Technical Report

UCAM-CL-TR-822
ISSN 1476-2986

Number 822

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Planning with preferences using maximum satisfiability

Richard A. Russell

October 2012

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted September 2011 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Gonville and Caius College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Summary

The objective of automated planning is to synthesise a plan that achieves a set of goals specified by the user. When achieving every goal is not feasible, the planning system must decide which ones to plan for and find the lowest cost plan. The system should take as input a description of the user's preferences and the costs incurred through executing actions. Goal utility dependencies arise when the utility of achieving a goal depends on the other goals that are achieved with it. This complicates the planning procedure because achieving a new goal can alter the utilities of all the other goals currently achieved.

In this dissertation we present methods for solving planning problems with goal utility dependencies by compiling them to a variant of satisfiability known as weighted partial maximum satisfiability (WPMax-SAT). An optimal solution to the encoding is found using a general-purpose solver. The encoding is constructed such that its optimal solution can be used to construct a plan that is most preferred amongst other plans of length that fit within a prespecified horizon. We evaluate this approach against an integer programming based system using benchmark problems taken from past international planning competitions.

We study how a WPMax-SAT solver might benefit from incorporating a procedure known as survey propagation. This is a message passing algorithm that estimates the probability that a variable is constrained to be a particular value in a randomly selected satisfying assignment. These estimates are used to influence variable/value decisions during search for a solution. Survey propagation is usually presented with respect to the satisfiability problem, and its generalisation, SP(y), with respect to the maximum satisfiability problem. We extend the argument that underpins these two algorithms to derive a new set of message passing equations for application to WPMax-SAT problems. We evaluate the success of this method by applying it to our encodings of planning problems with goal utility dependencies.

Our results indicate that planning with preferences using WPMax-SAT is competitive and sometimes more successful than an integer programming approach – solving two to three times more subproblems in some domains, while being outperformed by a smaller margin in others. In some domains, we also find that using information provided by survey propagation in a WPMax-SAT solver to select variable/value pairs for the earliest decisions can, on average, direct search to lower cost solutions than a uniform sampling strategy combined with a popular heuristic.

Acknowledgments

I would like to thank my supervisor Dr. Sean Holden for providing helpful feedback, kind encouragement and support throughout my graduate studies. I would also like to thank my thesis examiners, Dr. Mateja Jamnik and Professor Barry O'Sullivan for their encouraging words and insightful suggestions.

I am grateful to the Computer Laboratory, Gonville and Caius College, and the Engineering and Physical Sciences Research Council (EPSRC) for providing financial assistance during my time as a graduate here. This work has been been funded by a Doctoral Training Account provided by the EPSRC.

I would like to thank Ashutosh Mahajan for providing technical assistance with using the SYMPHONY5.1 source code; Menkes van den Briel for granting me permission to use the IPPLAN source code on which I reimplemented Do et al.'s (2007) encoding, and for answering my queries related to the source code; and Malte Helmert for providing source code for the SAS$^+$ translator.

I have enjoyed many interesting discussions with colleagues in the Computer Laboratory during my time here. Special thanks go to Christian Richardt, Andrew Lewis, Ramsey Khalaf, Ian Davies, Christopher Jenkins and Nicholas Pilkington.

I am forever grateful for the loving support of my family and friends. I am especially grateful to my grandmother, Joan, and my parents, Suzanne and Mark, who have always supported me in pursuing my interests. Without their encouragement, patience and understanding, this dissertation would not have been possible.

Contents

Chapter 1

Introduction

Planning can be described as the process of identifying a sequence of actions to perform that will result in the successful completion of a task. A wide variety of problems match this description, but amongst those problems there can be substantial differences in their characteristics. Common to all planning problems is the idea that a plan must have some net useful effect that accomplishes an objective set by a user, which is usually described as a collection of goals. The plan may then be executed by the user themselves or by an autonomous agent acting on the user's behalf. The development of automated planning systems that take as input a description of a planning problem and output solutions has been a long-standing goal of artificial intelligence.

Historically, planning research focussed on methods for finding plans that achieve all of a problem's goals, but since the turn of the century there has been a growing research trend to place emphasis on finding high quality plans. An understanding of the quality of a plan is needed in order to make decisions that involve trade-offs during planning. Such scenarios arise in realistic problems either because there are more goals than it is feasible to achieve with limited resources or because some goals are not important enough to warrant the cost involved in achieving them. These type of problems are referred to as *oversubscription* or *partial satisfaction* problems.

Faced with one of these problems, asking the user to choose a feasible subset of goals from an oversubscription problem goes against the ambition of automated planning. A simple approach that maximises the number of goals achieved is not sufficient either since the importance of goals to the user can vary. The planning system must be supplied with a description of the *goal preferences* that a user has for the problem so that it can make its own choices that match the user's requirements.

In many scenarios, the user may prefer that a group of goals are achieved together by a plan because they belong to a relationship that the user is interested in. Conversely, it is also reasonable that a user may want to avoid plans that achieve more than one goal

in a specific group because they overlap in their usefulness and act as substitutes for one another. In these cases, the user will feel a degree of satisfaction with a solution according to how well it meets their preferences. A measure of the degree of satisfaction is often called the *utility*. In the above case, the utility of a goal is dependent on the collection of goals that are achieved, and we will refer to such situations as exhibiting *goal utility dependencies*.

Alternative types of preferences are needed in situations where priorities might make it more desirable to achieve a set of goals in a certain order. Similarly, there may be facilities that are only available at certain times, and when they are available, the user would prefer that these facilities are used over alternatives. Common to these types of preferences is the idea that some plans may be preferable to others even if they achieve the same set of goals for the same cost. Collectively, such preferences are referred to as *trajectory preferences*.

Once a set of preferences has been specified, we might ask the planning system to synthesise a most preferred plan. This is likely to be a much more computationally intensive procedure than those that perform classical planning, as it is not sufficient to simply find a plan: for each plan the system encounters, it must either find a better one or prove that a more preferred plan does not exist. Only when the latter has been established can an optimal planning system terminate. Thus, the search procedure must engage in a systematic exploration of the solution space. For practical problems, a simple exhaustive search covering all possible plans is infeasible. Therefore it is important to be able to establish when a particular branch of the search cannot contain any better solutions and to prune that branch accordingly. Without the ability to exclude large portions of valid plans by pruning, we will be defeated by the exponential growth in the number of valid plans.

