost and any ϵ will not change the relative solution costs of minimum-length plans, or there exists a minimum cost difference between operators $\delta = \min(c(o) - c(o') \mid o, o' \in O \wedge c(o) > c(o'))$.

If $0 < \epsilon < \frac{\delta}{l}$, the sum of ϵ terms for an optimal plan must be less than δ, and thus can only change the cost-order of plans which are either both suboptimal, or of equal cost according to c. □

Theorem 5.1. *For any solvable SAS^+ planning problem having strictly positive action costs, there exists a set of generalised landmark constraints (with the domain constraints for all the bounds literals involved) such that solving a MIP with these constraints will compute $h^*(s_0)$.*

Proof. An optimal operator count $\mathcal{C}$ (which may initially be empty) can be obtained by solving the MIP. If $\mathcal{C}$ does not represent the projection of a plan, then the generalised landmark constraint:

$$\sum_{o \in O} [Y_o \geq \mathcal{C}(o) + 1] \geq 1$$

can be added.

This constraint can be read "at least one operator must be applied at least one more time". This is clearly violated by $\mathcal{C}$, and can only possibly invalidate subsets of $\mathcal{C}$. If any strict subset of $\mathcal{C}$ were feasible, $\mathcal{C}$ would not be optimal. Consequently this new constraint changes the optimum solution without affecting the admissibility of the MIP.

There are only finitely many distinct operator counts with the same cost, and each iteration of this process invalidates exactly one operator count.

Consequently this process will eventually terminate with: an operator count that is the projection of an optimal plan, if any plan exists; an infeasible MIP; or an operator count containing more operators than states in the state space, implying that no solution exists. □

This process will generate sufficiently many generalised landmarks to compute h^*, but each landmark invalidates only one new operator count. Consequently, using these landmarks in a heuristic would likely be inefficient. If we were to omit bounds literals for some operators from the landmark, it would invalidate many more operator counts. This is similar to traditional landmarks which are stronger when they contain a small subset of operators. To obtain smaller, more focused landmarks we turn to the conflict analysis built into modern "Conflict-Directed Clause Learning" SAT solvers.

5.4 SAT Encoding for Operator Sequencing

Assumptions are a feature of most SAT solvers' incremental interfaces. These allow the user to temporarily assert unit clauses. Importantly, if the resulting formula including these unit clauses is not satisfiable, the final conflict in the unsatisfiability proof can always be re-written in terms of a subset of the assumptions. This conflict clause represents a necessary (though not in general sufficient) property required of any model. In our SAT encoding assumptions will be used to ensure that only the operators selected by the operator counting heuristic are actually used.

The most important high-level constraint in achieving this is the at-most-k constraint (denoted $\leq_k$). $\leq_k (S)$ enforces that k or fewer literals from a set S are simultaneously true. This is a well studied constraint in satisfiability, and we use the sequential counter encoding by Sinz (2005), which is identical to the $O(n)$ encoding of Rintanen (2006) in the $\leq_1$ case.

We denote by $X =_i x$ that $X = x$ holds after operator layer i and by o_i that operator o occurs in layer i.

Given an operator count $\mathcal{C}$ and a number of layers $L = \sum_{o \in O} \mathcal{C}(o)$, the following constraints for each layer l form the core SAT model:

$$
\begin{aligned}
& \leq_1 (\{o_l \mid o \in O\}) \\
\forall X \in V : \quad & \leq_1 (\{X{=}_l x_i \mid x_i \in D(X)\}) \\
\forall X{=}x \in s_0 : \quad & X{=}_0 x \\
\forall o \in O : \quad & \bigwedge_{X \not= x \in pre(o)} (\neg o_l \vee X{=}_{l-1} x) \\
\forall o \in O : \quad & \bigwedge_{X \not= x \in post(o)} (\neg o_l \vee X{=}_l x) \\
\forall X{=}x \in V : \quad & X{=}_{l+1} x \Rightarrow X{=}_l x \vee \bigvee_{o \in prod(X \not= x)} o_l \\
\forall X{=}x \in s_* : \quad & X{=}_L x \vee [\Sigma \mathcal{C}(o) \geq L + 1] \\
\forall o \in O : \quad & \leq_{\mathcal{C}(o)} (\{o_l \mid l \in [1, L]\}) \vee [Y_o \geq \mathcal{C}(o) + 1]
\end{aligned}
$$

Additionally we add the following assumptions: $\neg[\Sigma \mathcal{C}(o) \geq L + 1]$ (i.e. that the goal is achieved by layer L); and $\neg[Y_o \geq \mathcal{C}(o) + 1]$ for each operator o. Any resulting conflict clause will be written in terms of the negation of a subset of these assumptions.

The conflict clause will thus contain $[\Sigma \mathcal{C}(o) \geq L + 1]$ and some subset of the bounds literals implied by the operator count. Specifically it will be of the form:

$$[\Sigma \mathcal{C}(o) \geq L + 1] \vee [Y_{o_1} \geq \mathcal{C}(o_1) + 1] \vee \cdots \vee [Y_{o_n} \geq \mathcal{C}(o_n) + 1]$$

This clause must be a necessary condition on all plans of length L or less. Since it is also satisfied by any operator count having more than L actions in total, it is also a necessary condition on all plans. This translates to a generalised landmark

cut by replacing $\vee$ with $+$ and appending ≥ 1. The only complication is the $\neg[\Sigma\mathcal{C}(o) \geq L+1]$ literal, which we tackle by adding an artificial operator T with zero cost (representing the total operator count) to the MIP, constrained such that:

$$Y_T = \sum_{o \in O} Y_o$$

Using this new operator, we can replace the total operator count literal $[\Sigma\mathcal{C}(o) \geq L+1]$ with the bounds literal for the artificial operator T:

$$[\Sigma\mathcal{C}(o) \geq L+1] \equiv [Y_T \geq L+1]$$

It should be noted that the SAT formula we describe only ensures that no more operators occur than were chosen in the operator count. Thus it can sequence any subset of an operator count, allowing it to be used with approximate solutions while guaranteeing that the same proof of admissibility as in Theorem 5.1 applies.

We described earlier the two "hand-made" generalised landmarks one would add in order to improve the delete relaxation in the gripper domain. However the first generalised landmark generated from the SAT solver was:

$$\begin{aligned}&[Y_{\textit{grip-1-l}} \geq 1] + [Y_{\textit{drop-1-r}} \geq 1] + [Y_{\textit{move-l}} \geq 2] \\ &+ [Y_{\textit{drop-2-r}} \geq 1] + [Y_{\textit{grip-2-l}} \geq 1] + [Y_{\textit{move-r}} \geq 2] \\ &+ [Y_T \geq 7] \geq 1\end{aligned}$$

In spite of the conflict analysis in MiniSAT (Eén and Sörensson, 2004), this cut clearly contains irrelevant bounds literals, and "cut strengthening" (removing irrelevant parts of generated cuts) will definitely be an important improvement to techniques using generalised landmarks. In some scheduling domains, cut strengthening has been shown to be responsible for orders of magnitude decreases in run-time (Ciré, Coban, and Hooker, 2013).

Each operator omitted from a conflict roughly doubles the number of operator counts that conflict applies to, drastically decreasing the number of iterations needed to find a perfect operator count. In our experiments, most of the conflicts learnt included no more than 10% of the total operators. While this sounds good, in practice this still means many conflicts were over 200 operators long, so there is clear room for improvement.

5.5 Planning using Logic-Based Benders

Logic-Based Benders Decomposition (LBBD) (Hooker and Ottosson, 2003) is an approach to decomposing combinatorial search problems into a master MIP and one or more combinatorial sub-problems. The master and sub-problems share some variables such that the sub-problem becomes easier to solve or prove infeasible once those variables it shares with the master are fixed. Importantly, LBBD allows for mixing of different optimisation technologies which may be better suited to the master problem and sub-problems.

The master problem represents a relaxation of the original problem, and the sub-problem checks for and *explains* flaws in that relaxation. Explanations in this context are constraints on the variables in the master problem. By incrementally adding these explanations, the master problem incrementally approaches the true solution.

First the master MIP is solved, and the optimal values of the shared variables are taken from the master, and this optimal assignment is assumed within the sub-problem, which is then solved. If the sub-problem is satisfiable then a solution to the original problem has been found. If, as in our case, the objective function is fully modelled in the master problem, then this solution is optimal.[5]

[5]In general, where the sub-problem requires optimisation this is not true, but we omit this case for simplicity as it does not apply to our decomposition. See Hooker and Ottosson (2003).

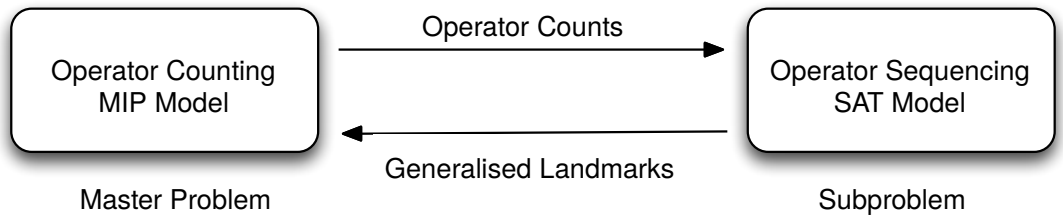


Figure 5.2: A Logic-Based Benders Decomposition Approach to Optimal Planning

If the sub-problem is not satisfiable, some violated necessary condition on the shared variables is detected, and a constraint (the Benders cut) representing this condition is added to the master problem. The process is then iterated until the master problem's relaxation becomes satisfiable.

In our case, the master MIP is any operator-counting heuristic, the operator counts (strictly speaking the bounds literals) are shared variables, and the termination condition is that the optimal operator count is perfect.

Canonical planning (where each operator can be applied at most once) is NP-complete (Vidal and Geffner, 2006), meaningfully easier than the full planning problem. Many domains in planning are canonically plannable, that is there exists a plan containing only one instance of each operator. Our sub-problem of sequencing the operators is pseudo-polynomially reducible to a canonical planning problem, by replacing each operator o with $\mathcal{C}(o)$ copies of itself, since $\mathcal{C}(o)$ is usually small in optimal plans the sub-problem is often much easier than the full problem in practice.

In our problem solving the master MIP problem takes considerably longer than the SAT sub-problem. Hence we wish to consider how to speed up the LBBD solving process by using approximate answers to the master problem.

Solving the linear relaxation of the master MIP problem is considerably faster than solving it to integrality. Given an optimal solution to the linear relaxation, rounding up the variables will simply increase the operator counts available to

the sequencing sub-problem (compared to the MIP optimal). Since the SAT sub-problem only makes use of the upper bounds on operator counts, if it finds this relaxed sub-problem is unsatisfiable, it will generate a cut which removes the current LP optimum. If on the other hand it finds a solution to this relaxed sequencing sub-problem, then we have a feasible plan to the original problem. If this plan has the same cost as the (rounded up) lower-bound found by the LP, then we have optimally solved the planning problem. If the plan cost is more than the lower bound, this solution can be used to bound the search process: the MIP no longer needs to explore branches where the lower bound exceeds the cost of the best suboptimal plan found.

If a plan is found from a rounded-up operator count from an LP optimum, the MIP needs to branch before we can continue adding cuts. Importantly all modern MIP solvers provide user-specified cut generation and heuristic solution facilities via callbacks. We call our SAT sequencing procedure inside the python callback interface of Gurobi 5.6 (Gurobi Optimization, 2013) if both the cardinality and objective of the rounded-up operator count is within 20% of the linear count. We test for this to avoid generating SAT formulas for counts that are too far from the linear optimum, as the memory cost of adding layers to the formula is quite high. If we call the sequencing procedure, we either add a violated cut or a heuristic solution.

We do not modify Gurobi or MiniSAT's default branching behaviours, though it should be noted that state-of-the-art planning-as-satisfiability solvers use heuristics to simulate backward-chaining (Rintanen, 2012). We expect similar improvements could be possible by tailoring branching strategies in a MIP solver for the operator counting problem.

There is one caveat to using callbacks to add cuts to the MIP: it is impossible to lazily add bounds literals during the MIP's search. Consequently we do two things: pre-allocate bounds literals up to $[Y_o \geq 2]$; and add a relaxed version

of the cut where each absent literal $[Y_o \geq k]$ is replaced by Y_o/k. This happens in only a small percentage of cuts as even rounded-up operator counts rarely contain more than one of each operator. Cuts containing such Y_o/k terms are weaker than the normal landmarks, as for any value of Y_o:

$$[Y_o \geq k] \leq Y_o/k$$

However unless there is more than one missing bounds literal, this constraint still invalidates the current linear optimum. If a weakened constraint is generated that does not invalidate the current linear optimum, the MIP search is restarted after any weakened terms are replaced with the correct bounds literals.

This general approach of computing constraints violated by a close-to-optimal solution has many similarities with the improved landmark generation procedure of Haslum, Slaney, and Thiébaux (2012). In particular the incremental lower-bounding procedure h^{++} is very similar to our approach: both maintain a relaxation of the planning task (a delete-free problem with conjunctive conditions or an operator-counting MIP respectively); both incrementally refine their relaxation by finding flaws in the current relaxation's optimum (required conjunctive conditions or generalised landmarks respectively); and both exclusively find necessary properties of all plans, rather than focusing on extending promising plan prefixes, as in A^*-based planners.

5.6 Experiments

The main purpose of the experiments is to experimentally validate Theorem 5.1 and to investigate how sequencing performs on a wide array of near-optimal operator counts. We use IPC-2011 benchmarks, since our current prototype does not handle conditional effects required by IPC-2014 benchmarks, and since the

implementation is preliminary it scales poorly to the significantly larger IPC-2014 benchmarks.

From the IPC-2011 benchmarks, we omit the `floorplan` domain as *hpp*'s parser rejects the domain file, and the `tidybot` and `parking` domains as *OpSeq*'s Python base heuristic encoder exceeds the 1-hour time limit in more than 90% of instances.

The initial MIP master problem contains constraints from the dynamically-merged flow heuristic (Bonet and Briel, 2014) (including LM-Cut landmarks (Helmert and Domshlak, 2009)), and the h^+ base encoding of Imai and Fukunaga (2014).

Since sequencing considers all subsets of an operator count, rounding the linear optimum up makes for harder-to-sequence solutions than the true MIP optima most of the time. Nonetheless, Figure 5.3 shows that 99% of all the sequence calls take less than 1 second, although there is a significant long-tailed distribution: 0.01% of sequence calls took over 5 minutes. Some of this variance should be reduced by modifying the SAT solver's variable choice heuristics: the sequential counter implementation of $\leq_1$ which we use can be made significantly more robust using such techniques (Marques-Silva and Lynce, 2007).

We show breakdowns for the sequence times in each domain in the left-hand part of Table 5.1. The "Seqs" column shows the number of sequence calls made in all instances of a domain. The "Dom SeqTime" columns show the average sequence times for all sequence calls made in that domain. The "Inst SeqTime" columns show the arithmetic mean of instance averages. This biases the results towards the behaviour seen in larger instances where fewer sequence calls occur. For example, in `barman`, the overall average sequence time was 0.38 seconds, but, as should be expected, most of the nearly 80,000 sequence calls occurred in easier instances so the average sequence times for each instance, treating each instance's average as a single data point, show the average was 13 seconds. The first fifteen instances of `barman` have sub-second geometric mean sequence times, the largest

five however have geometric means between 4 and 47 seconds, but under 400 sequence calls were made.

We observe similar sub-second geometric means in most domains in both cases, though the arithmetic means are noticeably larger in larger instances. We believe there are two reasons for this: firstly the sequence times include generating the SAT formula, which often takes longer than solving in the first sequence call; and secondly, earlier calls have fewer learned clauses to aid solving.

To evaluate our technique as a dual (incremental lower-bounding) algorithm, we use a dual of the standard IPC quality score, where instead of dividing the best known plan cost by the plan cost found by the planner, the lower bound found by the planner is divided by the best lower bound proved by any planner. We compare against *hpp*, a comparable incremental lower bounding solver; and *SymBA*-2* the winner of the most recent IPC optimal track (Torralba et al., 2014). We use the versions of both planners submitted to the IPC 2014 optimal track.

As observed by Haslum, Slaney, and Thiébaux (2012), optimal planners using admissible heuristics in state-based (or symbolic) search can also be used as lower-bounding procedures by observing the smallest f value of nodes in the open list. Since *SymBA**, like most other planners, regularly prints its best known lower bound, it is trivial to obtain a lower bound from its output, even if it has not successfully solved the planning problem. Table 5.1 shows for each solver the number of instances solved optimally (column "C"); the number of instances where the solver found the best bound of any solver (column "="); and the dual quality score described above (column "Q").

We see from these results that *SymBA*-2* is extremely effective, beating both dual techniques in all three metrics in all but 5 domains, although in 4 of these 5 domains, *OpSeq* earns the best dual quality score, and in 3 domains even beats *SymBA** in coverage. *OpSeq* also beats the previous state-of-the-art in incremental lower bounding in 9 of the 11 domains investigated.