A compilation approach to planning encodes the problem in another framework such as *constraint satisfaction* or *satisfiability*. A general purpose solver from that framework is then used to solve the encoding. The encoding is specially constructed so that once a solution to the encoding is obtained, a solution to the original planning problem can be extracted from it. Compilation approaches have been very successful in classical planning: a SAT-based planner was the winner of the propositional optimal[1] track in the fourth (2004) and fifth (2006) International Planning Competitions. Rintanen (2010) has shown that incorporating a planning specific heuristic into a SAT solver leads to a SAT-based planner that can outperform some of the best satisficing classical planners.

Many of the target frameworks used for a compilation approach to planning have opti-

[1] In these planning competitions, an optimal plan had the smallest makespan over all other plans. Since then, there has been a departure from this definition to use action costs instead to define the optimal plan, which we favour and focus on in this dissertation.

misation variants. These typically involve the definition of an optimisation function that is to be minimised or maximised during the solution procedure. This makes it possible to construct encodings such that their optimal solutions correspond to optimal plans. The optimisation variant of the satisfiability problem is called maximum satisfiability (Max-SAT).

In this dissertation, we concentrate on a variation of the Max-SAT problem known as *weighted partial Max-SAT*. A problem in this class consists of hard and soft clauses, where soft clauses have an integer valued weight associated with them. An optimal solution to such a problem is a truth assignment that satisfies all hard clauses and maximises the sum of weights of satisfied soft clauses. Research into these solvers is in its infancy: the first Max-SAT evaluation was held in 2006 as part of the International Conference on Theory and Applications of Satisfiability Testing; the weighted partial Max-SAT category was added to the evaluation in 2007.

Recently a new method called *survey propagation* (Mézard and Zecchina 2002) has been proposed, originating from ideas developed within the statistical physics community, that can be used to solve a higher proportion of the hardest satisfiability problems in a class of problems known as random k-SAT. Survey propagation is used to estimate, for each variable, the probability that it is constrained to take the value *True*, the value *False* or is unconstrained in a randomly selected satisfying truth assignment. These are then used to identify strongly biased variables: those which are much more likely to be assigned one value than another. The most strongly biased variables are set to their more likely values in a process known as *decimation*. This results in a simplified problem that is hopefully easier to solve by a more conventional method.

The underlying ideas behind survey propagation can be adapted to the Max-SAT problem, which leads to an algorithm known as SP(y) (Battaglia et al. 2004). An optimal solution to a Max-SAT problem is a truth assignment that satisfies the greatest number of clauses. Instead of treating all assignments as equal, the SP(y) method defines a distribution over truth assignments which has its highest value for assignments that satisfy the greatest number of clauses. This alters the probability estimates in order to direct the decimation procedure towards truth assignments that satisfy as many clauses as possible.

This dissertation presents an approach to handling preferences within a Max-SAT framework. We develop an encoding scheme for solving STRIPS planning problems with goal utility dependencies and action costs as a weighted partial Max-SAT problem. We then adapt the survey propagation technique to handle weighted partial Max-SAT problems, which consist of hard and soft clauses together with non-uniform integer weights. This leads to a new set of equations, which we refer to as the WPSP(y) equations. We then use probability estimates derived from this method as heuristic guidance in a weighted partial Max-SAT solver to test the effectiveness of the theory.

1.1 Dissertation outline

In Chapter 2 we present an overview of background material discussing planning models, weighted partial Max-SAT and encoding planning as satisfiability. Alternative approaches are briefly discussed for completeness. Chapter 3 defines goal utility dependencies and relates this type of utility function to the existing *net benefit* model for planning with preferences. Then we present a new procedure for encoding net benefit planning with goal utility dependencies as weighted partial Max-SAT.

In Chapter 4 we introduce the survey propagation method and its application. We summarise a derivation of the survey propagation method which illustrates the reason for its success. Building upon this, we present a derivation of a new set of equations which we refer to as the weighted partial survey propagation (WPSP(y)) equations. These are more general than the SP(y) equations in that they consider the case where clauses may be hard or soft, and may have arbitrary integer weights.

In Chapter 5 we present two main sets of experimental results that evaluate the success of our new techniques described in Chapters 3 and 4. These aim to test the following hypotheses,

- Representing planning problems with goal utility dependencies in weighted partial Max-SAT allows problems to be solved more quickly than can be done when using an integer programming based represention.

- Using bias estimates provided by WPSP(y) to guide a weighted partial Max-SAT solver directs the search towards solutions of higher quality than using uniform sampling from the same set of variable/value pairs. Moreover, we expect to observe an improvement over the default heuristic that is used in the weighted partial Max-SAT solver that we study.

The first hypothesis is motivated by empirical observations indicating that integer programming encodings of STRIPS planning do not perform as well as SAT encodings. The second hypothesis is motivated by an understanding of what the SP(y) and WPSP(y) methods attempt to do. In summary, they define a probability distribution that is peaked around assignments that have the highest sum of satisfied clause weights. Through a series of approximations, the methods attempt to estimate marginal values of this distribution. If these estimates are accurate, then variables that have a high marginal probability of being *True* or *False* are likely to take that value in a truth assignment selected at random according to that distribution. If the distribution is sufficiently peaked, then a truth assignment selected according to that distribution is likely to be a low cost solution.

Therefore, setting variables that are strongly biased towards one value or another should direct the solver towards a low cost solution.

We make our concluding remarks and discuss directions for future work in Chapter 6. In summary, this dissertation makes the following original contributions:

- The first encoding scheme for representing STRIPS net benefit planning problems with goal utility dependencies as weighted partial Max-SAT formulas. The encoding scheme is flexible enough to also represent partial satisfaction problems which do not exhibit goal utility dependencies, such as the *PSP net benefit* class of problems from the 2008 *International Planning Competition*.

- An implementation built upon SATPLAN, which we call MSATPLAN, of a planning system that produces encodings according to the above scheme. MSATPLAN then uses a general-purpose weighted partial Max-SAT solver to find an optimal solution to the encoding, from which a plan is extracted that is optimal amongst all other plans of equal or smaller makespan[2]. We conduct an empirical evaluation of the performance of MSATPLAN against a planning system that solves planning problems with goal utility dependencies via an integer programming encoding.

- A derivation of a new set of equations called the WPSP(y) equations, which can be applied to any weighted partial Max-SAT problem, not just encodings of planning problems. This allows the survey propagation method of decimation to be applied to our encodings.

- An implementation of an optimal weighted partial Max-SAT solver based upon an existing successful solver, MINIMAXSAT. Our implementation uses the WPSP(y) equations to perform a decimation procedure in the initial stages of search for an optimal truth assignment to a weighted partial Max-SAT problem.

- An empirical evaluation that studies the effects of performing this decimation using the WPSP(y) equations within our weighted partial Max-SAT solver. We compare against other heuristics that are used in the successful MINIMAXSAT solver. Our evaluation is performed using our encodings of STRIPS net benefit planning problems with goal utility dependencies; however, the approach has broader applicability, and, with only slight modifications, it could potentially be used on any weighted partial Max-SAT problem.

[2]The makespan of a plan relates to the number of 'parallel steps' that it executes. A precise definition is presented in the next chapter.

The material presented in Chapter 3 and the empirical results related to that work, presented in Chapter 5, are an expanded version of material that was presented as a full technical paper (Russell and Holden 2010) at the *International Conference on Automated Planning and Scheduling* (ICAPS) 2010.

Chapter 2

Background

In this chapter we present an overview of the formalisms that we will work with throughout the rest of the dissertation. We discuss relevant alternative approaches from the literature. A more detailed literature survey related to preferences and survey propagation is contained in Chapters 3 and 4, respectively.

2.1 Planning models

When describing planning models, we work with an abstraction of the world. The world can exist in a collection of *states*. Within the world there are *objects* and *agents* that can perform actions. Performing an action causes a *transition* between states.

The field of planning is separated into subfields based upon various simplifying assumptions about these abstractions. For instance, the world can either be static, meaning that transitions between states only occur when actions are performed, or it can be dynamic, which requires agents to repeatedly sense their environment to check for changes.

This dissertation is concerned with the subfield often referred to as *classical planning*. In this subfield, a single agent operates in a world that is finite and static; states are fully observable; actions have deterministic effects and when executed, their effects are instantaneous.

Within classical planning, there are several formalisms for describing planning problems, of which we will discuss the influential STRIPS language, the community standard PDDL, and the multi-valued state representation SAS^+.

2.1.1 STRIPS

The STRIPS (STanford Research Institute Problem Solver) language (Fikes and Nilsson 1971) was designed to be simple enough to be used for efficient inference but expressive enough to describe useful planning problems. It became the standard model for planning research in subsequent decades and many extensions have been made to the model to handle aspects of the real-world such as conditional effects, durative actions and uncertainty.

In STRIPS, properties of the world are described using formulas written in first order logic; however, the salient aspect of the STRIPS language, and a large contributor to the formalism's success, is its simple action representation. Each action is described using a precondition formula, which describes what must be true in the world in order for the action to be applicable, and a set of effects, which describe how the world changes if the action is applied. Effects are represented in the form of an *add* and a *delete* list. The add list describes the properties of the world that might not have been true before but will be true after the action is executed. Similarly, the delete list describes those properties that might have been true before but will be false after its execution.

Let us illustrate how STRIPS can be used to model a planning problem that requires the moving of people between cities. Two people, Alison and Adam, can move between the cities Paris and London. Alison is initially in London and Adam is initially in Paris. We can model the location of a single person in first order logic using the predicate $At(p, l)$, which is true if person p is at location l, where p and l are variables. Note that in first order logic, the set of objects in the world for this problem is $\{Alison, Adam, Paris, London\}$. Let us assume that constant symbols of the same name are mapped to these objects. We want to avoid instantiations that are valid in first order logic but are semantically meaningless to us, for example, $At(Paris, Alison)$. Thus, most implementations of STRIPS incorporate a typing system, which could be replicated in first order logic by the use of unary predicates, for example, $City(Paris)$ denotes $Paris$ as a city and $Person(Alison)$ denotes $Alison$ as a person.

For compactness, actions are typically described using action schema that accept parameters, for example, $\mathsf{Go}(p : Person,\ l_1 : City,\ l_2 : City)$, is an action schema for moving a person from one city to another. It has a precondition $At(p, l_1)$, an add effect $At(p, l_2)$ and a delete effect $At(p, l_1)$, where we have used ':' to indicate typing of variables. The typing restrictions avoid semantically erroneous instantiations of the action schema.

A problem will include a description of the initial state. This will include a typed list of objects and a list of predicates that are true initially. The domain will specify the actions that can be performed. Together, this implicitly describes the set of atoms that can be generated through repeated application of actions. Note that some valid instantiations of

predicates may not be reachable from the initial state, and this has implications for the choice of state representation.

A STRIPS problem can be translated to a propositional STRIPS problem by considering all possible ground atoms of the first order theory, which correspond to the possible facts, and ground instantiations of action schemas. For reasons of convenience, we will work with this formalism, although the reader should be aware that this translation must take place, and can lead to large problem specifications.

In propositional STRIPS, a classical planning problem is described by a 4-tuple $\langle \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ where

- $\mathcal{F}$ is a finite set of ground atomic formulas that correspond to the facts about the world,
- $\mathcal{I} \subseteq \mathcal{F}$ is the initial state, which are the facts that are initially true,
- $\mathcal{G} \subseteq \mathcal{F}$ is the subset of facts that are the goals of the problem,
- $\mathcal{A}$ is a set of ground actions. Each action $a \in \mathcal{A}$ has
  - a precondition set $\mathit{Pre}(a) \subseteq \mathcal{F}$ of ground atomic formulas that must all be true for a to be applicable,
  - a set $\mathit{Add}(a) \subseteq \mathcal{F}$ of add effects which are the facts that are made true if a is executed,
  - a set $\mathit{Del}(a) \subseteq \mathcal{F}$ of delete effects which are the facts that are made false if a is executed. We require that $\mathit{Add}(a) \cap \mathit{Del}(a) = \emptyset$.

States of the world are described as sets $S \subseteq \mathcal{F}$ which denote the facts that are true in the state: a fact $f \in \mathcal{F}$ is true in state S if and only if $f \in S$. An action $a \in \mathcal{A}$ is applicable in state S if and only if $\mathit{Pre}(a) \subseteq S$. Note that this forces the preconditions of actions to be positive literals; alternative models such as ADL allow actions to have negative literals as preconditions. A plan is an ordered sequence of actions $(a_1, \ldots, a_m)$ with $()$ denoting the empty plan that contains no actions.