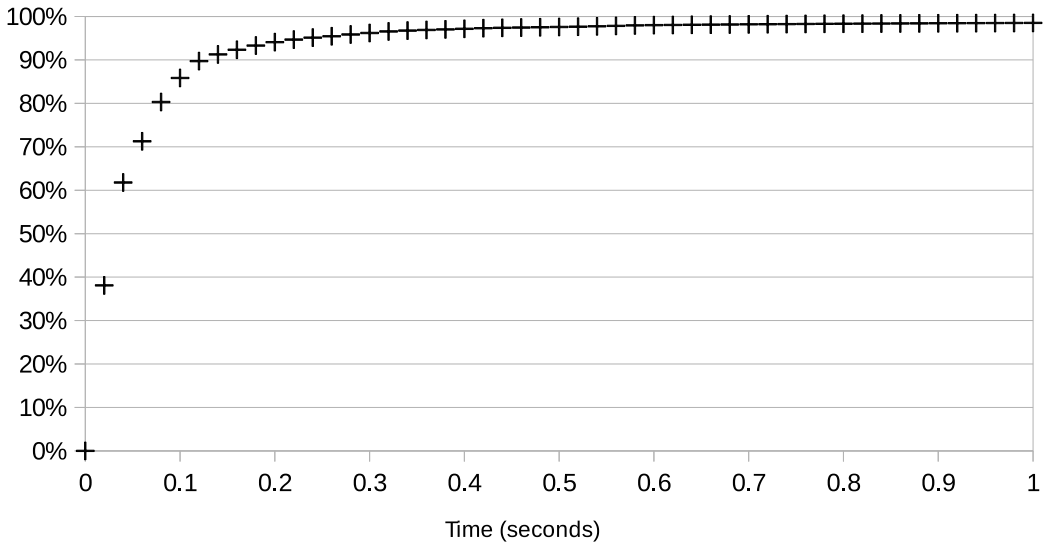


Figure 5.3: Cumulative Frequency of Sequence Times

We see from the quality scores in the "Q" columns, that even when *OpSeq* fails to solve it usually finds good lower bounds. This can be seen particularly in the `nomystery` and `transport` domains. For primal techniques it is far quicker to find a plan than to prove that a plan is optimal. Similarly for a dual technique it is much quicker to find h^* than it is to prove that we have found it by finding a plan.

Combinations of primal and dual techniques are among the most effective optimisation techniques: commercial mixed integer programming solvers use both lower bounding from the linear relaxation and sophisticated primal heuristics to find solutions close to the linear optimum. It could be argued that this is to some extent what *SymBA** does too: it interleaves spending time on improving abstractions (dual bounds) with searching in the original search space (Edelkamp, Kissmann, and Torralba, 2012).

Benchmark	Seqs	Dom SeqTime		Inst SeqTime		*OpSeq*			*Hpp*			*SymBA*-2*		
		Arith	Geom	Arith	Geom	C	=	Q	C	=	Q	C	=	Q
barman	79392	0.38	0.06	13.06	4.84	0	0	9.37	0	0	9.14	**11**	**20**	**20.00**
elevators	7437	2.55	0.09	2.08	0.26	11	11	19.38	0	0	16.47	**19**	**20**	**20.00**
nomystery	8761	0.04	0.01	0.25	0.03	5	10	18.33	5	8	8.00	**15**	**18**	**19.82**
openstacks	1655	11.21	0.12	63.19	1.60	0	0	5.52	0	0	5.52	**20**	**20**	**20.00**
parcprinter	102	1.65	0.09	2.38	0.19	**20**	**20**	**20.00**	**20**	**20**	**20.00**	17	17	18.63
pegsol	91466	0.45	0.02	13.53	0.12	2	5	15.97	0	0	12.43	**19**	**20**	**20.00**
scanalyzer	57023	0.04	0.02	6.36	0.15	1	3	7.99	3	**14**	**18.93**	**9**	10	14.32
sokoban	121117	0.38	0.11	1.77	0.34	0	2	10.70	1	2	11.27	**20**	**20**	**20.00**
transport	5223	4.42	0.17	33.76	8.38	5	13	**19.47**	0	0	12.41	**11**	**14**	17.81
visitall	97	15.84	0.08	14.49	0.27	**14**	**20**	**20.00**	5	13	19.21	12	12	15.70
woodworking	160	11.83	0.61	11.57	0.97	**20**	**20**	**20.00**	18	18	19.95	19	19	19.74
Total	—	—	—	—	—	78	104	166.74	52	75	153.33	**172**	**189**	**206.02**

Table 5.1: Average sequence times, Coverage (C), Number of best bounds (=), and Dual quality scores (Q) for IPC-2011 sequential optimal track benchmarks. (1 hour time-out, 4GB memory limit)

5.7 Related Work

We have discussed in several places the relationship between our work and h^{++}. Counter-example guided Cartesian abstraction refinement (Seipp and Helmert, 2013), which incrementally refines abstractions, rather than landmarks or a delete relaxation, could also be considered an incremental lower-bounding technique.

The only other approach using SAT-based planning techniques in cost-optimal planning that the authors are aware of uses MaxSAT combined with a SAT encoding of the delete relaxation (Robinson, Gretton, Pham, and Sattar, 2010). SAT planning has also been used to generate upper bounds to improve performance of state-based search (Robinson, Gretton, and Pham, 2008).

There have also been a number of "Optimal" SAT encodings, which find cost-optimal plans only when those plans happen to have the same makespan as the makespan-optimal plan (Chen, Lu, and Huang, 2008; Giunchiglia and Maratea, 2010). However this is clearly not the same as true cost-optimal planning.

Other generalisations of landmarks have been proposed including "multivalued landmarks" (Zhang et al., 2013), which identify operator sets which must collectively be executed more than once: $\sum Y_o \geq k$. These can be encoded in our generalised landmarks using an artificial operator to group the operators in the landmark (just as for the total operator count) leading to a constraint like $[\sum Y_o \geq k] \geq 1$; or directly as set of $O(\binom{N}{k-1})$ "standard" generalised landmarks.

There exist other heuristics bounded only by h^*, including many heuristics enhanced by the P^C and P^C_{ce} compilations (Haslum, Slaney, and Thiébaux, 2012a; Keyder, Hoffmann, and Haslum, 2012) and abstraction heuristics such as merge-and-shrink (Helmert, Haslum, et al., 2014). Merge-and-shrink like most heuristics in optimal planning provides only a lower bound to guide search; whereas if an optimal operator count is sequenced successfully, the planning problem is solved. Similar behaviour is also seen in the P^C and P^C_{ce} compilations: when no flaws can

be extracted the planning problem is solved optimally.

There are more sophisticated planning-as-SAT encodings that we could have added our counting constraint to, in particular the $\exists$-step and $\forall$-step encodings (Rintanen, Heljanko, and Niemelä, 2006; Wehrle and Rintanen, 2007), and the SASE encoding (Huang, Chen, and Zhang, 2012). However the absence of a tighter upper bound than simply the total number of operators required to refute any possible sequence of an operator count made the core advantage of these encodings less obvious. It would be interesting to compare these base encodings with ours, and a theorem giving such an upper bound would likely be an important breakthrough for the LBBD approach to planning.

5.8 Conclusions and Further Work

We have defined a simple generalisation of landmarks which allows the encoding of admissible heuristics upper-bounded only by h^*. We also introduce a SAT-based, complete algorithm for generating a generalised landmark violated by a given operator count which is usually very fast. We experimentally confirmed that h^* can be computed using only this algorithm, and demonstrated that it outperforms the previous state-of-the-art in incremental lower bounding: h^{++}.

Our approach can usually generate violated constraints from solutions to the linear relaxation of an operator count heuristic, rather than the NP-hard MIP. Importantly, if such a cut can be generated, it is guaranteed to invalidate the current linear optimum, and the current rounded solution, ensuring such heuristically generated cuts are relevant. This is in contrast to a similar improvement to a complete algorithm for generating traditional landmarks (Haslum, Slaney, and Thiébaux, 2012b), which relies on approximate hitting sets, with no guarantee that the generated landmark invalidates the current optimum hitting set, nor indeed that the landmark will change the next approximate hitting set generated.

This suggests a more traditional use for our generalised landmark generation algorithm: applying our algorithm to the delete relaxation can generate traditional landmarks, and the relative performance of this approach would be interesting to investigate.

There are other more conventional applications for generalised landmarks as well, such as pre-processing to generate an initial set of generalised landmarks which can then be used in an analogue of Incremental LM-Cut (Pommerening and Helmert, 2013). We expect this to provide improved heuristic guidance near the root of the search where it is most valuable. While we use a complete algorithm to generate landmarks, there is an obvious fast but incomplete algorithm obtained by simply terminating early when long-tailed behaviour is observed.

Such an approach could also potentially be used as a kind of look-ahead in an optimal planner: if an operator counting heuristic returns a sufficiently small operator count, our approach could test if the operator count is sequenceable, and if so, terminate the search early with an optimal plan.

We plan to investigate an extension to our approach which explains the states in which generalised landmarks apply, such that landmarks generated in the initial state can be easily re-used in a state-based search when they become applicable again.

However we chose to explore the more novel LBBD approach to planning using generalised landmarks in the hope that this decomposition between counting and sequencing will lead to cost-optimal planning algorithms capable of handling richer constraints such as numeric state variables, resources and scheduling constraints. These ideas have been extensively addressed, including techniques taking advantage of SMT (Nareyek et al., 2005; Hoffmann et al., 2007; Gregory et al., 2012).

We believe this approach is interesting and promising because it allows a principled interaction between state-of-the-art heuristics and explanation-based com-

binatorial search approaches including SAT, SMT and LCG. Any constraint capable of explaining its inferences can be added to the SAT sub-problem, potentially allowing direct integration of cost-optimal planning with SMT and state-of-the-art scheduling approaches based on constraint programming with lazy clause generation. This means that, by extending the approach we present, we should be able to solve similar problems to Planning Modulo Theories (Gregory et al., 2012) by taking advantage of the extensive range of *existing* theories and constraints *already implemented* by the SMT and constraint programming communities.

Chapter 6
Conflict-Directed Heuristic Learning

In this chapter we introduce Conflict-Directed Heuristic Learning, which incrementally learns a near-perfect heuristic using an IDA-like depth-first state-space search.*

6.1 Introduction

There have been a number of recent algorithms developed which learn from conflicts encountered during search, notably PDR (Suda, 2014) and DFS-CL(Steinmetz and Hoffmann, 2016). Neither of these algorithms find cost-optimal plans, although PDR can be configured to produce minimum length plans.

These two algorithms differ slightly in the nature of conflicts they detect and can learn to avoid. Each search node in PDR (also called a "proof obligation") consists of a state in the state space and a count of actions.

Conflicts in PDR are detected when the goal is not achievable from the state in at most the specified number of actions. From such a conflict, PDR learns a clause and action count, blocking a set of states which cannot reach the goal in the specified number of actions. In contrast, search nodes in DFS-CL do not include any count of future actions and thus recognise conflict only when a state cannot reach the goal at all.

It appears at first glance that PDR could be trivially generalised by replacing the limit on the number of actions with a limit on the total action cost. How-

ever, PDR terminates when the clauses representing the set of states which cannot reach the goal are the same in two adjacent layers. This recognises a fixed point which occurs when adding actions is not increasing the reachable set of states. It is not obvious how to generalise this termination condition to the cost-optimal case.

The idea of limiting the total cost of actions used to reach the goal is however the same fundamental idea behind the IDA* algorithm (Korf, 1985). In this case conflicts are inaccuracies in the heuristic function identified when the minimal f value of a state's successors is greater than the f-value of that state. IDA* can be augmented with a so-called "transposition table", which stores improved heuristic estimates for previously expanded states (Reinefeld and Marsland, 1994).

IDA* with a transposition table eventually learns the perfect heuristic for reachable states on the optimal path, and despite not generalising this knowledge, the transposition table can still be used to avoid an exponential amount of work because the same state can occur on an exponential number of paths.

This chapter introduces "Conflict-Directed Heuristic Learning" (CDHL), which generalises IDA* by using regression to analyse why a state's heuristic value is too low and learning a clause that generalises to many states which may be seen in the future.

```
IDAStarTT(h, s, h_max)
    if(Goal(s))
        return h
    while(h(s) ≤ h_max)
        o := argmin_o(h(s[o]) + c(o)|o ∈ O)
        if(h(s[o]) + c(o) > h(s))
            h := UpdateH(h, s, h(s[o]) + c(o))
        else
            h := IDAStar(h, s[o], h(s) − c(o))
            if(h(s[o]) + c(o) ≤ h_max)
                return h
    return h
```

Figure 6.1: Pseudo-code for learning IDA* (including IDA* with a transposition table and CDHL)

6.2 Preliminaries

***SAS*$^+$ planning** A SAS^+ planning task is a tuple $\langle V, O, s_0, s_*, c\rangle$ where V is a set of finite domain state variables, O is a set of operators, s_0 is a full assignment of each variable to one of its values representing the initial state, and s_* is a partial assignment of some subset of V representing the goal states. Finally c is a function $O \to \mathbb{N}^+$ that assigns a non-negative cost to each operator.

Each variable $X \in V$ has a domain $D(X)$, we sometimes abuse notation and write $X = x \in V$ which should be read $X \in V \wedge x \in D(X)$. Each operator o has a set of preconditions $pre(o)$ which is a partial assignment representing the preconditions of that operator, and a set of postconditions $post(o)$ which is a partial assignment representing the effects of the operator. Producers, $prod(X = x) = \{o \mid X = x \notin pre(o) \wedge X = x \in post(o)\}$ are the operators which cause X=x to become true. Note that for simplicity, we do not distinguish between preconditions and prevail conditions in this chapter.

A state s in the search space is a full assignment of every variable to a value. State s is said to satisfy a partial assignment F if all assignments in F are also in s,

i.e. $X{=}x \in F \Rightarrow X{=}x \in s$. A state is said to be a goal state if it satisfies the partial assignment s_*.

An operator $o \in O$ is applicable in s if s satisfies the partial assignment $pre(o)$. If o is applicable in state s, applying o yields a new state $s[o]$ which is the same as s except that all assignments $X{=}x \in post(o)$ replace any assignment to X.

A plan π is a sequence of operators $o_1, \cdots o_n$ such that o_1 is applicable in s_0, each subsequent operator is applicable in the state resulting from applying the previous operators in sequence, and the final state, $s_0[o_1, \cdots, o_n]$, satisfies s_*. An optimal plan has the minimum sum of operator costs of all plans, a SAS^+ planning task may have many optimal plans.

STRIPS planning is an equivalent formalism to SAS^+ which represents a planning problem as a tuple $\langle F, O, I, G \rangle$, where F is the set of facts that can become true in some state in the state space, I and G are subsets of F representing the initial and goal states respectively. While the initial state is fully specified, any state s such that $s \supseteq G$ is a goal state.

The set of operators, O, is defined by four functions, $pre(o)$, $add(o)$ and $del(o)$, each of which return a subset of F representing, respectively: the minimum set of facts that must be true to apply o; the facts that are added to the state after o was applied; and the facts deleted from the state after o was applied.

From this definition it follows that any state having strictly more facts than another cannot be further from the goal. Indeed, in STRIPS planning especially, it is what is **not** true in a state that defines its heuristic value. The "Clausal Heuristics" we will define in later sections of this chapter exploit this observation.

SAS^+ problems can be transformed fairly straightforwardly into STRIPS problems by defining $F = \{[X = x] | X \in V, x \in D(V)\}$. We also exploit the isomorphism between subsets of finite sets and assignments to sets of boolean variables to model STRIPS states as partial models of boolean formulae which are then

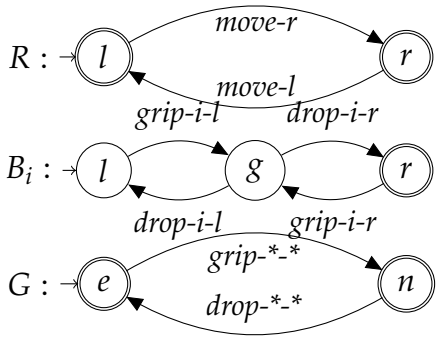


Figure 6.2: Domain Transition Graphs in gripper.

extended or refuted by a satisfiability problem solver.

The satisfiability problem is the NP-complete problem of trying to find an assignment to a set of boolean variables such that the assignment is consistent with an arbitrary boolean formula. Throughout this chapter we will assume that any formula is in conjunctive normal form (CNF): meaning the formula is a conjunction of disjunctions (clauses).

Note that a formula of the form

$$p_1 \wedge \cdots \wedge p_n \Rightarrow q_1 \vee \cdots \vee q_n$$

is equivalent to the clause:

$$\neg p_1 \vee \cdots \vee \neg p_n \vee q_1 \vee \cdots \vee q_n$$

by transformation using De Morgan's laws.

Bounds literals are boolean literals of the form $[i \geq k]$ (encoding the fact that the variable i takes a value no less than the constant k). Bounds literals can encode integers as a set of boolean variables linear in the integer's range. In this chapter

all integer variables encoded this way have the property that assigning a larger value allows a strictly larger set of models of the formula. Consequently we only ever care about the lower bound of integer variables, and the set of bounds literals can be theoretically infinite in size, and we merely instantiate the ones that participate in some constraint. We can then safely assign the variable the smallest k such that the formula is not provably unsatisfiable.

Potential heuristics map each atomic fact in a domain to a (possibly negative) cost which can be summed to find an admissible heuristic estimate for the state:

$$h(s) = \sum_{f \in s} w(f)$$

In our gripper example where actions all have a cost of 1, an admissible potential heuristic could have $w([B_i = l]) = 2$, and $w([B_i = h]) = 1$, and all other weights are zero.

6.3 Reduced-cost

In order to explain the intuition behind the introduced algorithm and to facilitate proofs we introduce the notion of reduced cost of operators. Reduced-cost is closely related to consistency, and measures the increase in f value caused by applying an operator in a specific state.

Using our example potential heuristic from section 6.2, the "move-r" action would have a reduced-cost of 1 in the initial state as applying this action does not reduce the heuristic value. However "grip-1-l" and "grip-2-l" would both have reduced costs of zero because they decrease the heuristic value by the same amount as their cost.

Definition 6.1 (Reduced cost)**.** *The reduced cost of an operator o in state s with respect*

to a heuristic h is defined to be:

$$\gamma(h,s,o) = \begin{cases} h(s[o]) + c(o) - h(s) & \textit{if } pre(o) \subseteq s \\ \infty & \textit{otherwise} \end{cases}$$

From this definition we can see that if h is consistent, no operator will have negative reduced cost in any state.