The result of executing a plan P from the starting state S is $\mathit{Result}(P, S)$ which is recursively defined as

$$\mathit{Result}((), S) = S, \tag{2.1}$$

$$\mathit{Result}((a), S) = \begin{cases} \mathit{Add}(a) \cup (S \setminus \mathit{Del}(a)) & \text{if } \mathit{Pre}(a) \subseteq S, \\ S & \text{otherwise,} \end{cases} \tag{2.2}$$

$$\mathit{Result}((a_1, \ldots, a_m), S) = \mathit{Result}((a_2, \ldots, a_m), \mathit{Result}((a_1), S)). \tag{2.3}$$

Note that we have used Bylander's convention that executing an action in a state in which it is not applicable results in no change in the state. A plan P is a *solution* to a planning problem $\langle \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ if and only if $\mathcal{G} \subseteq \mathit{Result}(P, \mathcal{I})$.

Propositional STRIPS planning is PSPACE-complete (Bylander 1994). If actions are allowed to have only positive effects then the problem is NP-complete. Severe restrictions must be placed on actions in order to make the planning problem polynomial time solvable: for example, if each action can have only positive preconditions and one effect.

2.1.2 SAS$^+$

The SAS$^+$ (Simplified Action Structures) (Bäckström and Nebel 1995) formalism was initially developed to study tractable restrictions for planning problems. A state is represented using a set of multi-valued variables. Constraints, such as action preconditions and effects, assert that variables should take on particular values from their domains. Each action specifies, in addition to its preconditions and effects, prevail-conditions, which describe the values of precondition variables that go unchanged by the operator.

Edelkamp and Helmert (1999) present an automatic conversion procedure from STRIPS planning problems to a SAS$^+$ representation. As previously alluded to, not all ground instantiations of predicates are reachable; moreover, for some groups of predicates, only one member of that group can be true at any one time. Modelling locations provides a good example of this. $\mathit{At}(\mathit{Alison}, \mathit{Paris})$ and $\mathit{At}(\mathit{Alison}, \mathit{London})$ cannot be simultaneously true for the following reason: $\mathit{At}(\mathit{Alison}, \mathit{London})$ is the only one of the two initially true and whenever the Go action adds a fact of the form $\mathit{At}(\mathit{Alison}, l')$, it deletes the currently true fact $\mathit{At}(\mathit{Alison}, l)$. So Alison is only ever at one location. Thus, we could represent the predicate that is true by enumerating the possible locations and storing the integer corresponding to the predicate that is true. Actions then cause changes in the value of such variables. This is an example of one of the ways Edelkamp and Helmert convert the STRIPS problem to a multi-valued representation.

This multi-valued state representation is used as the basis for the integer programming encoding by van den Briel et al. (2008), which we will compare against later in this dissertation.

2.1.3 PDDL

The Planning Domain Definition Language (PDDL) is an attempt to standardise the language used to express planning problems so that they may be shared between researchers. PDDL 1.2 (McDermott et al. 1998) was adopted by the *International Planning Competition* (IPC) to compare the performance of competing planning systems. A problem

definition is made in two parts: the domain file in which requirements, constants and actions are specified, and the problem file in which objects, the initial state and goals are specified. Requirements define particular subsets of the language that a planner must support in order for the problem to be solved. In general, planning systems are not expected to support all features of the language; instead, they must identify the language features they need to model their target application.

PDDL 1.2 allows the user to express STRIPS problems; however, the language allows action preconditions to contain negative literals. This can produce confusion because some planning systems work with a definition of STRIPS – which is used in some textbooks (Russell et al. 1995) – where the preconditions of actions must be positive literals. Hence, the situation can arise where a PDDL problem which advertises only the STRIPS requirement cannot be solved by a STRIPS planning system because of the use of negative literals in action preconditions.

Since the initial version, extensions have been made to the language to move towards modelling more realistic problems and addressing plan quality. PDDL 2.1 (Fox and Long 2003) extends the language to allow temporal and numeric properties to be modelled. Actions can change the value of numeric variables and the applicability of actions can be conditional on functions of values of numeric variables. In particular, it introduces the idea of a plan metric, which can be calculated from the values of these numeric variables, that is to be minimised or maximised.

The temporal model adopted by the language allows durative actions to have discretised or continuous effects. In both cases, an action has a start and an end time. Precondition constraints on actions can be at the start or end of the action; invariant conditions, that must hold throughout the duration of the action, can also be specified.

In the discrete model, changes in numeric variables occur instantaneously at either the start or the end of the action. The continuous variant allows this change to occur gradually in-between the start and end times. To do this, each action is equipped with a special variable that denotes the time that has elapsed since the action started. This can be used together with a rate of change to calculate its value at any point within the interval defined by its start and end times.

PDDL 3.0, extends the language to allow the modelling of trajectory and goal preferences (Gerevini and Long 2006). Trajectory preferences are modelled using modal logic operators in first order formulas in the style of linear temporal logic (Pnueli 1977). This allows the user to specify constraints on plans such as ‘this fact must always be true’ or ‘this fact must be true at least once during the plan’. Modal operators cannot be nested, but there are some operators that replicate the behaviour of some nesting constructs that might be frequently required. One area of weakness in the language is that specifying constraints

on the occurrence of individual actions is not possible: trajectory constraints can only specify properties about states.

PDDL 3.0 allows goals and constraints to be *soft*, which means that it is not imperative that a plan satisfies such a constraint, but it should try to do so where possible. Furthermore, building upon the ability to specify a plan metric that was introduced in PDDL 2.1, violating a soft constraint can incur a penalty, see Figure 2.1. This allows the planner to make decisions that lead to plans that minimise the total penalty incurred as a result of violating soft constraints.

```
(:metric minimize (+ (is-violated DeliveryWithinOneDay)
                     (* 5 (is-violated DeliveryWithinTwoDays))) )
```

Figure 2.1: Weighting soft constraints so that failure to deliver within two days is five times worse than failing to deliver within one day.

2.2 The planning graph

Blum and Furst's seminal work on planning graphs (1997) for STRIPS-based planning has become the cornerstone for many automated planning methods. It is a graphical presentation of a planning problem that uses nodes to denote facts and actions, and edges between nodes to indicate when a fact is a precondition or an effect of an action.

Since fluents change over time and actions can be executed multiple times in a plan, each fact and action carries an extra time argument. Time is divided into integral steps, and a finite number of steps is considered, often referred to as the horizon or *makespan* of the plan. For instance, a planning graph of makespan m divides time into the values $0, 1, 2, \ldots, m$.

The graph consists of alternating layers of ground facts and fully instantiated actions grouped in increasing order of time. The layers are labelled such that the actions in layer t have their effects in fact layer $t + 1$ and their preconditions in fact layer t.