Iterations of IDA* with a consistent heuristic can be seen as a search for a solution considering only operators having zero reduced cost.[1] Subsequent iterations then use a new heuristic with an increased value for $h(s_0)$. IDA* is often enhanced with a so-called transposition table, in which case it learns a new heuristic with an increased value for each state it backtracks over.

In the pseudo-code from figure 6.1, this is equivalent to UpdateH(h,s,v) returning h' where:

$$h'(s') = \begin{cases} v & \text{if } s' = s \\ h(s) & \text{otherwise} \end{cases}$$

If we have some insight into the algebraic structure of h, it may be possible to explain why a state has no zero-reduced-cost operators and encode this in a new heuristic in such a way that it will not only increase the value for the state being pruned, but many future states.

To this end we introduce a new class of heuristics, "Clausal heuristics", which are highly amenable to explanation, and incrementally learning new features. This clausal representation is not necessarily the optimal one for learning succinct, accurate heuristics, but is possibly the simplest for proving why the technique works and exploring the relationship with related learning techniques in

[1] Any admissible heuristic can be made consistent by maximising over that state's f-value and that of its parent.

SAT and CP. We briefly discuss alternatives in section 6.7.

6.4 Clausal heuristics

In our gripper example, the following clauses are obviously true of all states:

$$\neg[B_1 = r] \Rightarrow [h \geq 1] \wedge \neg[B_2 = r] \Rightarrow [h \geq 1]$$

Finding the minimum value of h consistent with this particular SAT formula for a state assigns a cost-estimate of one to all non-goal states, and zero to any goal. However more complex formulae can encode more powerful heuristics.

For example, if a ball is neither in the right room, nor in the robot's hand, we know we must be at least 3 moves from achieving that goal: the robot must pick-up the ball; move into the right room; and finally drop it. This could be added to a clausal heuristic as the clause:

$$\neg[B_i = r] \wedge \neg[B_i = g] \Rightarrow [h \geq 3]$$

The focus of this section is on how to automatically derive such clauses from the conflicts encountered in a IDAStarTT-like search.

Definition 6.2 (Simple clausal heuristics)**.** *A simple clausal heuristic consist of a conjunction of boolean clauses, C each of the form:*

$$\neg f_x \wedge \cdots \wedge \neg f_z \Rightarrow [h \geq k_i]$$

where each f_i is a fluent in the state space, and k_i is some constant.

h can be evaluated in a state s as

$$h(s) = min(k|C \wedge \neg[h \geq k+1] \wedge \bigwedge_{f \in F \setminus s} \neg f)$$

An important property of a clausal heuristic is that only one clause in C is actually ever necessary to obtain the same heuristic estimate for any particular state. We refer to that clause as the *explanation* for the bound.

$$e_h(s) = \text{oneof}(c|c \in C \wedge \text{UNSAT}(\neg[h \geq h(s)-1] \wedge c \wedge \bigwedge_{f \in F \setminus s} \neg f))$$

any deterministic function can be used for "oneof".

Because of this, evaluating a clausal heuristic for a state is at worst polynomial in the number of clauses, even though we are using a satisfiability solver. This is because the solver does not need to make any decisions (except for the value of h) as they are all fixed by the state being evaluated.

A clausal heuristic can be used to construct an abstract state space by mapping each state to its explanation. An operator o is applicable in an abstract state $\neg f_1 \wedge \cdots \neg f_n \Rightarrow [h \geq p]$ if $pre(o) \cap \{f_1 \cdots f_n\} = \emptyset$. This is a useful way to understand and visualise the updates made to heuristics.

6.4.1 Integrating successor-generation

In our IDAStarTT pseudo-code in figure 6.1, we choose to apply an operator with minimal reduced cost. A naive implementation of this would need to evaluate the heuristic many times, and a lot of these evaluations would be redundant.

To simplify the integration of the successor generator, we define some arbitrary strict total ordering $\prec$ on operators, and enhance our heuristic to also return the minimum operator with zero reduced-cost.[2]

[2]In our experiments we sort first by maximum cost, then disambiguate arbitrarily. This causes

Clauses in our heuristics can now take the form:

$$\neg f_x \wedge \cdots \wedge \neg f_z \Rightarrow [h \geq w_i] \vee [o \succeq x_i]$$

remembering that $[o \succeq ||O|| + 1] \equiv$ False, and $[o \succeq min(O)] \equiv$ True.

From our combined heuristic and successor-generator, the next action to attempt in a state s, denoted succ(s), is defined as:

$$\text{succ}(s) = \min(q | q \in O \wedge C \wedge \neg[h \geq h(s) + 1] \wedge [o \succeq q] \wedge \bigwedge_{f \in F \setminus s} \neg f)$$

The principal advantage of defining a total order on actions is that it enables a single additional clause per state on the stack to prune all of the paths that have already been explored. This makes incrementally improving the heuristic-estimate of a state significantly easier, and simplifies garbage collection of unhelpful clauses without sacrificing completeness.

This single explanation is:

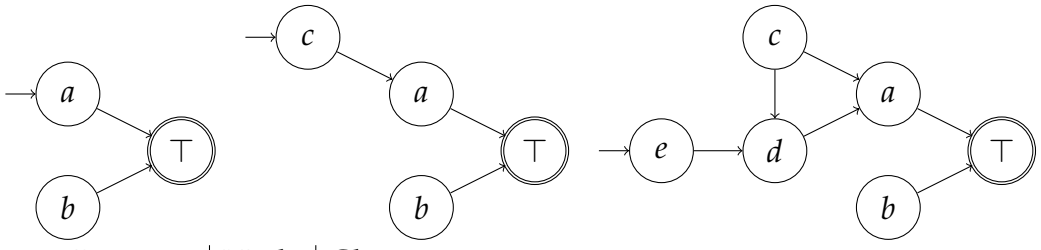
$$\begin{aligned} e_{succ}(s) = \operatorname*{argmin}_c (\{q | \\ & q \in O \wedge c \in C \wedge \\ & \text{UNSAT}(\neg[o \succeq q] \wedge \neg[h \geq h(s) + 1] \wedge c \wedge \bigwedge_{f \in F \setminus s} \neg f)\}) \end{aligned}$$

6.4.2 Incrementally refining clausal heuristics

The key idea in refining these heuristics is to prune each possible successor in a generalisable way by identifying a set of states where applying that action will always lead to the same clause applying.

Consider our gripper example. A simple total order would be lexicographical

the depth-first search to expand children with smaller h-values first.



Iteration	Node	Clause
0	a	$\neg[B_1 = r] \Rightarrow [h \geq 1]$
0	b	$\neg[B_2 = r] \Rightarrow [h \geq 1]$
1	c	$\neg[B_1 = r] \wedge \neg[B_1 = g] \Rightarrow [h \geq 2]$
2	d	$\neg[B_1 = r] \wedge \neg[R = r] \Rightarrow [h \geq 2]$
2	e	$\neg[B_1 = r] \wedge \neg[B_1 = g] \wedge \neg[R = r] \Rightarrow [h \geq 3]$

Figure 6.3: Improving a clausal heuristic for gripper

in the name of the operators. If we start with the blind heuristic figure 6.3 shows several iterations of improvements to that heuristic. Visualised by transforming the heuristic into an abstraction.

Improving a clausal heuristic is possible exactly when either:

- succ(s) operator is not applicable in s; or
- succ(s) has positive reduced cost in s.

To achieve this update several steps occur: Initially the clausal heuristic consists of the two clauses encoding the blind heuristic:

$$\neg[B_1 = r] \Rightarrow [h \geq 1] \wedge \neg[B_2 = r] \Rightarrow [h \geq 1]$$

This heuristic can be visualised as the first automaton if figure 6.3, corresponding to iteration 0. Unfortunately edges in these automata cannot necessarily be mapped directly to operators, but should be read as "there exists an operator that connects these two abstract states"..

IDAStarTT evaluates the heuristic value for the initial state as 1, and arbitrarily picks the first of the above clauses as the explanation. It then attempts to

compute the minimum reduced cost operator by trying the operators in order. The lexicographically minimal operator is "drop-1-l", which is both inapplicable in the initial state, and has a positive reduced cost in any state because the same explanation clause would apply in the state after applying the operator.

We can add either

$$\neg[B_1 = r] \wedge \neg[B_1 = g] \Rightarrow [h \geq \infty] \vee [o \succeq \textit{drop-1-r}]$$

or

$$\neg[B_1 = r] \wedge \neg[R = r] \Rightarrow [h \geq \infty] \vee [o \succeq \textit{drop-1-r}]$$

or

$$\neg[B_1 = r] \Rightarrow [h \geq 2] \vee [o \succeq \textit{drop-1-r}]$$

to the clausal heuristic to progress. Our implementation would choose the first of these, but this choice is arbitrary.

The first two clauses are correct because all actions lexicographically less than "drop-1-r" cannot be applied in any state where the left hand side of the implication holds. In general we can choose any precondition of the action being blocked to add to the left hand side of the implication. The last clause holds because the heuristic estimate for any states where ball 1 is not already in the right room is obviously not decreased by any action less than "drop-1-r". Note that even though no state's h-value would decrease after applying this operator, we cannot simply say

$$\top \Rightarrow [h \geq 2] \vee [o \succeq \textit{drop-1-r}]$$

as the goal may already be achieved. We must keep at least one reason that the goal is not yet achieved in the explanation clause.

Having added one of these clauses, we recompute the lexicographically-minimal operator consistent with $h(s) = 1$. Any of the above clauses makes $o = \textit{drop-1-l}$

inconsistent with $h = 1$, however we will assume the first of these clauses is added. The next smallest operator is "drop-1-r", which again is inapplicable.

$$\neg[B_1 = r] \wedge \neg[B_1 = g] \Rightarrow [h \geq \infty] \vee [o \succeq \textit{drop-2-l}]$$

This clause strictly dominates the previously learned clause, meaning that the previous clause can be deleted with no loss of information.

The next seven actions: "drop-2-l", "drop-2-r", "grip-1-l", "grip-1-r", "grip-2-l", "grip-2-r", and "move-l-r" all have positive reduced cost as $[B_1 = r]$ is still not achieved. Consequently we can learn:

$$\neg[B_1 = r] \wedge \neg[B_1 = g] \Rightarrow [h \geq 2] \vee [o \succeq \textit{move-r-l}]$$

Each of the four clauses we learn from trying these actions strictly dominates its preceding clause, allowing us to delete them.

The next action, "move-r-l", also has positive reduced cost for the same reason, but is interesting as it is the lexicographically last operator, and so there is no next action so the $[o \succeq ?]$ term can be omitted as it is equivalent to false, allowing us to learn

$$\neg[B_1 = r] \wedge \neg[B_1 = g] \Rightarrow [h \geq 2]$$

Which is the new explanation for the heuristic-estimate of the initial state $h(s_0) = 2$. At this point, the heuristic can be visualised as the second automaton in figure 6.3, corresponding to iteration 1.

As yet we have not done any search, merely having reasoned about the heuristic and the initial state to find properties of that state that prevent us from finding zero-reduced-cost successors.

Having increased the heuristic estimate at the root from 1 to 2, we enter the second major iteration of IDA*. We now attempt to compute the minimum reduced-

cost action again. We can start with "drop-2-l" because of the previously learned clause:

$$\neg[B_1 = r] \wedge \neg[B_1 = g] \Rightarrow [h \geq \infty] \vee [o \succeq \textit{drop-2-l}]$$

Neither "drop-2-l" nor "drop-2-r" add $[B_1 = r]$ or $[B_1 = g]$, and thus cannot reduce the heuristic value of the state, so we learn:

$$\neg[B_1 = r] \wedge \neg[R = r] \Rightarrow [h \geq 3] \vee [o \succeq \textit{grip-1-l}]$$

After applying "grip-1-l" however, the h value of the resulting state has decreased by 1, meaning the operator had zero reduced cost. We can now recurse after applying this operator with a maximum h-value of 1.

We now search for a zero-reduced cost operator in this new state starting with "drop-1-l", and learn that it increases the heuristic value to 2.

$$\neg[B_1 = r] \wedge \neg[R = r] \Rightarrow [h \geq 3] \vee [o \succeq \textit{drop-1-r}]$$

We then find that "drop-1-r" is inapplicable in this state because $[R = r]$ does not hold, and thus learn:

$$\neg[B_1 = r] \wedge \neg[R = r] \Rightarrow [h \geq 3] \vee [o \succeq \textit{drop-2-l}]$$

None of the remaining operators add $[B_1 = r]$ and so cannot change the heuristic value, and must have positive reduced cost. The cost after each of these operators is however only 1, so we reduce the "next heuristic value" to 2 and learn:

$$\neg[B_1 = r] \wedge \neg[R = r] \Rightarrow [h \geq 2]$$

This corresponds to node d in figure 6.3.

The recursive call of IDAStarTT returns as we have proved the goal is not

reachable with an h-budget of 1 after applying "grip-1-l".

$$\neg[B_1 = r] \wedge \neg[B_1 = g] \wedge \neg[R = r] \Rightarrow [h \geq 3] \vee [o \succeq \textit{grip-1-r}]$$

We now return to the top level of the stack and continue searching for zero-reduced-cost operators. None of the remaining operators add either $[B_1 = r]$ or $[B_1 = g]$ and therefore must have positive reduced cost at the initial state, so we can learn:

$$\neg[B_1 = r] \wedge \neg[B_1 = g] \wedge \neg[R = r] \Rightarrow [h \geq 3]$$

corresponding to node e in figure 6.3. At this point, the heuristic can be visualised as the third automaton in figure 6.3, corresponding to iteration 2.

This process of identifying and correcting flaws in the heuristic continues until a sequence of zero-reduced-cost operators is found connecting the initial and goal states.

Formally, we improve the heuristic by applying two rules in any order repeatedly until fixed-point:

1. When $\text{pre}(\text{succ}(s)) \not\subseteq s$ we learn a new clause $\phi \wedge p \Rightarrow [h \geq w_i] \vee [o \succeq \text{succ}(s) + 1]$ for some $p \in \text{pre}(\text{succ}(s))$

2. Alternatively when $\gamma(h, s, succ(s)) > 0$, we learn a new clause $(\phi_0 \Rightarrow [h \geq w_i] \vee [o \succeq \text{succ}(s) + 1]) \vee \mathcal{R}(e_h(s[succ(s)]), succ(s))$.

Here $\phi_0 \Rightarrow [h \geq w_i][o \succeq succ(s)] = e_{succ}(s)$; and $\mathcal{R}(\phi \Rightarrow [h \geq w_i], o)$ is the regression operator, which regresses a clause ϕ through a single action o, and returns a clause representing a property of the set of states S s.t. for all $\forall s' \in S$: $h(s'[o]) \geq w_i$.

$\mathcal{R}$ is defined as:

$$\mathcal{R}(\phi \Rightarrow [h \geq w_i], o) = \bigwedge(\neg f_i \vee f_i \in add(o)) \Rightarrow [h \geq w_i + cost(o)]$$

6.4.3 Integrating duplicate-detection

In order to handle zero-cost actions we must avoid expanding duplicate states on the current path. To do this we add one more type of variable to our clauses: $[\langle g, d\rangle \leq \langle y_i, z_i\rangle]$. Immediately before applying any zero-cost action, a clause of the form $\neg f_x \wedge \cdots \wedge \neg f_z \Rightarrow [\langle g, d\rangle \leq \langle g(s), d(s)+1\rangle]$ is added to the heuristic, where g represents the "g-value" of the node, and d the depth of the node being expanded.

Consider a variant of our gripper example where movement actions have zero cost. In the initial situation it would be possible to "move-r" then "move-l" and return to the same state. We could however add the clause:

$$\neg[R = r] \wedge \neg[B_1 = r] \wedge \neg[B_1 = g] \wedge \neg[B_2 = r] \wedge \neg[B_2 = g] \wedge \neg[G = n] \Rightarrow \\ [\langle g, d\rangle \leq \langle 0, 1\rangle] \vee [h \geq \infty]$$

Before expanding the initial state then when evaluating the heuristic after the cycle, the h-value of the node would be too high because $g = 0$ and $d = 3$ and IDAStarTT would backtrack.

This clause could then be regressed through the "move-l" operator in much the same way as usual. The only special handling is of the $[\langle g, d\rangle \leq \langle 0, 1\rangle]$ term, which is handled analogously to $[h \geq k]$ terms by decrementing g by the cost of the operator, and d by 1. Causing us to learn:

$$\phi \wedge \neg[B_1 = r] \wedge \neg[B_1 = g] \wedge \neg[B_2 = r] \wedge \neg[B_2 = g] \wedge \neg[G = n] \Rightarrow \\ [\langle g, d\rangle \leq \langle 0, 0\rangle] \vee [h \geq k] \vee [o \geq \textit{move-r}]$$

Assuming that the previous $e_{succ}(s_0[\textit{move-r}])$ was:

$$\phi \Rightarrow [h \geq k] \vee [o \geq \textit{move-l}]$$

Note that $[g \leq \langle 0, 0 \rangle] \equiv \mathit{False}$, as we count depth starting at 1. This means we have effectively learned that, to be allowed to move right, we must first have advanced the state of one of the balls in some way.

Definition 6.3 (Clausal heuristic)**.** *A clausal heuristic consists of a set of clauses of the general form:*

$$\neg f_x \wedge \cdots \wedge \neg f_z \Rightarrow [h \geq w_i] \vee [o \succeq x_i] \vee [\langle g, d \rangle \leq \langle y_i, z_i \rangle]$$

Note that $[h \geq \infty] \equiv [o \succeq max(O)] \equiv [\langle g, d \rangle \leq \langle 0, 0 \rangle] \equiv$ False, so all elements on the right of the implication are optional.

6.4.4 Garbage collection

As briefly noted earlier, only 3 clauses per state on the stack are necessary to guarantee completeness. As long as $e_h(s)$, $e_{succ}(s)$, and $e_g(s)$ are retained for all s on the current path, all other clauses can be removed, and CDHL will not explore the same path with the same f-bound again.

6.5 Lazily explaining arbitrary heuristics

Any heuristics' lower-bound in a particular state can be explained in terms of some subset of the fluents that do not hold in a state. In the worst case, the explanation can be exactly the false fluents: adding more facts can only increase the set of plans originating in that state, and thus cannot increase the distance to the goal. However more succinct explanations are sometimes possible than this naive encoding. All heuristics can then be translated to abstractions using at most one such explanation per reachable state in the state space.

Theorem 6.1. *Any consistent heuristic h encoded into some conjunction clauses of the*

form

$$\neg f_i \wedge \cdots \wedge \neg f_j \Rightarrow [h \geq h(s_i)]$$

for any reachable states s_i will define a consistent clausal heuristic.

Note that although we can encode any heuristic in this clausal abstraction formalism, it may require as many clauses as states in the state-space.

Potential heuristics are a particularly interesting class of explainable heuristic. These heuristics are typically defined in terms of SAS+ rather than STRIPS problems, however SAS+ problems are easily transformed into STRIPS. Given this transformation, the problem's fluents can be partitioned into disjoint mutually exclusive sets $F_0 \cdots F_n$.

In gripper, we may want to explain $h(s_0) = 4$. Using our potentials described in section 6.2, we can see that the variables R and G are irrelevant to the heuristic value and can be ignored entirely. We could therefore generate a clause:

$$\neg[B_1 = r] \wedge \neg[B_1 = h] \wedge \neg[B_2 = r] \wedge \neg[B_2 = h] \Rightarrow [h \geq 4]$$

Perhaps more interestingly, in the state immediately after "grip-1-l", we can generate a similar clause:

$$\neg[B_1 = r] \wedge \neg[B_2 = r] \wedge \neg[B_2 = h] \Rightarrow [h \geq 3]$$

Note that we omit the $[B_1 = l]$ literal as this has a higher weight than the true value of this variable: $[B_1 = h]$. If $[B_1 = l]$ were true, we would necessarily be in a worse state, so the assertion that $h(s) \geq 3$ would still hold.

In general, we need only include a literal in a potential heuristic explanation if that fluent being true in a state is strictly better than the value of the same variable that is actually true in the current state.

Formally, given a state s, a potential heuristic h with weights $w([X_i = x_j])$ for

each fluent $[X_i = x_j] \in F$, $h(s)$ can be explained by $\phi \Rightarrow [h \geq h(s)]$ where

$$\phi = \bigwedge_{[X_i=x_j]\in F} (\exists k.[X_i = x_k] \in s \wedge w([X_i = x_j]) < w([X_i = x_k]) \Rightarrow \neg[X_i = x_j])$$

The right-hand side of each implication can be evaluated completely at explanation time, reducing ϕ to a simple conjunction of negated fluents.

6.5.1 Emulating IDA*

If when learning new clauses we replace ϕ with exactly the set of false fluents we explicitly turn-off generalisation, making the resulting algorithm equivalent to IDA* with a transposition table. If, further, we immediately discard all non-essential clauses, we also forget the learned heuristic values once they are popped from the stack, effectively making the algorithm a variant of IDA*. Our implementation uses a hash-table to store any clause where ϕ entails exactly one syntactic state, making comparison to typical transposition-table implementations more fair.

6.6 Experiments

In table 6.1 we compare our implementation in 3 modes: IDA*, IDA* with a transposition-table, and CDHL. All 3 started with a potential heuristic optimised to maximise the heuristic estimate at the initial state. We can see from this table that the generalised explanations do confer some benefit. In particular, we can see from figure 6.4 that the number of expanded nodes is often greatly reduced, however this comes at the cost of increased processing time per node, sometimes outweighing the benefit as seen in figure 6.5. There exists one instance where IDA*-TT uses fewer nodes, we presume this is due to garbage collection.

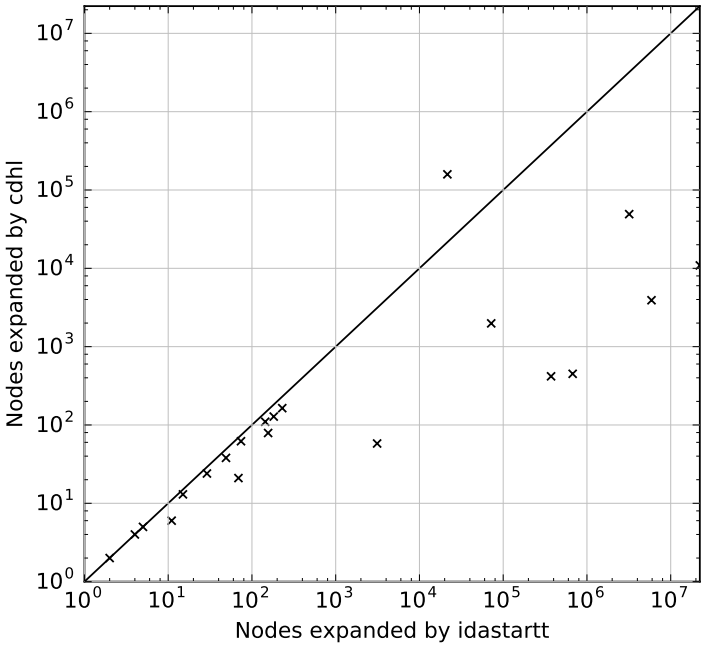


Figure 6.4: Nodes expanded in search for CDHL and IDA* with a transposition table

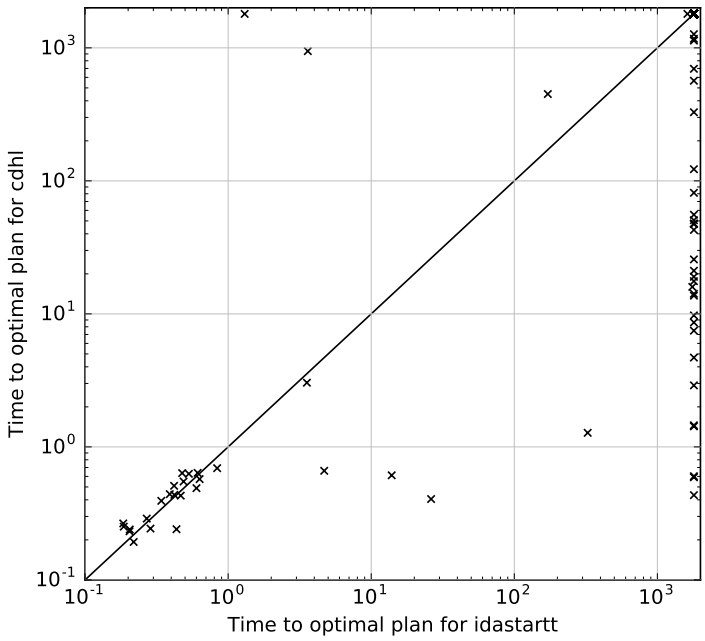


Figure 6.5: Time to optimal solution for CDHL and IDA* with a transposition table

Domain	IDA*	IDA*-TT	CDHL
barman	**0**	**0**	**0**
elevators	0	**7**	**7**
floortile	**0**	**0**	**0**
nomystery	4	6	**7**
openstacks	0	5	**7**
parcprinter	2	**5**	2
parking	0	**1**	0
pegsol	0	**6**	2
scanalyzer	**3**	**3**	2
sokoban	0	**3**	2
tidybot	1	3	**11**
transport	**0**	**0**	**0**
visitall	14	**15**	14
woodworking	0	1	**4**
Total	24	55	**58**

Table 6.1: Coverage of IDA* and CDHL on the 2011 IPC benchmarks. (Max time: 30 minutes, Max memory: 100 MB)

6.7 Conclusions and Future Work

Our results show that the number of expanded nodes can be reduced by several orders of magnitude compared to IDA*, some of this benefit can be achieved using a simpler (and much faster) transposition table, however reductions in nodes explored as significant as three orders of magnitude can be achieved by generalising the knowledge learned during search, even when starting with a state-of-the-art potential heuristic.

These reductions come at a significant increased processing time per node. We hypothesise that traditional watched-literal data-structures (Moskewicz et al., 2001) used to store clausal information may be poorly suited to this use: watched literals perform best when some literals are "cold" (never true) so their watch lists can become long at little cost to the search. However in planning, most fluents are both true and false on the optimal path to the goal. Other data-structures (perhaps based on decision diagrams or match-trees) may be better suited to this

use-pattern.

Additionally recent work by Pommerening, Helmert, and Bonet (2017) opens up the possibility of learning higher-dimensional potential heuristics which appear to often be perfect with quite small numbers of features.

Part III

Temporal Planning and Scheduling

Introduction to Part III

Temporal planning and scheduling are closely related problems with a blurry boundary. The first algorithm we present in this part of the thesis is Automatic Logic-Based Benders Decomposition. This algorithm tackles alternative scheduling problems, which are part-way between temporal planning and scheduling problems.

Traditional scheduling problems require a solution to choose **when** to perform a fixed set of actions; alternative scheduling adds to this some choice of **which** actions to perform; temporal planning then further adds the choice of **how many** actions to perform.

We also propose a theoretical extension to this approach, Operator Scheduling, which would enable us to take that final step.

Chapter 7
Automatic Logic-Based Benders Decomposition with MiniZinc

This chapter introduces a general approach to applying Logic-Based Benders Decomposition to an arbitrary constraint program. We evaluate this approach on variations of alternative scheduling problems, and show how it might be extended to tackle more general temporal planning problems.[1]

7.1 Introduction

Logic-based Benders decomposition is among the most effective approaches for finding optimal solutions to complex configuration and scheduling tasks, frequently two or three orders of magnitude faster than pure MIP or CP approaches (Hooker and Ottosson, 2003).

The essence of logic-based Benders decomposition is to take the problem P, and derive a *relaxed master* P_M (typically a MIP) which relaxes or omits some constraints in P, and a set of independent sub-problems $P_1, \ldots, P_k$ such that $P \Leftrightarrow P_M \wedge P_1 \wedge \cdots \wedge P_k$. The master solves P_M and the solution μ is checked by each sub-problem solver for P_i. If μ does not satisfy P_i a cut is added to the master to eliminate μ and other solutions which will not satisfy P_i for the same reason.

[1] The research presented in this chapter (except section 7.6) was published in Toby O. Davies, Graeme Gange, and Peter J. Stuckey (2017). "Automatic Logic-Based Benders Decomposition with MiniZinc". In: *AAAI Conference on Artificial Intelligence (AAAI)*.

However, designing a concrete instantiation of this framework is typically nontrivial. The first difficulty is in choosing the relaxation P_M. If the master omits important constraints entirely, the candidate solutions are too optimistic, and the method converges slowly. Conversely, if the constraint is not substantially relaxed, solving the master becomes unmanageable (effectively regressing to a pure MIP approach).

A second issue is the extraction of feasibility cuts from infeasible sub-problems. Logic-based Benders cuts are typically couched in terms of the *inference dual* (Hooker and Ottosson, 2003). However typical sub-problem solvers (in particular, classical CP solvers) do not provide sufficient information to reconstruct a compact justification of failure, so in practice cuts are derived by exploiting the independence of the sub-problems and knowledge of their structure (Ciré, Coban, and Hooker, 2013).

In this chapter, we present a fully automatic approach for solving constrained optimisation problems via logic-based Benders decomposition. The automated decomposition diverges from typical instances of logic-based Benders decomposition in a key respect: it constructs only a single complete 'sub-problem'. Rather than explicitly decomposing our problem into independent sub-problems (either manually or heuristically), we instead rely on the conflict analysis capabilities of lazy clause generation solvers to identify relevant subsystems.

Though this approach makes the sub-problem considerably more difficult, it offers several advantages. Unlike classical LBBD, it can cope with subsystems which are not fully disjoint. There is no need to design problem- or objective-specific cuts; cut derivation is entirely generic. And while generating cuts, the sub-problem solver also acts as a primal heuristic.

This formulation reveals an interesting duality. From the perspective of the MIP solver, this is an instance of logic based Benders decomposition where the sub-problem solver doubles as a primal heuristic. But from the perspective of the

CP solver, our framework is an instance of large neighbourhood search (LNS) (Pisinger and Ropke, 2010) – the MIP proposes promising candidate regions, which the CP solver progressively explores and expands – in which the neighbourhood selection heuristic also supplies valid lower bounds. We will refer to the search space implied by a set of assumptions as a neighbourhood throughout this chapter.

The contributions of this chapter are as follows:

- A variant of logic-based Benders decomposition which reveals a duality between LBBD and large-neighbourhood search (LNS)
- An automated approach for applying LBBD techniques to arbitrary constraint models
- A simple way of integrating traditional CP optimisation-as-repeated-satisfaction into the LBBD framework.

7.2 Preliminaries

7.2.1 Constraint Programming and Lazy Clause Generation

Constraint programming (CP) systems solve a problem of the form $\exists V.D \wedge C$, D is a conjunction of unary constraints (*a domain*) constraining each integer variable[2] $v \in V$ to take a finite set of values, and C is an arbitrary collection of constraints on variables V. A domain D that entails a single value for each variable in V is a *valuation* which we denote by μ. We use notation $\mu(v)$ to return the value of variable v given by valuation μ. Each constraint $c \in C$ is implemented using a propagator which given a domain D infers new unary constraints d which must hold, i.e. $D \wedge c \Rightarrow d$. The domain is then updated to $D' = D \wedge d$. Failure is detected if $d =$ `false`. Success is detected if D' is a valuation (under reasonable

[2]Here we treat Boolean variables as 0/1 integers.

assumptions about the strength of propagator inferences). Otherwise when no further inference is possible, the system guesses a new unary constraint l, and recursively considers the two systems $\exists V.(D \wedge l) \wedge C$ and $\exists V.(D \wedge \neg l) \wedge C$.

A *lazy clause generation (LCG)* solver in addition tracks the reasons for its inferences. For each inference $D \wedge c \Rightarrow d$ it stores a *reason* clause $d_1 \wedge \cdots \wedge d_n \Rightarrow d$ which is a consequence of c, that explains the inference. When failure is detected it uses the reasons to construct a *nogood* by resolution which explains the failure. The nogood is then added to the solver to prevent the same failure re-occuring.

An LCG solver can be extended to support an *assumption interface*. Given a set of unary constraint assumptions μ the solver solves $\exists V.(D \wedge \mu) \wedge C$. If the solver determines the problem has no solution it can return a clause of the form $\mu_1 \wedge \cdots \wedge \mu_n \Rightarrow \texttt{false}$ where $\mu_i \in \mu$, that explains which assumptions were responsible for the failure, i.e. such that $\exists V.(D \wedge \mu_1 \wedge \cdots \wedge \mu_n) \wedge C$ is unsatisfiable.

7.2.2 The MINIZINC solver pipeline

MINIZINC (Nethercote et al., 2007) is a high-level declarative modelling language for constrained optimisation problems. Underlying solvers typically do not support MINIZINC directly. Instead, an instance of the high-level MINIZINC model is compiled down to a simpler FLATZINC. During this *flattening* step, existential quantification and complex Boolean structure are eliminated. Each solver provides a library of predicate definitions to control this flattening; global constraints are handled in one of three ways:

- If the solver provides a declaration but no definition, the constraint call is passed directly to the solver.

- If a definition for the predicate is provided, the definition is expanded (and recursively flattened).

- Otherwise, the default flattening is expanded.

This flattened model is then fed to the solver, paired with directives for formatting output. This architecture provides a uniform framework for allowing high-level model specifications while exploiting the heterogeneous capabilities of different solvers.

In Section 7.3, we shall demonstrate how to (mis-)use this framework to support a logic-based Benders approach.

7.2.3 Classical and Logic-based Benders Decompositions

Logic-based Benders decomposition (Hooker and Ottosson, 2003) replaces the linear programming dual used in classical Benders decomposition with the more general *inference dual* – the problem of inferring the tightest objective bound from a set of constraints, its solution being a proof of the optimal bound. This proof is then translated into a sound bounding function suitable for addition to the master – in the case of a MIP master problem, the optimality proof must be translated into one or more linear inequalities over variables occurring in the master. Unfortunately, most sub-problem solvers cannot provide information to reconstruct the inference dual, supplying only the primal solution. In this case, it is common to instead design specialised cuts based on problem structure, identifying some subset E of assignments to shared variables such that $f(E) \leq z$. In contrast, we use a clausal cut of the form: $E \Rightarrow false$ (or equivalently $E \Rightarrow [\![z > k]\!]$ in place of optimality cuts), these cuts are explained in more detail in section 7.3.1.

7.2.4 Cut strengthening and MUS construction

Without explicit dual information, the generated Benders cuts are quite coarse. These may then be *strengthened*, using the sub-problem solver as a feasibility ora-

cle – progressively discarding assumptions, and checking that the (now relaxed) sub-problem remains infeasible.

When the master assignment is viewed as a conjunction of propositions, this "cut strengthening" process corresponds exactly to minimal unsatisfiable subset (MUS) construction: given an unsatisfiable conjunction C of constraints, identify one or more minimal subsets $C' \subseteq C$ which preserve infeasibility.

MUS construction arises in various contexts, and has received particular attention in SAT and CP (Dershowitz, Hanna, and Nadel, 2006; Liffiton and Malik, 2013; Junker, 2004; Hemery et al., 2006). As in the LBBD case, SAT MUS construction algorithms use a decision procedure (here a SAT solver) as an oracle, repeatedly choosing $P \subset C$ and testing satisfiability of $C \setminus P$ until an MUS has been identified. These algorithms differ in their strategy for choosing subsets to test – sequential (Bakker et al., 1993), dichotomic (Felfernig, Schubert, and Zehentner, 2012) and geometric progression (Marques-Silva, Janota, and Belov, 2013) strategies have been proposed. One general refinement to these approaches, *clause-set refinement* (Dershowitz, Hanna, and Nadel, 2006), is directly applicable to the CP case.