A special action known as a NOOP, short for 'no operation', is used to denote that a fact persists from one fact level to the next. For example, a NOOP for the fact $At(Alison, Paris)$ could be denoted by $\mathsf{NOOP\text{-}At\text{-}Alison\text{-}Paris}_t$ and would have a precondition $At_t(Alison, Paris)$ and an effect $At_{t+1}(Alison, Paris)$, where the subscript indicates the time step for the fact or action.

Part of the reason for the planning graph's success is its use of a limited form of consistency enforcement. This is provided by binary mutual exclusion links, which are referred to as mutexes. Mutexes are linked either between two facts or two actions, within the same

layer. If two facts at level t in the planning graph are connected by a mutex then this indicates that those two facts cannot be simultaneously true at time step t. Similarly, two actions at level t that are marked mutex cannot be simultaneously executed at time step t; however, from the conditions under which we mark facts and actions as mutex, as described below, we will not be able to identify all possible mutexes. Consequently, if two facts or actions are not marked mutex in the planning graph at a given level, this does not imply that they can be simultaneously true or simultaneously executed at that level in a valid plan, respectively.

The construction of a planning graph is carried out as follows. All facts occurring in the initial state are added to the 0th fact layer; there are no mutexes in the 0th fact layer. Then the following steps are iterated for $i = 0, \ldots, m-1$,

- Add to the ith action layer, every ground action that has all its preconditions appearing pairwise non-mutex in fact layer i. For every fact f_i occurring in fact layer i, add to action layer i a NOOP action $a_{f,i}$ which has $\mathit{Pre}(a_{f,i}) = \{f_i\}$ and $\mathit{Eff}(a_{f,i}) = \{f_{i+1}\}$.

- For every action, including NOOP actions, appearing in action layer i, add to fact layer $i+1$ all the facts that are contained in the add lists of at least one action present in action layer i.

The above two rules are iterated using the following conditions for when two facts or actions are mutex. Two actions a and b that occur in layer i of the planning graph are marked mutex if any of the following conditions are true,

- Inconsistent effects – at least one effect of a is the negation of an effect of b, or vice versa.

- Competing needs – a has at least one precondition which is marked mutex at level i with a precondition of b.

- Interference – at least one effect of a is the negation of a precondition of b, or vice versa.

Two facts f and g that occur in layer $i > 0$ of the planning graph are marked mutex if

- Inconsistent support – both f and g cannot be achieved by a single action in layer $i-1$ and all pairwise choices of supporting actions in layer $i-1$ are marked mutex in that layer.

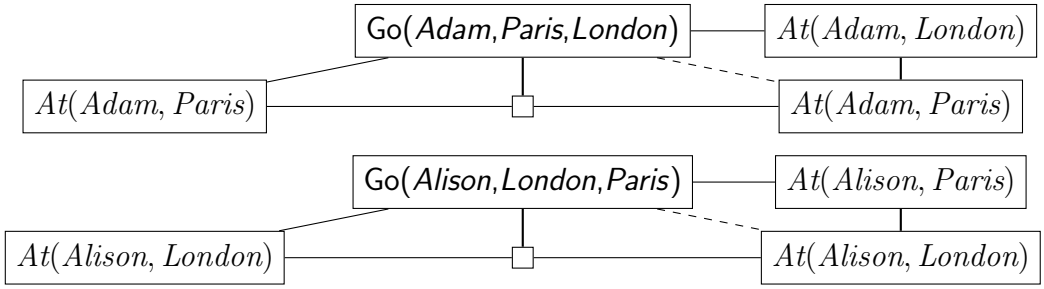


Figure 2.2: An example planning graph. NOOPs are indicated by empty squares. Effects in the delete list are indicated by a dashed line. Mutexes are indicated by thick edges.

A simple backtracking search can be performed on a planning graph to directly solve a problem. If all goals are to be achieved, we first need to expand the planning graph to at least the level at which all goals appear in the planning graph pairwise non-mutex. Then one can perform a backtracking search on the graph to try to find a plan. If that fails, the graph should be extended by an extra level and the process repeated.

A planning graph is said to *level-off* if the expansion of it by one layer adds no actions to the new layer that were not present in the previous action layer. If two facts do not appear in or are marked mutex in the final layer of a levelled-off planning graph, then we can deduce that no plan starting from the same initial state can co-achieve those two facts; hence, the problem is unsolvable. Otherwise, if a planning graph has levelled-off and we have still not found a solution, we may have to continue expansion for a finite number of extra layers in order to either find a solution or determine insolubility – see Blum and Furst (1997) for a termination condition.

At present, the planning graph is seldom used as the only strategy for solving planning problems. Instead, it is often used as the primary data structure for extracting information from planning problems for use with alternative search paradigms. One example of this is the use of the relaxed planning graph in heuristic search.

2.2.1 The relaxed planning graph

A *relaxed* planning graph (Bonet and Geffner 2001) is constructed from a problem in the same way as the normal planning graph with the exception that the delete list of every action is ignored. Note, that this implies that a relaxed planning graph contains no mutexes.

A relaxed plan can be extracted from a relaxed planning graph in polynomial time (Hoffmann and Nebel 2001).[1] An *admissible* heuristic is one that never overestimates the true

[1] At a first glance, this might seem to contradict our earlier statement that STRIPS planning with only positive effects is NP-complete (Bylander 1994), but Bylander's NP-complete result applies when

cost to the goal. Finding the shortest relaxed plan could be used to provide an admissible heuristic estimate for the remaining number of actions needed to achieve all goals: the shortest relaxed plan must be shorter than the shortest non-relaxed plan because it does not need extra actions that repair the deleterious effects of actions from the non-relaxed problem. Unfortunately, computing the shortest relaxed plan is NP-hard for STRIPS problems (Bonet and Geffner 2001).

2.3 Planning as satisfiability

Although Bylander (1994) shows that STRIPS planning is PSPACE-complete, as Kautz and Selman (1996) observes, the hardness comes from allowing plans to have exponential length; if we restrict our attention to plans of polynomially bounded length, STRIPS planning becomes NP-complete. Hence, we can represent STRIPS planning in other problem classes that are also NP-complete, in particular, the well-known *satisfiability* problem.

2.3.1 The satisfiability problem

The satisfiability problem is a central problem in complexity theory. It was the first problem that was shown to be NP-complete (Cook 1971). It is a problem of considerable research interest and has found applications in software and hardware verification (Prasad et al. 2005), bioinformatics (Lynce and Marques-Silva 2006) and security protocol verification (Armando et al. 2003).