Clause-set refinement exploits the capability of SAT solvers to return an unsatisfiable core. When a query $C \setminus P$ returns `UNSAT(`C'`)` (showing there is some MUS excluding P), MUS construction may simply replace $C \setminus P$ with C' before continuing.

Figure 7.1 illustrates a sequential approximate MUS algorithm, with clause-set refinement and resource limits. The algorithm maintains 3 sets of literals originating from the unsatisfiable subset C to be minimised, C contains clauses which we have not yet attempted to prove must be in the final MUS; M contains clauses which we have successfully proved must be in the final MUS; and U contains clauses which we have tested, but could not prove must belong in the MUS within the specified resource limit *lim*. At the end of each iteration, the set $C \cup M \cup U$ is

```
MUS-seq(C, lim)
    M := ∅; U := ∅
    while(∃c ∈ C)
        C := C \ c
        case(is-sat(M ∪ U ∪ C, lim))
            [SAT]:
                M := M ∪ {c};
            [UNSAT(C')]:
                C := C ∩ C'
                U := U ∩ C'
            [UNKNOWN]:
                U := U ∪ {c}
    return M ∪ U
```

Figure 7.1: Pseudo-code for sequential MUS construction with conflict-set refinement and resource limits. M contains those propositions definitely in the MUS, and U those which could not be eliminated. We may terminate this procedure at any time, returning $M \cup U \cup C$ as an unsatisfiable core.

a valid unsatisfiable subset.

The algorithm tests each clause c in sequence to see if it must be in the unsatisfiable subset by testing if $C \setminus c$ is still an unsatisfiable subset using *is-sat* as a satisfiability oracle. This procedure takes an argument *lim*, which limits the computational effort allowed to solve the SAT problem. *lim* could be a limit on runtime, number of propagations, or number of branches, we assume a limit on the number of branches in this chapter.

Because it has a resource limit, *is-sat* can return `UNKNOWN`, in addition to `SAT`, or `UNSAT`(X). If $C \setminus c$ is satisfiable, then c must be in the MUS. If we can prove that $C \setminus c$ is unsatisfiable, we know c is not a member of the MUS and can be discarded, additionally we can use clause-set refinement to intersect C and U with the newly identified unsatisfiable core. Finally if *is-sat* returns `UNKNOWN`, we assume that c is necessary for the formula to remain unsatisfiable and add c to U.

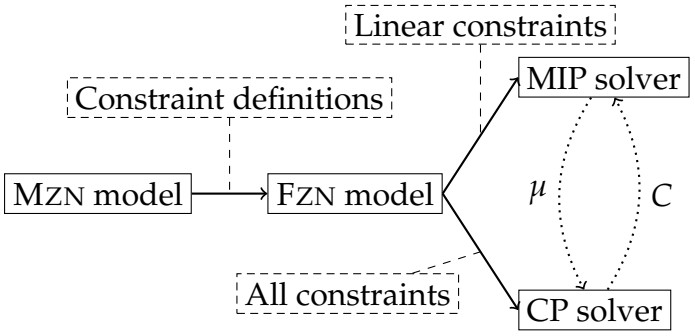


Figure 7.2: Architecture of the automatic Logic-based Benders Decomposition.

7.3 Automating Logic-based Benders Decomposition

The key idea behind this approach is quite simple, and is shown in Figure 7.2. During flattening, any constraints not supported by a solver are replaced by an equivalent (modulo introduced variables) solver-specific decomposition. If we discard any subset of these constraints, what remains is necessarily a valid relaxation.

The master problem M is constructed directly from the existing flattened model, retaining those constraints for which compact encodings exist – primarily *element* and (reified) *linear* constraints – and discarding the rest of the model.

Similarly, we do not attempt to identify independent sub-problems, or partition variables between master and sub-problem. The single "sub-problem" S consists of the full flattened model, and the master and sub-problem are solved over all variables appearing in any master constraint; and all variables respectively.

The high-level LBBD procedure is shown in Figure 7.3: we first solve the master M to optimality, then try to extend the partial solution μ to a full solution μ^* of the sub-problem S. We then extract and minimise one or more cuts C' from the sub-problem and re-solve the (now tighter) master. We use X to track literals that have appeared in any clause so far, and omit these from subsequent solves to guarantee cuts are independent. Note: μ, μ^*, C and C' are all sets of bounds

```
solve-lbbd(obj, M, S):
  μ* := ⊥
  while(solve-master(M) = SAT(μ))
    S := S ∧ [[obj ≥ μ(obj)]]
    case(solve-lcg(S, μ, μ*, ∞))
      [UNSAT(∅, μ*)]
        return(μ*)
      [UNSAT(C, μ*)]
        X := ∅
        while(solve-lcg(S, μ − X, μ*, lim) = UNSAT(C, μ*))
          (C', μ*) := minimise-cut(S, C, μ*, lim)
          M := M ∧ (C' ⇒ false)
          if(C' = ∅) return(μ*)
          M := M ∧ [[obj < μ*(obj)]]
          X := X ∪ C'
  return μ*
```

Figure 7.3: Pseudo-code for Logic-based Benders decomposition with a complete subproblem.

literals, and μ and μ^* are valuations.

The algorithm in 7.3 solves a constraint program using LBBD. It repeatedly solves the master MIP to optimality, and obtains a partial assignment μ to all variables that participate in some linear constraint. It then attempts to extend the partial assignment into a full assignment by solving the sub-problem using lazy clause generation.

The sub-problem solver *solve-lcg*(S, μ, μ^*, *lim*) takes the constraints S and assumptions μ, the current best solution μ^*, and a (possibly infinite) decision limit. It then searches for the optimal solution consistent with the assumptions. This means the sub-problem never returns SAT, but instead always returns either UNKNOWN (if *lim* is not infinite) or a (potentially new) model, together with an unsatisfiable core. Whenever a new model μ^* is found, it is immediately invalidated by adding a new constraint to S: $[\![obj < \mu^*(obj)]\!]$.

When the subproblem returns an empty unsatisfiable subset of μ, we terminate immediately with the best solution found. This result means that no as-

signment in μ was responsible for the sub-problem being unable to find a better solution than μ^*, so we must have found the optimum.

When a non-empty clause can be learned, we try to learn several independent clauses from this MIP assignment. We identify an inconsistent part of the assignment, C, and minimize it using a process broadly similar to figure 7.1, remembering that the subproblem solver used in cut minimization can find new solutions, which will be returned as μ^*. This minimized cut is then added to the master problem. We iterate (as described in more detail in section 7.3.2) to generate independent cuts, excluding parts of the assignment already found to occur in previous cuts.

Because the sub-problem never returns SAT, the termination condition is different to traditional LBBD: search terminates when the master fails (typically after being returned an empty cut).

This allows our variant of LBBD to exploit CP optimisation: If a new incumbent is found, we may now tighten the objective bound in both the master and sub-problem in the hope that the sub-problem can utilise this bounds information to generate more succinct cuts. In particular, the sub-problem can generate an empty cut as soon as it finds any model with an objective equal to the lower bound.

In classical LBBD, the master can be seen as incrementally building a MIP model representing a projection of the problem onto the master variables. By adding these objective-based cuts, and allowing the sub-problem to use them to simplify its cuts, we can instead view the master as building a projection of the subset of solutions strictly better than the incumbent. The inner **while** loop represents an approximation of looping over sub-problems, described more fully in Section 7.3.3.

7.3.1 Deriving and encoding cuts

In the above formulation, we have not explicitly decomposed the CP model into disjoint sub-problems; indeed, the sub-problem is exactly the model that would be used to solve the problem with a direct CP approach.

Using a classical CP solver, this is not an ideal approach; the solver cannot readily pinpoint the unsatisfiable sub-problem. However, conflict directed clause learning allows the solver to identify potentially infeasible subsystems, and activity-driven search heuristics direct the solver to explore those subsystems.

The LCG sub-problem detects infeasibility when it is forced to backtrack past the most recent assumption. From the final conflict, we can traverse the implication graph (in the same manner as conflict analysis) to derive a cut C of the form $[\![x_1 \geq k_1]\!] \wedge \cdots \wedge [\![x_n < k_n]\!] \Rightarrow \texttt{false}$ consisting only of (possibly relaxed) assumptions. This can be added to the master as: $(1 - [\![x_i \geq k_i]\!]) + \cdots + (1 - [\![x_n < k_n]\!]) \geq 1$. If these cuts happen to be over Boolean variables, then no further action is needed, but for more general cuts such (e.g. $[\![x > 30]\!] \wedge [\![y \leq 10]\!] \Rightarrow \texttt{false}$), we must first introduce fresh Boolean variables encoding these atoms.

In the case that all bounds literals from the lower to upper bound have been instantiated, these new literals can be encoded in the MIP by the following constraints:

$$[\![x \geq k+1]\!] \geq [\![x \geq k]\!] \qquad \forall k \geq lb \tag{7.1}$$

$$x = lb(x) + \sum_{i=lb(x)+1}^{ub(x)} [\![x \geq i]\!] \tag{7.2}$$

However it is desirable to be able to lazily generate these bounds literals only when they appear in some cut. Consequently we define upper and lower bounds for equation 7.2 which are correct for any subset B of bounds literals belonging

to variable x. Assume $B = \{[\![x \geq k_1]\!], \cdots, [\![x \geq k_n]\!]\}$, and $k_i < k_{i+1}$, and let $k_0 = lb(x)$ and $k_{n+1} = ub(x)$.

$$x \geq k_0 + \sum_{i=1}^{n} (k_i - k_{i-1})[\![x \geq k_i]\!] \tag{7.3}$$

$$x \leq k_1 - 1 + \sum_{i=1}^{n} (k_{i+1} - k_i)[\![x \geq k_i]\!] \tag{7.4}$$

When B is the full set of bounds literals, we can see that the RHS of both equation 7.3 and equation 7.4 converge to the RHS of equation 7.2. The encoding is revised as fresh bounds are introduced as adding additional bounds literals can only tighten the inequalities. Observe that this encoding permits x to take values *between* introduced bounds.

7.3.2 Cut minimisation

The nogoods $(C \Rightarrow \texttt{false})$ derived by the LCG solver will typically involve only a small subset of problem variables – in the case of independent sub-problems, the nogoods will refer to only one sub-problem. These form valid cuts, but are not necessarily minimal. These cuts can be reduced by applying any of the MUS construction approaches outlined in Section 7.2.4, we use MUS-seq (Figure 7.1). This achieves much the same effect as the cut strengthening outlined by Hooker (2007).

Here we see some side-effects of sub-problem completeness. During cut minimisation, the sub-problem solver may find a model μ^*. As we have only a single complete sub-problem containing all the constraints, this model is a feasible solution to the overall problem.

We can then add a constraint to the sub-problem constraining the objective to be strictly better than this new solution, potentially allowing the cut to be further simplified. An interesting side effect is that during cut minimisation the sub-problem solver will find new incumbent solutions, tighten the objective bound

and continue searching until it either proves no solutions better than the new incumbent exist in this neighbourhood or it exceeds its resource budget. Consequently, if the sub-problem solver were executed with an unbounded resource limit, it would always return an empty cut (essentially just running the LCG solver to completion). In the algorithm of Figure 7.1, this means we never positively identify a bound as being in the MUS; instead, constraints may always be eliminated by a later conflict.

7.3.3 Deriving multiple independent cuts

Typically in LBBD, each sub-problem can be used to learn a cut per iteration. We approximate this using the observation that a minimal cut often contains variables exclusively from one sub-problem. Thus, after obtaining a cut $C \Rightarrow false$, we attempt to generate additional independent cuts by removing all assumptions which occur in C, and asking the CP solver for a new cut over the remaining assumptions. It is possible that this process will learn multiple independent cuts from the same subsystem before moving on to the next, however it is not obvious that this is a bad thing.

Similar to cut minimisation, it is necessary to limit the resource budget of the sub-problem solver (*solve-lcg-lim*), as the search space implied by the reduced set of assumptions can become arbitrarily large, and the **while** loop only stops generating cuts when the sub-problem solver returns `UNKNOWN` or an empty cut.

7.3.4 Cut generation as large neighbourhood search

As we noted in Section 7.1, removing assumptions corresponds to growing the neighbourhood explored by the sub-problem. In both cut-minimisation, and multiple cut generation, we remove assumptions that caused failure from the initial

neighbourhood generated by the MIP. This leads to an exploration strategy similar to explanation-based LNS (Prud'homme, Lorca, and Jussien, 2014).

The number of assumptions provided to the sub-problem solver will vary hugely during the solving process, so we rely on the resource limits to prevent long-tailed solve times. So long as one cut is generated then the MIP is guaranteed to generate a different solution, and the search space will be fully explored eventually. Since the initial solve given the full MIP assignment is run without any limit, this is guaranteed to terminate eventually.

7.4 Experiments

We have implemented the described approach, modifying the assumption interface of CHUFFED, a lazy clause generation solver, to report feasibility cuts.

We evaluated the approach on several sets of scheduling problems, described below. For each class, we tested Chuffed (CP) (Chu, 2011), Gurobi 6.5 (MIP) (Gurobi Optimization, 2013) and our logic based Benders decomposition approach (LBBD). For LBBD, cut minimisation was run with a budget of 512 conflicts, after which the current cut was returned.

The problem classes we consider are described in the next four paragraphs. In each case identical MiniZinc models were provided to each solver.

Planning and Scheduling This problem requires scheduling independent tasks on a set of machines with capacity limits, and machine-dependent task durations. These instances have been used to evaluate the effectiveness of logic-based Benders decomposition in a number of prior works (Hooker, 2007; Heinz, Ku, and Beck, 2013; Ciré, Coban, and Hooker, 2013), minimising cost, makespan or tardiness. We report on models minimising cost (PS-cost) and makespan (PS-makespan) in our results tables.

Alternative Resource Scheduling with Sequence Dependent Setups This problem, like the planning and scheduling problem, requires scheduling independent tasks on a set of machines. However machines cannot run tasks in parallel, and tasks require setup time which varies with machine and preceding task. The LBBD approach of Tran and Beck (2012) separates the machine allocation from scheduling, resulting in a per-machine TSP which is solved with a dedicated TSP solver. There are 3 sub-classes of this benchmark: production-dominated (ARS-p-dom), setup-dominated (ARS-s-dom), and balanced (ARS-balanced).

Single-source capacitated plant location problem The SSCPLP (Barcelo, Fernandez, and Jörnsten, 1991) is a discrete plant location problem, where each customer is assigned to a single facility, such that the combined cost of open facilities and customer service is minimised. This is not an ideal candidate for LBBD, having a natural MIP encoding, but an LBBD approach in which the master decides which plants to open and the sub-problem assigns customers to plants is similar to the classical Benders approach which has been used for variants of this problem (Geoffrion and Graves, 1974). The SSCPLP is also interesting as it is the basis for the following problem.

Capacity- and distance-constrained plant location problem The CDCPLP (Albareda-Sambola, Fernández, and Laporte, 2009) extends the SSCPLP, adding a fleet of distance-limited vehicles required to service customers. An LBBD approach was presented in (Fazel-Zarandi and Beck, 2011) which allocates customers to facilities in the master, leaving a bin-packing sub-problem per facility to be solved with a CP solver (after first trying a greedy heuristic).

Experiments Table 7.1 compares our LBBD approach (using *Gurobi 6.5* and *Chuffed*) to MIP (*Gurobi 6.5*) and CP (*Chuffed*) on 3 metrics: average solution quality; average time to prove optimality; and the number of instances proved optimal, for

		Quality %			Time			Num. Optimal			
	Count	LBBD	MIP	CP	LBBD	MIP	CP	LBBD		MIP	CP
PS-makespan	335	**100.0**	57.4	89.0	**61.7**	369.4	101.3	**311**	+25	150	286
PS-cost	335	**100.0**	80.7	88.8	**95.7**	400.1	122.3	**301**	+29	131	281
SSCPLP	57	89.9	**100.0**	72.6	271.9	**95.8**	589.6	34	+0	**50**	1
CDCPLP	300	73.2	**99.8**	57.0	**424.4**	426.7	597.3	99	+23	**140**	3
ARS-balanced	270	**100.0**	**100.0**	83.6	144.3	**47.0**	180.1	224	+0	**269**	211
ARS-p-dom	270	**100.0**	**100.0**	83.6	106.8	**73.9**	179.9	255	+2	**261**	206
ARS-s-dom	270	**100.0**	**100.0**	72.9	207.0	**100.7**	265.8	189	+0	**253**	171
Total	1837	**95.3**	88.7	79.3	**173.8**	245.6	248.6	**1413**	+79	1254	1159

Table 7.1: Quality score, mean runtime, and number of instances proved optimal in 600s.
+N indicates LBBD solved N instances unsolved by either MIP or CP.

each of the benchmark sets described above.

We can see in Figure 7.4 that *Chuffed* performs very well on some small instances, but for a time-budget of over 13 seconds, LBBD is the fastest to prove optimality of the 3 techniques tested. This is supported by Table 7.1, where we see that LBBD is both fastest on average to prove optimality, and proves the largest number of instances optimal.

In addition to LBBD's expected strength at proving optimality, Table 7.1 shows that our implementation (including the LNS primal heuristic which naturally occurs from our single-sub-problem formulation) makes for an excellent primal solver, finding higher quality solutions on average than both other techniques.

Solution quality obtained by a solver s for each instance i is defined to be $c_{tbp}(i)/c_s(i)$ where $c_s(i)$ is the objective of the best solution to instance i found by s in the time limit, and *tbp* is the theoretical best portfolio of all 3 solvers. We report this as a percentage, which can be seen as a ratio of the performance of the theoretical best portfolio of the 3 solvers. We see that our approach achieves 95% of the performance of the theoretical best portfolio, giving us the best of both of these contrasting optimisation technologies.