Problem description

A *propositional* or *Boolean* variable v has a domain $\mathsf{Dom}(v) = \mathbb{B} = \{\mathbf{t}, \mathbf{f}\}$ corresponding to the values true and false, respectively. A literal is either a propositional variable v or its negation $\neg v$. A formula is either a literal or is constructed from other formulas by repeated applications of the operations of disjunction ($\vee$), conjunction ($\wedge$) and negation ($\neg$) together with the use of parentheses to indicate application. For a set of propositional variables V, the set of formulas $\mathcal{L}(V)$ over those propositional variables is the minimum set that satisfies the following properties:

- If $v \in V$ then $v \in \mathcal{L}(V)$;

actions can have negative preconditions, whereas Hoffmann and Nebel's result applies when actions can have only positive preconditions.

- If $A \in \mathcal{L}(V)$, then $\neg A \in \mathcal{L}(V)$;
- If $A \in \mathcal{L}(V)$ and $B \in \mathcal{L}(V)$, then $A \wedge B \in \mathcal{L}(V)$ and $A \vee B \in \mathcal{L}(V)$.

A propositional satisfiability problem is a tuple (V,F), where V is a set of propositional variables and $F \in \mathcal{L}(V)$. A truth assignment to the set of variables V is a function $\mathcal{T}\colon V \to \mathbb{B}$ that maps a variable to a Boolean value. If $\mathcal{T}$ is defined for each variable in V then the truth assignment is *complete*, otherwise it is *partial*. An interpretation over propositional variables V with a complete truth assignment $\mathcal{T}$ to the variables V, is a function $[\![\cdot]\!]_{\mathcal{T}}\colon \mathcal{L}(V) \to \mathcal{B}$, which is recursively defined in the following way. For propositional formulas $A, B \in \mathcal{L}(V)$:

- $[\![A \wedge B]\!]_{\mathcal{T}} = \mathbf{t}$ if and only if $[\![A]\!]_{\mathcal{T}} = \mathbf{t}$ and $[\![B]\!]_{\mathcal{T}} = \mathbf{t}$; otherwise, $[\![A \wedge B]\!]_{\mathcal{T}} = \mathbf{f}$.
- $[\![A \vee B]\!]_{\mathcal{T}} = \mathbf{t}$ if and only if either $[\![A]\!]_{\mathcal{T}} = \mathbf{t}$ or $[\![B]\!]_{\mathcal{T}} = \mathbf{t}$; otherwise, $[\![A \vee B]\!]_{\mathcal{T}} = \mathbf{f}$.
- $[\![\neg A]\!]_{\mathcal{T}} = \mathbf{t}$ if and only if $[\![A]\!]_{\mathcal{T}} = \mathbf{f}$; otherwise, $[\![\neg A]\!]_{\mathcal{T}} = \mathbf{f}$.

For a propositional variable $v \in V$, $[\![v]\!]_{\mathcal{T}} = \mathbf{t}$ if and only if $\mathcal{T}(v) = \mathbf{t}$; otherwise, $[\![v]\!]_{\mathcal{T}} = \mathbf{f}$.

A *clause* is a disjunction of literals $(\ell_1 \vee \cdots \vee \ell_k)$. We sometimes use the shorthand $\{\ell_1, \ldots, \ell_k\}$, where we have omitted the disjunction symbols. A formula is in conjunctive normal form (CNF) if it is a conjunction of clauses. Any formula in propositional logic can be expressed in conjunctive normal form (Tseitin 1968).

We can now define the search problem class PROPSAT.

Definition 1. PROPSAT*: Given a propositional satisfiability problem (V, F) where F is expressed in CNF, find a truth assignment $\mathcal{T}$ such that $[\![F]\!]_{\mathcal{T}} = \mathbf{t}$ or deduce that no such truth assignment exists.*

2.3.2 Satisfiability solvers

How might a program be written that solves a problem in PROPSAT? We can divide such algorithms into two classes: *systematic* and *local search*. A systematic solver will explore the entire space of truth assignments and will return the first solution it encounters; if it fails to find a solution after completing its exploration, it can conclude that no solution exits. In contrast, a local search procedure – for example, WalkSat (Selman et al. 1994) – repeatedly samples from the space of truth assignments until it finds a truth assignment that is a solution. Local search procedures often take a cutoff parameter that limits the number of samples it can take before giving up on finding a solution.

We say that a solver is *complete* if it is guaranteed to find a satisfying assignment to a problem when one exists. Systematic solvers by their very nature are complete. In contrast, local search procedures are not complete because there remains the possibility that there is a satisfying truth assignment that was not sampled. One important basis for a systematic solver is the Davis-Putnam-Lodgemann-Loveland method.

Davis-Putnam-Lodgemann-Loveland (DPLL) algorithm

The DPLL procedure (Davis et al. 1962) is the foundation of most state-of-the-art systematic satisfiability solvers. Its basic procedure is summarised in Algorithm 1. The procedure takes a set C of propositional clauses. If C is empty, then the formula is trivially satisfiable. If C contains an empty clause then C is unsatisfiable. Two methods that perform unit propagation and remove pure literals are iterated until no change occurs in the clauses in C.

Unit propagation is implemented in the method $\mathsf{unitPropagation}(C)$, which iterates over each clause of the form $\{\ell\}$ and sets the literal ℓ to true. The method returns a set of clauses $C' \subseteq C$ which contains only the clauses in C that do not contain the literal ℓ or can be derived from a clause in C by removing $\neg\ell$ from the clause.

Pure literal elimination is implemented in the method $\mathsf{removePureLiterals}(C)$. A *pure literal* ℓ is one for which $\neg\ell$ does not appear in any clause in C. All pure literals can be safely set to true. The method $\mathsf{removePureLiterals}(C)$ returns a set of clauses $C' \subseteq C$ such that a clause is in C' if and only if it did not contain a pure literal in C.

When the set of clauses can no longer be simplified by applying unit propagation or pure literal elimination, if there are still unsatisfied clauses, DPLL chooses a variable from the set of currently unassigned variables to assign a value to. The chosen variable is referred to as a *decision* variable. The method $\mathsf{getDecisionVar}(C)$ returns the next decision variable. We do not know whether a decision variable should be set to true or false so we must try both values. Hence, the DPLL method branches on decision variables and backtracks if an empty clause is derived.