However our approach is more than just a portfolio, in Figure 7.5 we can see

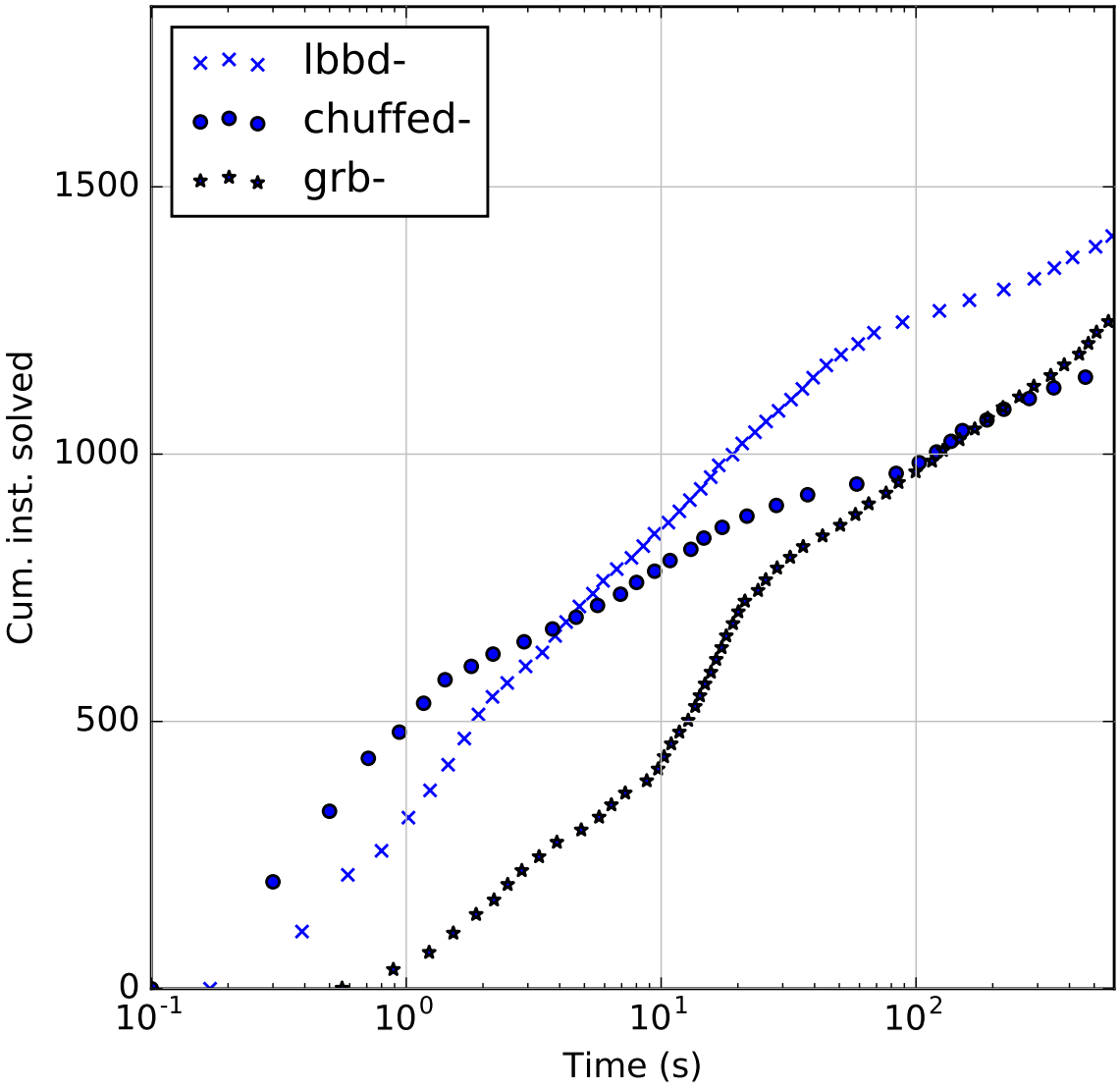


Figure 7.4: Solutions proved optimal vs time

that many optimality proofs are faster using our LBBD approach than MIP or CP. In particular note the 79 instances which were only proved optimal by LBBD.

Table 7.2 examines the hard "c" instances from the planning and scheduling benchmarks in detail. These instances are some of the most studied in the LBBD literature (Hooker, 2007; Heinz, Ku, and Beck, 2013; Ciré, Coban, and Hooker, 2013; Ciré, Coban, and Hooker, 2015), and this table is designed to be reasonably comparable to tables in previous work (Ciré, Coban, and Hooker, 2013; Ciré, Coban, and Hooker, 2015). Our experiments use a much shorter time limit than those papers, however we can still make broad comparisons: most notably *Chuffed* performs much better than the traditional CP approaches considered in previous work. To our knowledge, this is the first time a LBBD approach has been directly compared with an LCG CP solver. This strong performance may suggest that the clausal learning of LCG, and cut-generation in LBBD have similar strengths. It is also interesting to note that the custom LBBD solver of Ciré, Coban, and Hooker (2015) was only able to solve 6 more instances in 2 hours than our LBBD approach solved in 10 minutes.

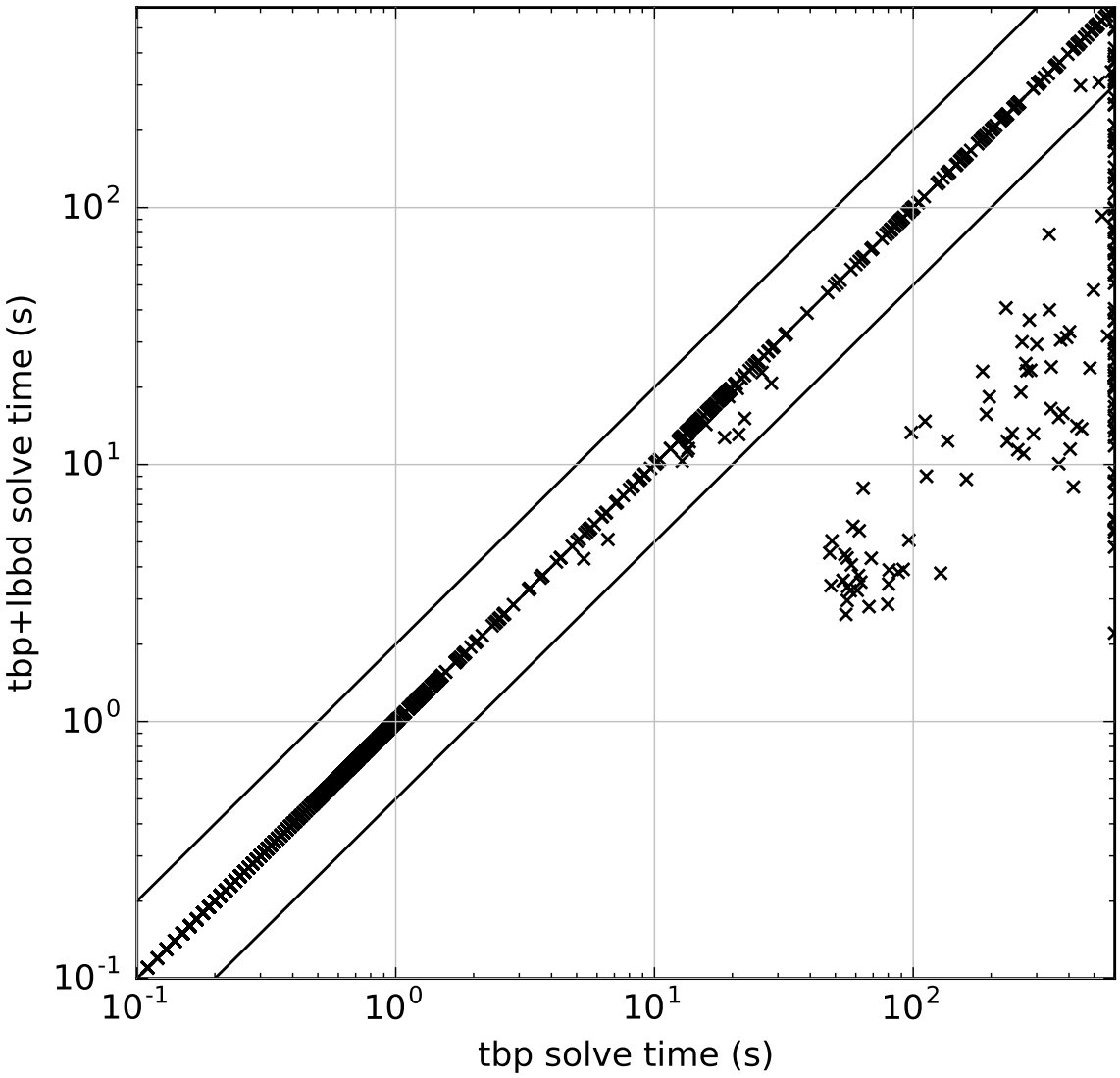


Figure 7.5: Performance of the the theoretical best portfolio with and without LBBD. 79 instances were solved by LBBD but neither MIP nor CP.

	Num. Optimal			Mean Runtime		
	LBBD	MIP	CP	LBBD	MIP	CP
2m-10j	**5**	**5**	**5**	0.218	17.288	**0.148**
2m-12j	**5**	**5**	**5**	0.304	51.444	**0.162**
2m-14j	**5**	**5**	**5**	0.412	85.742	**0.288**
2m-16j	**5**	**5**	**5**	1.13	211.964	**0.508**
2m-18j	**5**	2	**5**	1.876	406.58+	**0.782**
2m-20j	**5**	2	**5**	1.95	402.704+	**0.814**
2m-22j	**5**	0	**5**	143.898	—	**103.226**
2m-24j	**4**	0	**4**	197.684+	—	**194.526**+
2m-26j	2	0	**3**	**385.768**+	—	447.648+
2m-28j	**4**	0	1	**134.132**+	—	546.024+
2m-30j	**3**	0	2	**418.734**+	—	451.346+
2m-32j	**3**	0	0	**255.462**+	—	—
3m-10j	**5**	**5**	**5**	0.26	17.626	**0.154**
3m-12j	**5**	**5**	**5**	0.414	49.222	**0.24**
3m-14j	**5**	**5**	**5**	0.614	111.21	**0.298**
3m-16j	**5**	1	**5**	1.156	488.596+	**0.474**
3m-18j	**5**	0	**5**	2.82	—	**2.328**
3m-20j	**5**	0	**5**	**2.686**	—	4.392
3m-22j	**5**	0	4	**4.476**	—	125.788+
3m-24j	**5**	0	**5**	**17.948**	—	70.838
3m-26j	**5**	0	3	**44.138**	—	251.972+
3m-28j	**5**	0	2	**67.198**	—	404.374+
3m-30j	**2**	0	**2**	**405.072**+	—	476.176+
3m-32j	**3**	0	1	**421.492**+	—	597.066+
4m-10j	**5**	**5**	**5**	0.294	15.592	**0.19**
4m-12j	**5**	**5**	**5**	0.392	35.982	**0.246**
4m-14j	**5**	4	**5**	0.608	188.058+	**0.294**
4m-16j	**5**	**5**	**5**	0.94	213.0	**0.618**
4m-18j	**5**	1	**5**	1.452	560.298	**0.62**
4m-20j	**5**	0	**5**	2.534	—	**1.272**
4m-22j	**5**	0	**5**	**5.622**	—	13.858
4m-24j	**5**	0	4	**34.734**	—	127.848+
4m-26j	**5**	0	4	**119.762**	—	158.276+
4m-28j	**5**	0	**5**	168.614	—	**29.424**
4m-30j	**3**	0	**3**	411.584+	—	**380.55**+
4m-32j	3	0	**5**	361.884+	—	**326.076**

Table 7.2: Planning and Scheduling "c" instances: "Xm-Yj" schedules Y jobs over X machines.

7.5 Conclusion and Further Work

We have introduced the first "model-and-run" LBBD solver, which additionally uses a natural LNS-like primal heuristic. This solver may not be able to outperform all custom LBBD implementations, e.g. ARS with a dedicated TSP solver, but can tackle any problem with no additional implementation cost. The resulting hybrid combines the strengths of MIP and CP and can be superior to both of them on appropriate problems.

One important feature of many LBBD solvers that we have not addressed is relaxations of the sub-problem encoded in the master, especially using continuous variables. We expect this can be achieved by defining relaxations for global constraints using fresh variables (which cannot then cause conflict in the sub-problem). We expect these to be important for evaluating this approach against a broader array of benchmarks.

7.6 Extension: Operator Scheduling

Temporal planning solves a similar class of problems to alternative scheduling, but is typically modelled differently. In this section we will show how temporal planning problems modelled in a temporal SAS+ formalism can be transformed into a sequence of alternative scheduling problems.

Temporal SAS+ is an extension of classical SAS+ where durative actions are treated as a pair of start and end events, each having preconditions and postconditions just like regular SAS+ (Eyerich, Mattmüller, and Röger, 2009). In addition to these instantaneous preconditions, durative actions can have invariants: preconditions which must hold for the duration of the action.

Other planners use a Linear Program to solve a simple scheduling problem as part of a forward search, most notably the related popf and optic planners (Coles,

Coles, et al., 2010; Benton, Coles, and Coles, 2012).

Of more immediate relevance are planners which compile to constraint programming formulations, of particular interest is CPT, which tackles temporal problems (Vidal and Geffner, 2006) and the recently introduced PaP and TCPP planners which treat each SAS+ variable as an independent timeline which are explicitly synchronised (Barták, 2011; Ghanbari Ghooshchi et al., 2017).

7.6.1 A linear-size CP encoding of partial-order planning

We present an encoding that combines aspects of these approaches in a way that enables the addition of resource constraints that can be tackled with existing scheduling constraints. We observe that each variable must take exactly one value at any time. This might be represented in a typical scheduling problem as a disjunctive constraint.

Given this observation we can treat the planning problem given a fixed set of actions as a scheduling problem, and we can use an estimate of the number of required copies of an operator generated, for example, by an operator-counting heuristic. We use $\mathcal{C}(o)$ to denote the number of copies in the current restricted problem. We also slightly abuse notation to use $\mathcal{C}(X)$ to denote the total number of actions which change the value of variable X.

We then encode the scheduling model as a set of events: the start and end of the application of each action. We use $o_{i,\vdash}$ and $o_{i,\dashv}$ to denote the events corresponding to the start and end of the i-th application of operator o.

We use $prec(o_{\vdash})$, $prec(o_{\dashv})$ for the preconditions of the start and end of operators, and $post(o_{\vdash})$, $post(o_{\dashv})$ for their postconditions. We assume that any "overall" precondition $X = x$ of an operator o is also a precondition of the end event, and satisfies the predicate $invar(o, X)$. We will sometimes use events in place of operators in these functions as a shorthand, so, for example, $prec(o_{i,\dashv})$ should be

considered equivalent to $prec(o_{\dashv})$.

These events define the starts and ends of various time-windows in which each variable takes a specific value.

We use the following variables in our model:

- $Seq(X, i)$, the i-th event effecting variable X.
- $Val(X, i)$, the value of X after the i-th event effecting it.
- $S(X, i)$, the time the i-th value of X is added.
- $E(X, i)$, the time the i-th value of X is deleted.
- $InPlan(e)$, true iff e is included in the plan.
- $SI(X, e)$, the i such that $Seq(X, i)$ is the supporter of precondition $X = Val(X, i)$ for event e.
- $EI(X, e)$, the i such that $Seq(X, i)$ is the effect of event e on X.
- $T(e)$, the time at which e occurs.
- $\Delta(o, i) = T(o_{i,\dashv}) - T(o_{i,\vdash})$, the duration of the i-th application of operator o.

All of these variables which are a function of an event are optional integers (Mears et al., 2014), conditional on $InPlan(e)$.

These variables then participate in the following constraints:

$$\forall X \in V: \quad increasing([S(X,i)|i \in \{1..\mathcal{C}X\}]) \tag{7.5}$$

$$\forall X \in V: \quad increasing([T(o_{\vdash,i})|i \in 1..\mathcal{C}(o)]) \tag{7.6}$$

$$\forall e \in E, X = x \in prec(e): \quad InPlan(e) \Rightarrow S(X, SI(X,e)) < T(e) \leq E(X, SI(X,e)) \tag{7.7}$$

$$\forall e \in E, X = x \in post(e): \quad InPlan(e) \Rightarrow S(X, EI(X,e)) = T(e) \tag{7.8}$$

$$\forall X \in V, i \in \{1..\mathcal{C}(X)\}: \quad E(X,i) \leq S(X,i+1) \tag{7.9}$$

$$\forall o_{i_\dashv} \in E, X = x \in prec(e): \quad invar(o,X) \Rightarrow T_S(X, o_{i,\dashv}) \leq T(o_{i,\vdash}) \tag{7.10}$$

$$\forall e \in E, X = x \in prec(e): \quad InPlan(e) \Rightarrow Val(X, SI(X,e)) = x \tag{7.11}$$

$$\forall e \in E, X = x \in post(e): \quad InPlan(e) \Rightarrow Val(X, EI(X,e)) = x \tag{7.12}$$

$$\forall o \in O, i \in \{1..\mathcal{C}(o)\}: \quad InPlan(o_{\dashv,i}) \equiv InPlan(o_{\vdash,i}) \tag{7.13}$$

$$\forall X \in V, i \in \{1..\mathcal{C}(X)\}: \quad \neg InPlan(Seq(X,i)) \Rightarrow Val(X,i) = Val(X,i-1) \tag{7.14}$$

$$\forall X \in V, i \in \{1..\mathcal{C}(X)-1\}: \quad \neg InPlan(Seq(X,i)) \Rightarrow \neg InPlan(Seq(X,i+1)) \tag{7.15}$$

$$\forall X = x \in s_0: \quad Val(X,0) = x \tag{7.16}$$

$$\forall X = x \in s_*: \quad Val(X, \mathcal{C}(X)) = x \tag{7.17}$$

The objective is then to minimise $max(\{T(Seq(X,\mathcal{C}(X)))|X \in V\})$.

Equation 7.5 ensures that the start times for each window in the timeline for variable X are in order. Equation 7.6 breaks symmetries between the order in which different copies of an action can occur. Equation 7.7 ensures that an event must occur strictly after its preconditions are satisfied, and before they next become unsatisfied. Equations 7.8, and 7.9 ensure that the time-windows in which variables take a specific value are equal to the event times of the causes of these changes. Equation 7.10 ensures that any "over all" preconditions of actions are

satisfied no later than the start of the action. Equations 7.11 and 7.12 ensure that preconditions and postconditions of an event hold on all relevant variables before and after each event. Equation 7.13 ensures that any operator that starts must also finish, and vice-versa. Equations 7.14 and 7.15 ensure respectively that actions not occuring in the plan have no effect on the value of a variable, and that such non-occuring actions occur at the end of the sequence. Equations 7.16 and 7.17 ensure the initial and final values of each timeline are consistent with the initial and goal states respectively.

Note that the number of variables and constraints in this encoding both grow linearly in E. This is in contrast with CPT's similar encoding which requires a constraint for every pair of potentially conflicting actions (Vidal and Geffner, 2006). This is not necessarily asymptotically faster, as the domain of the variables in our encoding also increases linearly with E, but requires less memory and is typically faster in practice, all else being equal.

The remaining challenge is to choose appropriate action counts. In this choice we have two options based on ideas developed in this thesis: Use an operator counting heuristic and learn generalised landmarks as discussed in chapter 5; or, relax the scheduling sub-problem in such a way that the relaxation may be refined by restricting any relaxed operators chosen in the solution, and adding more events.

The former approach is theoretically straightforward assuming the reader is familiar with chapter 5. The alternative involves relaxing the precondition constraints for events supported by the last event in a variable sequence. This can be considered similar to the work of Robinson, Gretton, Pham, and Sattar (2010) in optimal SAT planning which adds a delete-relaxed suffix to a layer-based SAT formula.

The latter approach requires that we first transform our CP model from a restriction of the full planning problem into a relaxation. To this end we add a

delete-relaxation to our model.

7.6.2 Partial-order relaxation

We can't simply apply a delete-relaxed "suffix" to our plan as there isn't necessarily a single state where we can sensibly start using delete-relaxed operators. Instead we allow delete-relaxed actions to be interleaved with regular actions. To achieve this we add one delete relaxed copy of each operator to our action counts, we refer to the set of these relaxed events as E^+, and $o^+_{\vdash}$ and $o^+_{\dashv}$ to represent the start and end events derived from relaxing operator o.

In order to accommodate relaxed actions in our partial-order formulation, we modify constraint 7.9 to read:

$$\forall X \in V, i \in \{1..\mathcal{C}(X)\} : \quad (Seq(X,i) \notin E^+) \Rightarrow E(X,i) \leq S(X,i+1) \tag{7.18}$$

$$\tag{7.19}$$

This change allows the effect of an event to persist indefinitely, in-spite of X taking other values at later times. We must also modify the goal achievement constraint, equation 7.17 to read

$$\forall X = x \in s_* : \exists i. Val(X,i) = x \wedge (i = \mathcal{C}(X) \vee Seq(X,i) \in E^+) \tag{7.20}$$

$$\tag{7.21}$$

This is slightly different to a traditional delete-relaxation where previous values remain true after the relaxed action, but this variant makes it simpler to prove that this is in fact a relaxation of the full planning problem. Nonetheless the resulting relaxation is sufficiently similar we do not feel it warrants a distinct name.

However this is not sufficient: the optimal solution to this program would simply compute the optimal delete relaxed plan, regardless of the other opera-

tors available. This precludes the use of this model in a relaxation-refinement approach as it will not converge to a true solution to the problem.

To address this we must add two new constraints:

$$\forall o \in O: \quad T(o_{\vdash,\mathcal{C}(o)}) < T(o^+_\vdash) \tag{7.22}$$

$$\forall o \in O: \quad \neg InPlan(o_{\vdash,\mathcal{C}(o)}) \Rightarrow \neg InPlan(o^+_\vdash) \tag{7.23}$$

This requires the solution to use real operators before any delete-relaxed operators of the same type.

This model is a relaxation of the original planning problem regardless of the action count: any plan for the original problem can be transformed into a plan for this model by replacing the $\mathcal{C}(o)+1$-th application of an operator o in the original plan with o^+ in the relaxed plan, and omitting all subsequent applications of o. All operators omitted this way are redundant, as all of the positive effects will already be available earlier as a result of the relaxed action.

This partially-relaxed model allows us to apply a relaxation-refinement approach: First the optimal solution to the relaxed model is obtained. If this contains any delete-relaxed actions, a new model is constructed with one more real copy of each relaxed operator used. Otherwise the solution to the relaxed model is also a solution to the original problem, and thus must be optimal.

This approach has no fixed-points other than solutions: if an operator's count is incremented, the previous solution is no-longer valid in this or any future model because at least one more real copy of the operator must be used strictly before a delete-relaxed version may be applied.

7.6.3 Conclusions and Further Work

We have introduced a relaxation-refinement approach to temporal planning that solves the planning problem by solving a series of optional scheduling problems.

The results of these solves inform the way that the relaxation is refined. Our model requires only a linear number of variables in the number of events being scheduled.

Unlike CPT, our approach does not rely on a specialised solver to carefully propagate some constraints only in certain directions (Vidal and Geffner, 2006). Having such a compilation of temporal planning to general CP allows us to consider new ways to integrate existing scheduling constraints used in resource-constrained scheduling problems.

One such approach is to add envelope-actions (Coles, Fox, Halsey, et al., 2009) which could be annotated to indicate resource consumption, allowing us to add constraints like:

$$\forall R : cumulative([\langle T(o_{i,\dashv}), \Delta(o,i), k\rangle | \langle o,k\rangle \in requires(R), i \in \mathcal{C}(o)], capacity(R))$$

where $requires(R)$ is the set of $\langle o,k\rangle$ pairs where operator o that consumes k units of a reusable resource with a total concurrent usage limit of $capacity(R)$. In our running gripper example, we could use an envelope action to represent "holding(ball)", which could allow us to represent the capacity to hold several balls simultaneously. This constraint could safely be applied to the delete-relaxed actions in addition to real actions. However care will need to be taken in considering the interaction between such constraints and the delete relaxation in the general case.

We had hoped that the linear-size encoding would counteract the overhead of the MiniZinc modelling language, as compared to CPT's highly specialised and optimised CP system. However, since CPT only supports basic concurrency with fixed action duration and no required concurrency, our preliminary experiments suggest that the additional overhead of supporting required concurrency largely eliminates this benefit. Additionally existing temporal planning bench-

marks are not ideally encoded for this approach, and we would need to investigate specialised domains which truly exploit the potential for adding scheduling constraints in order to demonstrate the true value of this approach.

Chapter 8
Conclusion

Here we reiterate the contributions, conclusions and further work identified in this thesis.

In this thesis we set out to explore the ways in which knowledge derived from conflict can be exploited to find and prove optimal plans in three key variants of the planning problem.

We introduced a number of algorithms, listed in table 8.1, in the development of these algorithms, several new notions of conflict and representations of knowledge were also introduced. In many cases these conflicts and representations of knowledge have potential applications in different algorithms, which we have not yet explored.

We also examined a number of existing algorithms through the lens of conflict-directed learning.

Algorithm	Conflict	Knowledge Learned
FragPlan (Ch. 3+4)	Resource limitations	Improving agent plan
OpSeq (Ch. 5)	Unsequencable	Generalised Landmark
CDHL (Ch. 6)	Suboptimality	Clause
MznLBBD (Ch. 7)	Unschedulable	Benders Cut
OpSched (Sec. 7.6)	Unschedulable	Add Events

Table 8.1: Summary of introduced algorithms

8.1 Part I

In Part I we examined multi-agent planning, introducing Fragment-Based Planning (FragPlan in table 8.1), certainly the most empirically effective of the algorithms introduced in this thesis. FragPlan is an anytime multi-agent plan-space planning algorithm for real-world resource constrained problems which can solve problems orders of magnitude larger than state-of-the-art planning or scheduling approaches.

Fragment-Based planning relies on shared resources to model agents' interactions, agents then plan independently and over-used resources are conflicts, which are subsequently assigned prices. Agents will then tend to avoid using these resources in subsequent plans. FragPlan can thus either be seen as learning the true cost of resources, or as learning an optimal set of per-agent plans. We prefer the latter view because there are domains in which there is not **one** set of prices from which we can derive the optimal joint plan, but a different set of prices for each agent.

In defining this algorithm we introduced the notion of "bounded-benefit" programs, a decidable subclass of *Golog* program. These have to potential to simplify modelling in traditional *Golog* dialects by exploiting a notion of cost to prune areas of the search.

We also introduced "precondition relaxations", an application of Lagrangian Relaxation to dynamic multi-agent systems. These have the potential to enable a principled way to combine per-agent heuristics into an admissible multi-agent heuristic, while preserving some degree of privacy.

8.2 Part II

In Part II we consider classical planning, and introduce two significant algorithms Operator Sequencing (OpSeq), and Conflict-Directed Heuristic Learning (CDHL). OpSeq is an application of Logic-Based Benders Decomposition to the classical planning problem. It exploits the observation that the cost of a plan is independent of the order in which the actions are applied, in order to build an ever-improving lower-bound on plan cost. At each iteration a Mixed-Integer Program is solved to generate an estimate of an optimal set of actions (called a plan projection or operator count). This projection is then either sequenced into a plan, or refuted and a "generalised landmark" is learned and added to the MIP to refine future operator counts.

We introduced a SAT-based sequencing/refutation sub-problem, but one can easily imagine a state-based algorithm that attempts to explore the reachable state-space restricted to using only the projected actions. Such a sub-problem solver could allow some of the learned landmarks to be re-used recursively in similar states. We believe this kind of sub-problem solver may be more effective (if more complex) than the SAT-based approach we present in this thesis.

More traditional applications of generalised landmarks remain unexplored. For example, one can imagine a fixed heuristically-generated set of these landmarks could be computed at the start of a search and exploited in a traditional heuristic similar to LM-cut, operator counting, or potential heuristic.

In this part we also introduce Conflict-Directed Heuristic Learning (CDHL), which generalises IDA* with a transposition table. We first analyse the addition of a transposition table to IDA* as a non-generalising, but nonetheless surprisingly effective learning algorithm, using the notion of reduced-cost of operators as the root of the type of conflict that IDA* detects and prunes.

Given this understanding, we exploit regression to examine which proper-

ties of a state are actually responsible for it being further from the goal than the heuristic initially estimated, and update the heuristic estimate for a set of states, not just the one we have actually seen. Specifically we learn a clause in terms of facts that **do not hold** in a state, and integrate into this clausal database a successor generator that is aware of reduced cost, allowing CDHL to learn to prune sub-trees earlier.

8.3 Part III

Part III explores temporal planning and scheduling problems. Learning from conflict in traditional scheduling problems is well explored in the constraint programming literature, and we introduce an Automatic Logic-Based Benders Decomposition (AutoLBBD), which out-performs a state-of-the-art conflict-directed learning CP system, at least on the expressive alternative scheduling problems we investigate.

AutoLBBD is the first "model-and-run" LBBD solver, capable of solving an arbitrary unannotated constraint program using this highly-effective conflict-directed algorithm. Our results show that this approach, while far from perfect, outperforms both traditional CP and MIP formulations of several alternative scheduling problems. This chapter is, to our knowledge, the broadest comparison of logic-based Benders with MIP and CP. Our automatic decomposition enabled us to investigate many different domains without significant additional engineering effort.

This work also highlighted the duality of Logic-Based Benders Decomposition and the local search technique Large Neighbourhood Search.

To exploit this AutoLBBD framework, we also described Operator Scheduling (OpSched), a CP model of expressive temporal planning, unfortunately, disappointing preliminary results suggest that the overhead of the MiniZinc modelling

language is significant. Additionally, we suspect that the types of problems considered in existing temporal planning benchmarks do not exploit the potential strengths of this approach.

We are particularly interested in investigating domains which could potentially exploit the rich resource-constrained scheduling constraints that can be added to a temporal problem in this framework. In developing this extension we also introduced a partial-order delete relaxation which may have applications in more traditional partial-order planners.

8.4 Summary

Our results show that conflict-directed reasoning is a highly effective approach to cost-optimal planning in industrial domains when appropriate decompositions and explanations are known.

We have introduced a number of novel notions of conflict, particularly within plan-space search, and algorithms which learn and exploit knowledge that can be learned from these conflicts. Several of these algorithms derive decompositions and explanations automatically from unannotated problem descriptions in the standard modelling languages PDDL and MiniZinc. We believe there is significant future work to be done in investigating more efficient ways to derive and exploit these decompositions and explanations.

We are excited about the directions for future research our investigations into conflict in plan-space planning has opened. It is our hope that new, more expressive modelling approaches for planning can be built upon the algorithms introduced in this thesis.

Bibliography

Albareda-Sambola, Maria, Elena Fernández, and Gilbert Laporte (2009). "The Capacity and Distance Constrained Plant Location Problem". In: *Computers & Operations Research* 36.2, pp. 597–611.

Appelt, Douglas E. and Martha E. Pollack (1992). "Weighted Abduction for Plan Ascription". In: *User Modeling and User-Adapted Interaction* 2.1-2, pp. 1–25.

Bäckström, Christer (1992). "Equivalence and Tractability Results for SAS+ Planning." In: *International Conference on Knowledge Representaion and Reasoning*, pp. 126–137.

Baier, Jorge A. et al. (2008). "Beyond Classical Planning: Procedural Control Knowledge and Preferences in State-of-the-art Planners". In: *National Conference on Artificial Intelligence (AAAI 08)*. Chicago, Illinois: AAAI Press, pp. 1509–1512.

Bakker, R. R. et al. (1993). "Diagnosing and Solving Over-Determined Constraint Satisfaction Problems". In: *International Joint Conference on Artificial Intelligence (IJCAI 93)*. Morgan Kaufmann, pp. 276–281.

Balas, Egon (1989). "The Prize Collecting Traveling Salesman Problem". In: *Networks* 19.6, pp. 621–636.

Barcelo, Jaime, Elena Fernandez, and Kurt O. Jörnsten (1991). "Computational Results from a New Lagrangean Relaxation Algorithm for the Capacitated Plant Location Problem". In: *European Journal of Operational Research* 53.1, pp. 38–45.

Barnhart, Cynthia et al. (1998). "Branch-and-Price: Column Generation for Solving Huge Integer Programs". In: *Operations Research* 46.3, pp. 316–329.

Barták, Roman (2011). "A novel constraint model for parallel planning". In: *Florida Artificial Intelligence Research Society Conference (FLAIRS 11)*. AAAI Press, pp. 9–14.

Benton, J., Amanda Coles, and Andrew Coles (2012). "Temporal Planning with Preferences and Time-Dependent Continuous Costs". In: *International Conference on Automated Planning and Scheduling (ICAPS 12)*. AAAI Press.

Biere, Armin, Marijn Heule, and Hans van Maaren (2009). *Handbook of Satisfiability*. Vol. 185. IOS press.

Blom, Michelle L. and Adrian R. Pearce (2010). "Relaxing Regression for a Heuristic GOLOG". In: *Starting AI Researchers' Symposium (STAIRS 2010)*. Amsterdam, The Netherlands: IOS Press, pp. 37–49.

Bonet, Blai (2013). "An Admissible Heuristic for SAS+ Planning Obtained from the State Equation". In: *International Joint Conference on Artificial Intelligence (IJCAI 13)*. IJCAI/AAAI, pp. 2268–2274.

Bonet, Blai and Menkes van den Briel (2014). "Flow-Based Heuristics for Optimal Planning: Landmarks and Merges". In: *International Conference on Automated Planning and Scheduling (ICAPS 14)*. AAAI Press, pp. 47–55.

Bonet, Blai and Malte Helmert (2010). "Strengthening Landmark Heuristics via Hitting Sets". In: *European Conference on Artificial Intelligence (ECAI 10)*. IOS press, pp. 329–334.

Brafman, Ronen I and Carmel Domshlak (2008). "From One to Many: Planning for Loosely Coupled Multi-Agent Systems." In: *International Conference on Automated Planning and Scheduling (ICAPS 08)*, pp. 28–35.

Chen, Yixin, Qiang Lu, and Ruoyun Huang (2008). "Plan-A: A Cost-Optimal Planner Based on SAT-Constrained Optimization". In: *International Planning Competition*.

Cheng, BMW et al. (1999). "Increasing constraint propagation by redundant modeling: an experience report". In: *Constraints* 4.2, pp. 167–192.

Chu, Geoffrey (2011). "Improving Combinatorial Optimization". PhD thesis. Department of Computing and Information Systems, The University of Melbourne.

Ciré, André A., Elvin Coban, and John N. Hooker (2015). "Logic-Based Benders Decomposition for Planning and Scheduling: A Computational Analysis". In: *Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems (COPLAS 15)*, pp. 21–29.

Ciré, André, Elvin Coban, and John N. Hooker (2013). "Mixed Integer Programming vs. Logic-Based Benders Decomposition for Planning and Scheduling". en. In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Vol. 7874. Springer, pp. 325–331.

Clarke, Edmund et al. (2003). "Abstraction and Counterexample-Guided Refinement in Model Checking of Hybrid Systems". In: *International Journal of Foundations of Computer Science* 14.04, pp. 583–604.

Classen, Jens and Gerhard Lakemeyer (2008). "A Logic for Non-Terminating Golog Programs". In: *Principles of Knowledge Representation and Reasoning*. AAAI Press, pp. 589–599.

De Giacomo, Guiseppe, Yves Lespérance, and Hector Levesque (2000). "ConGolog, a Concurrent Programming Language Based on the Situation Calculus". In: *Artificial Intelligence* 121.1-2, pp. 109–169.