Modern SAT solvers

Almost all modern SAT solvers make use of some form of conflict clause learning. These procedures were taken from the *constraint satisfaction problem* field and introduced to DPLL propositional satisfiability solvers (Bayardo and Schrag 1997). DPLL only sets variables either through a branching decision or when a clause becomes unit. When an empty clause is derived, we can identify the subset of the current partial assignment that caused this conflict.

Algorithm 1: DPLL procedure

Result: (b, A) where b is a boolean indicating if the formula is satisfiable and A describes a partial assignment corresponding to the decision variables that can be extended to a satisfiable assignment by unit propagation.

```
1  begin
2      if ∅ = C then return (⊤, ∅)
3      if ∅ ∈ C then return (⊥, ∅)
4      C_before ← ∅
5      repeat
6          C_before ← C
7          C ← removePureLiterals(C)
8          C ← unitPropagation(C)
9      until C = C_before
10     v ← getDecisionVar(C)
11     (isSat,A) ← dpll(C ∪ {v})
12     if isSat then return (⊤, A ∪ {v})
13     else
14         (isSat,A) ← dpll(C ∪ {¬v})
15         if isSat then return (⊤, A ∪ {¬v})
16         else return (⊥, ∅)
17 end
```

Consider a formula that contains the clauses $C_1 = \{a, \neg y\}$, $C_2 = \{a, \neg z\}$ and $C_3 = \{x, y, z\}$ amongst other clauses. If at some point during the search we have the partial assignment $x \mapsto \mathbf{f}$ and $a \mapsto \mathbf{f}$, where the assignments were made in left-to-right order, then C_1 forces the assignment $y \mapsto \mathbf{f}$ and C_2 forces the assignment $z \mapsto \mathbf{f}$ through unit propagation. Hence, clauses C_1 and C_2 are the reason clauses for the assignments to y and z, respectively. However, now C_3 has become empty because all of its variables are assigned false. This is referred to as encountering a conflict and C_3 is the conflict clause.

To be systematic, we backtrack to the most recently assigned variable that can fix the conflict. This is the assignment to z, but the reason clause for z is C_2. By resolving C_2 with C_3 we obtain the clause $\{x, y, a\}$ which indicates that we cannot have a satisfying assignment that assigns false to x, y and z. Now the most recently assigned variable is y, but the reason clause for y is C_1. Hence, by resolving $\{x, y, a\}$ with C_1 we obtain the clause $\{x, a\}$. Now we know that we cannot have a satisfying assignment where both x and a are assigned false. The clause $\{x, a\}$ subsumes $\{x, y, a\}$ since any truth assignment that violates the latter also violates the former. If x and a are both decision variables, that is to say they were added as branching decisions rather than through unit propagation, then we may add the clause $\{x, a\}$ to our formula in order to avoid revisiting this failed branch again during search.

In general, the above process repeats until we identify the most recently assigned decision variable that is part of the reason for the conflict at which point we can change our branching decision – if we do not encounter such a variable, then this indicates that the formula is unsatisfiable. Conflict clause learning retains these reason clauses and includes them in the DPLL procedure as if they were part of the original formula. In practice, retaining all learnt clauses is prohibitively expensive, so a policy that decides when to forget learnt clauses must be used.

Heuristics for choosing decision literals, such as VSIDS, and improvements in implementation techniques for DPLL algorithms, such as the *two watched literal* scheme, help to increase performance further (Moskewicz et al. 2001).

2.3.3 Encoding schemes for planning problems

STRIPS planning as satisfiability was first formulated by Kautz and Selman (1992). This initial formulation constructs a linear encoding of a STRIPS problem and then attempts to find a satisfiable assignment for the generated formula using a local search procedure, GSAT. From a satisfiable assignment, a valid plan can be reconstructed.

Linear encodings

A linear encoding divides time into a sequence of discrete steps, such that at each step exactly one action can be executed. They are called 'linear' because the number of variables in these encodings is linear in the number of time steps. A simple encoding scheme maps each ground action at each time step to a propositional variable. A disadvantage of this approach is that the number of ground instantiations of an action is exponential in the number of parameters of the action. If an action has p parameters, each with domain of size d, then there are at most d^p ground instantiations. If we are producing an encoding that allows m time steps, this contributes md^p propositional variables to the encoding size.

Since the complexity of propositional satisfiability is widely believed to be exponential in the number of variables in the worst case, it makes sense to try to reduce the number of variables in the encoding wherever possible. Since linear encodings admit only one action per time step, a more compact encoding can be achieved using a *split* action representation (Kautz et al. 1996). This encodes a single proposition for each argument of the action. For example, encoding the action $\mathsf{Go}(p, l_1, l_2, t)$ would lead to a proposition $GoArg1Alison_t$ that is true if and only if an action of the form $\mathsf{Go}(Alison, l_1, l_2, t)$ is executed. This would require a total of dp propositions to encode the ground instantiations of an action, which would contribute mdp propositional variables to the encoding size.

Further improvements can be made by overloading the splitting, which allows actions to share arguments. For example, $\mathsf{Action}(Go, t)$ and $\mathsf{Arg1}(Alison, t)$ would denote an action of the form $\mathsf{Go}(Alison, l_1, l_2, t)$. The MEDIC planner explored the space of these type of encodings (Ernst et al. 1997). The experimental results suggested that although these more complicated splitting schemes provide a lower worst-case complexity in the size of the encoding, smaller encodings can be produced using the simpler encoding scheme provided that a pre-processing simplification procedure is used. This simplification procedure uses typing of objects and action parameters, which limits the number of ground instantiations to the extent that a simple one-to-one mapping between ground actions and propositions is more compact than a complicated splitting procedure that allows nonsensical instantiations. Encouragingly, the results also show a positive correlation between smaller encodings and faster running times.

Kautz et al. (1996) also present a compact encoding for a *lifted* SAT solver that uses causal links; however, this is not reduced to a CNF formula in propositional logic, so the success of this approach is reliant on finding a strong first order (without quantification) satisfiability solver.

Parallelised encodings

One problem with linear encodings is that they allow only a single action to be executed at each time step. In some sense, this is wasteful because for each time step, only a very small number of variables can be set to true to encode that a single action is executed. The other remaining variables for that time step must be set to false. Given that the number of variables in a linear encoding is a linear function of the number of time steps, smaller encodings could be produced by allowing more than one action to execute at each time step and thus reducing the number of time steps needed in the encoding to find a plan. Schemes of this type are known as *paralellised* encodings.