Coles, A. I., M. Fox, D. Long, et al. (2008). "A Hybrid Relaxed Planning Graph-LP Heuristic for Numeric Planning Domains". In: *International Conference on Automated Planning and Scheduling (ICAPS 08)*. AAAI Press, pp. 52–59.

Coles, A. J., A. I. Coles, et al. (2010). "Forward-Chaining Partial-Order Planning". In: *International Conference on Automated Planning and Scheduling (ICAPS 10)*. AAAI Press, pp. 42–49.

Coles, Andrew, Maria Fox, Keith Halsey, et al. (2009). "Managing concurrency in temporal planning using planner-scheduler interaction". In: *Artificial Intelligence* 173.1, pp. 1–44.

Cook, Stephen A (1971). "The complexity of theorem-proving procedures". In: *ACM symposium on Theory of computing*. ACM, pp. 151–158.

De Giacomo, G., Y. Lesperance, and H. J. Levesque (2000). "ConGolog, a Concurrent Programming Language Based on the Situation Calculus". In: *Artificial Intelligence* 121.1-2, pp. 109–169.

Dershowitz, Nachum, Ziyad Hanna, and Alexander Nadel (2006). "A Scalable Algorithm for Minimal Unsatisfiable Core Extraction". In: *Theory and Applications of Satisfiability Testing - SAT 2006*. 4121. Springer, pp. 36–41.

Desaulniers, Guy, Jacques Desrosiers, and Marius M. Solomon, eds. (2005). *Column Generation*. Springer.

Edelkamp, Stefan, Peter Kissmann, and Alvaro Torralba (2012). "Symbolic A* Search with Pattern Databases and the Merge-and-Shrink Abstraction". In: *European Conference on Artificial Intelligence (ECAI 12)*. IOS Press, pp. 306–311.

Eén, Niklas and Niklas Sörensson (2004). "An Extensible SAT-Solver". In: *Theory and Applications of Satisfiability Testing (SAT 04)*. Vol. 2919. Springer, pp. 502–518.

Eyerich, Patrick, Robert Mattmüller, and Gabriele Röger (2009). "Using the Context-enhanced Additive Heuristic for Temporal and Numeric Planning". In: *International Conference on Automated Planning and Scheduling (ICAPS 09)*.

Fazel-Zarandi, Mohammad M. and J. Christopher Beck (2011). "Using Logic-Based Benders Decomposition to Solve the Capacity- and Distance-Constrained Plant Location Problem". In: *INFORMS Journal on Computing* 24.3, pp. 387–398.

Felfernig, A., M. Schubert, and C. Zehentner (2012). "An Efficient Diagnosis Algorithm for Inconsistent Constraint Sets". In: *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AI EDAM)* 26.01, pp. 53–62.

Fikes, Richard E and Nils J Nilsson (1971). "STRIPS: A new approach to the application of theorem proving to problem solving". In: *Artificial intelligence* 2.3-4, pp. 189–208.

Fisher, Michael (2004). "Temporal Development Methods for Agent-Based". In: *Autonomous Agents and Multi-Agent Systems (AAMAS 04)* 10.1, pp. 41–66.

Geffner, Hector and Blai Bonet (2013). "A Concise Introduction to Models and Methods for Automated Planning". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 8.1, pp. 1–141.

Geoffrion, Arthur M. and Glenn W. Graves (1974). "Multicommodity Distribution System Design by Benders Decomposition". In: *Management Science* 20.5, pp. 822–844.

Ghanbari Ghooshchi, Nina et al. (2017). "Encoding Domain Transitions for Constraint-Based Planning". In: *Journal of Artificial Intelligence Research* 58, pp. 905–966.

Giunchiglia, Enrico and Marco Maratea (2010). "Introducing Preferences in Planning As Satisfiability". In: *Journal of Logic and Computation* 21.2, pp. 205–229.

Gregory, Peter et al. (2012). "Planning Modulo Theories: Extending the Planning Paradigm". In: *International Conference on Automated Planning and Scheduling (ICAPS 12)*. AAAI Press, pp. 65–73.

Gurobi Optimization, Inc. (2013). *Gurobi Optimizer Reference Manual*. URL: `http://www.gurobi.com`.

Haslum, Patrik et al. (2009). "$h^m(P) = h^1(P^m)$: Alternative Characterisations of the Generalisation From h^{max} To h^m." In: *International Conference on Automated Planning and Scheduling (ICAPS 09)*. Vol. 9, pp. 354–357.

Haslum, Patrik (2013). "Heuristics for Bounded-Cost Search". In: *International Conference on Automated Planning and Scheduling (ICAPS 13)*, pp. 312–316.

Haslum, Patrik, John Slaney, and Sylvie Thiébaux (2012a). "Incremental Lower Bounds for Additive Cost Planning Problems". In: *International Conference on Automated Planning and Scheduling (ICAPS 12)*. AAAI Press, pp. 74–82.

— (2012b). "Minimal Landmarks for Optimal Delete-Free Planning". In: *International Conference on Automated Planning and Scheduling (ICAPS 12)*. AAAI Press, pp. 353–357.

Heinz, Stefan, Wen-Yang Ku, and J. Christopher Beck (2013). "Recent Improvements Using Constraint Integer Programming for Resource Allocation and Scheduling". In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 13)*. Vol. 7874. Springer, pp. 12–27.

Helmert, Malte and Carmel Domshlak (2009). "Landmarks, Critical Paths and Abstractions: What's the Difference Anyway?" In: *International Conference on Automated Planning and Scheduling (ICAPS 09)*. AAAI Press, pp. 162–169.

Helmert, Malte, Patrik Haslum, et al. (2014). "Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces". In: *Journal of the ACM* 61.3, 16:1–16:63.

Hemery, Fred et al. (2006). "Extracting MUCs from Constraint Networks". In: *Proceedings of the European Conference on Artificial Intelligence (ECAI 06)*, pp. 113–117.

Hoffmann, Jörg et al. (2007). "SAT Encodings of State-Space Reachability Problems in Numeric Domains". In: *International Joint Conference on Artificial Intelligence (IJCAI 07)*, pp. 1918–1923.

Hooker, J. N. (2007). "Planning and Scheduling by Logic-Based Benders Decomposition". In: *Operations Research* 55.3, pp. 588–602.

Hooker, John N. and Greger Ottosson (2003). "Logic-Based Benders Decomposition". In: *Mathematical Programming* 96.1, pp. 33–60.

Huang, Ruoyun, Yixin Chen, and Weixiong Zhang (2012). "SAS$^+$ Planning As Satisfiability". In: *Journal of Artificial Intelligence Research* 43.1, pp. 293–328.

Imai, Tatsuya and Alex Fukunaga (2014). "A Practical, Integer-Linear Programming Model for the Delete-Relaxation in Cost-Optimal Planning". In: *European Conference on Artificial Intelligence (ECAI 14)*. Vol. 263. IOS Press, pp. 459–464.

Jampani, Jagadish and Scott J. Mason (2008). "Column Generation Heuristics for Multiple Machine, Multiple Orders Per Job Scheduling Problems". In: *Annals of Operations Research* 159.1, pp. 261–273.

Joncour, Cédric et al. (2010). "Column Generation Based Primal Heuristics". In: *Electronic Notes in Discrete Mathematics* 36, pp. 695–702.

Junker, U. (2004). "QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems". In: *National Conference on Artificial Intelligence*. AAAI Press / The MIT Press, pp. 167–172.

Katz, Michael and Carmel Domshlak (2008). "Optimal Additive Composition of Abstraction-Based Admissible Heuristics." In: *International Conference on Automated Planning and Scheduling (ICAPS 08)*. AAAI Press, pp. 174–181.

Kautz, Henry A. and Bart Selman (1992). "Planning As Satisfiability". In: *European Conference on Artificial Intelligence (ECAI 92)*. Citeseer, pp. 359–363.

Kelly, Ryan F. and Adrian R. Pearce (2006). "Towards High Level Programming for Distributed Problem Solving". In: *International Conference on Intelligent Agent Technology (IAT-06)*. IEEE, pp. 490–497.

Keyder, Emil Ragip, Jörg Hoffmann, and Patrik Haslum (2012). "Semi-Relaxed Plan Heuristics". In: *International Conference on Automated Planning and Scheduling (ICAPS 12)*. AAAI Press, pp. 128–136.

Keyder, Emil and Hector Geffner (2009). "Soft Goals Can Be Compiled Away". In: *Journal of Artificial Intelligence Research* 36.1, pp. 547–556.

Korf, Richard E. (1985). "Depth-First Iterative-Deepening". In: *Artificial Intelligence* 27.1, pp. 97–109.

Korf, Richard E. (1990). "Real-Time Heuristic Search". In: *Artificial Intelligence* 42.2-3, pp. 189–211.

Lemaréchal, Claude (2001). "Lagrangian Relaxation". In: *Computational Combinatorial Optimization*. Vol. 2241. Springer, pp. 112–156.

Liffiton, Mark H. and Ammar Malik (2013). "Enumerating Infeasibility: Finding Multiple MUSes Quickly". In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Vol. 7874. Springer, pp. 160–175.

Lipovetzky, Nir and Hector Geffner (2009). "Inference and Decomposition in Planning Using Causal Consistent Chains". In: *International Conference on Automated Planning and Scheduling (ICAPS 09)*, pp. 217–224.

Marques-Silva, Joao, Mikoláš Janota, and Anton Belov (2013). "Minimal Sets over Monotone Predicates in Boolean Formulae". In: *Computer Aided Verification*. Vol. 8044. Springer, pp. 592–607.

Marques-Silva, Joao and Inês Lynce (2007). "Towards Robust CNF Encodings of Cardinality Constraints". In: *Principles and Practice of Constraint Programming (CP 07)*. Vol. 4741. Springer, pp. 483–497.

Marriott, Kim and Peter J. Stuckey (1998). *Programming with Constraints: An Introduction*. Cambridge, Mass.: MIT Press.

Mears, Christopher et al. (2014). "Modelling with option types in minizinc". In: *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*. Springer, pp. 88–103.

Mehlhorn, Kurt and Mark Ziegelmann (2000). "Resource Constrained Shortest Paths". In: *Algorithms (ESA 00)*. Vol. 1879. Springer, pp. 326–337.

Moskewicz, Matthew W et al. (2001). "Chaff: Engineering an efficient SAT solver". In: *Design Automation Conference*. ACM, pp. 530–535.

Nakhost, Hootan, Jörg Hoffmann, and Martin Müller (2012). "Resource-Constrained Planning: A Monte Carlo Random Walk Approach". In: *International Conference on Automated Planning and Scheduling (ICAPS 12)*, pp. 181–189.

Nareyek, Alexander et al. (2005). "Constraints and AI Planning". In: *IEEE Intelligent Systems* 20.2, pp. 62–72.

Nethercote, N. et al. (2007). "MiniZinc: Towards a Standard CP Modelling Language". In: *International Conference on Principles and Practice of Constraint Programming (CP 07)*. Vol. 4741. Springer, pp. 529–543.

Nieuwenhuis, Robert, Albert Oliveras, and Cesare Tinelli (2006). "Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T)". In: *Journal of the ACM* 53.6, pp. 937–977.

Ohrimenko, Olga, Peter J. Stuckey, and Michael Codish (2009). "Propagation via Lazy Clause Generation". In: *Constraints* 14.3, pp. 357–391.

Papadimitriou, Christos H. and Kenneth Steiglitz (1982). *Combinatorial Optimization: Algorithms and Complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Penberthy, J Scott, Daniel S Weld, et al. (1992). "UCPOP: A Sound, Complete, Partial Order Planner for ADL". In: *Conference on Principles of Knowledge Representation and Reasoning (KR 92)*. Vol. 92, pp. 103–114.

Pisinger, David and Stefan Ropke (2010). "Large Neighborhood Search". In: *Handbook of Metaheuristics*. Springer, pp. 399–419.

Pommerening, Florian and Malte Helmert (2013). "Incremental LM-Cut". In: *International Conference on Automated Planning and Scheduling (ICAPS 13)*. AAAI Press, pp. 162–170.

Pommerening, Florian, Malte Helmert, and Blai Bonet (2017). "Higher-Dimensional Potential Heuristics for Optimal Classical Planning". In: *AAAI Conference on Artificial Intelligence (AAAI 17)*. AAAI Press, pp. 3636–3643.

Pommerening, Florian, Gabriele Röger, et al. (2014). "LP-Based Heuristics for Cost-Optimal Planning". In: *International Conference on Automated Planning and Scheduling (ICAPS 14)*.

Prud'homme, Charles, Xavier Lorca, and Narendra Jussien (2014). "Explanation-Based Large Neighborhood Search". In: *Constraints* 19.4, pp. 339–379.

Reinefeld, Alexander and T. Anthony Marsland (1994). "Enhanced iterative-deepening search". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16.7, pp. 701–710.

Reiter, Raymond (2001). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.

Rintanen, Jussi (2006). "Compact Representation of Sets of Binary Constraints". In: *European Conference on Artificial Inteligence (ECAI 06)*. IOS Press, pp. 143–147.

— (2009). "Planning and SAT". In: *Handbook of Satisfiability*. Chap. 15, pp. 483–504.

— (2012). "Planning As Satisfiability: Heuristics". In: *Artificial Intelligence* 193, pp. 45–86.

Rintanen, Jussi, Keijo Heljanko, and Ilkka Niemelä (2006). "Planning As Satisfiability: Parallel Plans and Algorithms for Plan Search". In: *Artificial Intelligence* 170.12, pp. 1031–1080.

Robinson, Nathan, Charles Gretton, and Duc-Nghia Pham (2008). "Co-plan: Combining SAT-Based Planning with Forward-Search". In: *International Planning Competition*.

Robinson, Nathan, Charles Gretton, DucNghia Pham, and Abdul Sattar (2010). "Partial Weighted MaxSAT for Optimal Planning". In: *Pacific Rim International Conference on Artificial Intelligence*. Vol. 6230. Springer, pp. 231–243.

Röger, Gabriele and Bernhard Nebel (2007). "Expressiveness of ADL and Golog: Functions Make a Difference". In: *National Conference on Artificial Intelligence - Volume 2*. AAAI Press, pp. 1051–1056.

Russell, Stuart J. and Peter Norvig (2003). *Artificial Intelligence: A Modern Approach*. 2nd. Upper Saddle River, N.J. ; [Great Britain]: Prentice Hall.

Ryan, David M. and Brian A. Foster (1981). "An Integer Programming Approach to Scheduling". In: *Computer Scheduling of Public Transport Urban Passenger Vehicle and Crew Scheduling*. Amsterdam, The Netherlands: North Holland, pp. 269–280.

Sacerdoti, Earl D (1975). *A structure for plans and behavior*. Tech. rep. DTIC Document.

Sardina, Sebastian and Giuseppe De Giacomo (2009). "Composition of ConGolog Programs". In: *International Joint Conference on Artificial Intelligence (IJCAI 09)*, pp. 904–910.

Schoenauer, Marc, Pierre Savéant, and Vincent Vidal (2006). "Divide-and-Evolve: A New Memetic Scheme for Domain-Independent Temporal Planning". In: *Evolutionary Computing in Combinatorial Optimization*. Vol. 3906. Springer, pp. 247–260.

Schutt, Andreas et al. (2013). "Solving RCPSP/max by Lazy Clause Generation". In: *Journal of Scheduling* 16, pp. 273–289.

Seipp, Jendrik and Malte Helmert (2013). "Counterexample-Guided Cartesian Abstraction Refinement". In: *International Conference on Automated Planning and Scheduling (ICAPS 13)*. AAAI Press, pp. 347–351.

Shanahan, Murray (2000). "An Abductive Event Calculus Planner". In: *Journal of Logic Programming* 44, pp. 207–239.

Sinz, Carsten (2005). "Towards an Optimal CNF Encoding of Boolean Cardinality Constraints". In: *Principles and Practice of Constraint Programming (CP 05)*. Vol. 3709. Springer, pp. 827–831.

Steinmetz, Marcel and Jörg Hoffmann (2016). "Towards Clause-Learning State Space Search: Learning to Recognize Dead-Ends". In: *AAAI Conference on Artificial Intelligence (AAAI 16)*, pp. 760–768.

Suda, Martin (2014). "Property Directed Reachability for Automated Planning". In: *Journal of Artificial Intelligence Research* 50, pp. 265–319.

Tate, Austin (1977). "Generating project networks". In: *international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc., pp. 888–893.

Torralba, Alvaro et al. (2014). "SymBA^*: A Symbolic Bidirectional A^* Planner". In: *International Planning Competition*. AAAI Press, pp. 105–108.

Tran, Tony T. and J. Christopher Beck (2012). "Logic-Based Benders Decomposition for Alternative Resource Scheduling with Sequence Dependent Setups". In: *European Conference on Artificial Intelligence (ECAI 12)*. Vol. 242. IOS Press, pp. 774–779.

Van Den Briel, Menkes et al. (2007). "An LP-Based Heuristic for Optimal Planning". In: *Principles and Practice of Constraint Programming (CP 07)*. Vol. 4741. Springer, pp. 651–665.

Vidal, Vincent (2011). "YAHSP2: Keep It Simple, Stupid". In: *The 2011 International Planning Competition: Description of Participating Planners, Deterministic Track*, pp. 83–90.

Vidal, Vincent and Héctor Geffner (2006). "Branching and Pruning: An Optimal Temporal POCL Planner Based on Constraint Programming". In: *Artificial Intelligence* 170.3, pp. 298–335.

Wehrle, Martin and Jussi Rintanen (2007). "Planning as Satisfiability with Relaxed $\exists$-Step Plans". In: *AI 2007: Advances in Artificial Intelligence: 20th Australasian Joint Conference*. Vol. 4830. Springer, pp. 244–253.

Zhang, Lei et al. (2013). "Planning with Multi-Valued Landmarks". In: *AAAI Conference on Artificial Intelligence (AAAI 13)*, pp. 1653–1654.