Specifying the semantics of when it is possible to execute a set of actions together in a single time step affects the size of the encoding. One possibility is to allow a set of actions to be executed in a single time step if all possible total orderings of those actions are valid. Rintanen et al. (2006) refers to this as a $\forall$-step semantics. Alternatively, relaxing this condition so that a set of actions can be executed together in a single time step if at least one total ordering of the actions is possible is referred to as a $\exists$-step semantics, which admits more parallelism. $\exists$-step semantics are encoded in satisfiability by using a data structure called a disabling graph and the following sufficient condition: a valid total order for a group of actions exists if the subgraph of these actions over the strongly connected components of the disabling graph are acyclic. By introducing auxiliary variables, it is possible to enforce this acyclicity property in the CNF encoding; consequently, this

restricts the satisfiable truth assignments to those that correspond to plans that satisfy the ∃-semantics.

A parallelised encoding has been devised for action splitting that reduces the number of variables in the encoding over other approaches such as SATPLAN's standard encoding and ∃-step semantics (Robinson et al. 2008); however, the encoding often admits less parallelism than the other approaches, which means that longer time steps are required to find solution plans.

Planning graph encodings in SAT

The planning graph allows a certain level of parallelism at each time step: any group of pairwise non-mutex actions can be executed so long as all of their preconditions can be achieved by that time step. The conditions for labelling actions as mutex mean that any total order of the actions in such a group is valid. Thus, the parallelism that is admitted closely matches Rintanen et al.'s ∀-step semantics.

We can use the planning graph to construct compact parallelised encodings of planning problems (Kautz and Selman 1996). The mutex reasoning and reachability analysis involved in constructing the planning graph also helps to reduce the size of the encoding by pruning unreachable facts and unexecutable actions from the encoding. This is used in the BLACKBOX planner (Kautz and Selman 1999). BLACKBOX was developed up to 2003; a more recent implementation of the planning as satisfiability paradigm is SATPLAN (Kautz et al. 2006)[2].

Algorithm 2 summarises the encoding scheme used in SATPLAN06. It constructs a formula φ^T in CNF that encodes a planning graph of makespan T. This formula has the property that if a satisfiable assignment can be found for it, then a valid plan can be efficiently extracted from the assignment. $\mathcal{A}'$ extends $\mathcal{A}$ by incorporating the NOOP actions from the planning graph. $V_{act}(a, t)$ maps an action $a \in \mathcal{A}'$ that occurs in the planning graph at level t to a unique propositional variable. Similarly, $V_{pred}(\rho, t)$ maps a fact $\rho \in \mathcal{F}$ that appears in the planning graph at level t to a unique propositional variable.

Line 8 enforces that if an action is executed at level t then its preconditions are all true at level t. Lines 10 and 12 require that if a fact is true at level t then a supporting action is executed at level $t - 1$. Line 16 prevents two actions that are mutex at level t from being executed together at level t. Line 19 prevents two facts that are mutex at level t from being simultaneously true at level t. Goals reachable at level T are added as unit clauses in line 21 to insist that they are achieved.

[2]We sometimes append a year to the end of SATPLAN's name to indicate the source code we used, for example SATPLAN06 relates to the 2006 version.

Once a satisfiable truth assignment is found, for each action a and time step t such that $V_{pred}(a,t)$ is true in the assignment, the action a is added at that time step to the plan. A post-processing step then removes unnecessary and spurious actions – these are allowed as a result of not enforcing action effects – that are present in the plan.

Algorithm 2: Encoding the planning graph (φ^T).

Result: φ^T encoding the planning problem.

```
1  begin
2      φ ← true
3      foreach ρ ∈ I do
4          φ ← φ ∧ (V_pred(ρ, 0))
5      for t ← 1 to T do
6          foreach a ∈ A' applicable at level t − 1 do
7              foreach ρ ∈ Pre(a) do
8                  φ ← φ ∧ (¬V_act(a, t − 1) ∨ V_pred(ρ, t − 1))
9          foreach ρ ∈ F reachable at level t do
10             C ← ¬V_pred(ρ, t)
11             foreach a ∈ A' applicable at level t − 1 such that ρ ∈ Add(a) do
12                 C ← C ∨ V_act(a, t − 1)
13             φ ← φ ∧ C
14         foreach a_1, a_2 ∈ A' applicable at level t − 1 do
15             if Mutex(a_1,a_2,t − 1) then
16                 φ ← φ ∧ (¬V_act(a_1, t − 1) ∨ ¬V_act(a_2, t − 1))
17         foreach ρ_1, ρ_2 ∈ F reachable at level t do
18             if Mutex(ρ_1,ρ_2,t) then
19                 φ ← φ ∧ (¬V_pred(ρ_1, t) ∨ ¬V_pred(ρ_2, t))
20     foreach g ∈ G reachable at level T do
21         φ ← φ ∧ (V_pred(g, T))
22     φ^T ← φ
23     return φ^T
24 end
```

The Blackbox planning system also uses the failed-literal rule, which is sometimes referred to as *probing*, to simplify its SAT encodings of planning problems (Kautz and Selman 1999). This method attempts to set each literal in turn to true; unit propagation is then iterated to remove unit clauses. If the empty clause is derived at any point then we can conclude that the literal must be set to false if a satisfying assignment is possible. This simplification routine was found to solve some planning problems from the Blocksworld domain entirely without any branching decisions being made.

Londex constraints (Chen et al. 2007) extend the original mutex links from the plan-

ning graph to indicate mutual exclusion across time steps. This method uses the SAS$^+$ representation to determine, for any variable, the minimum number of actions required to change between pairs of values. This allows one to infer that corresponding pairs of propositional facts in a SAT encoding must be separated by the same number of time steps. Hence, we can mark any pair of facts that violates that condition as mutex; for example, facts f_1 and f_2 are marked mutex if $0 \leq t(f_2) - t(f_1) < r$, where t indicates the time step of the fact and r is the minimum number of actions required to make f_2 true from f_1. Furthermore, pairs of actions across time steps can be marked as mutex if there are conflicts between their preconditions and effects that can be detected using either original or long distance mutexes between facts. Adding these constraints to the SATPLAN04 encoding helped to reduce solution times and solve a greater number of problems in their evaluation.

What needs to be in the encoding?

Initial formulations of planning as satisfiability found that solution times could be reduced by adding unnecessary axioms to the encoding that explicitly prohibited impossible conditions (Kautz and Selman 1992). For example, including an axiom to prevent an object being on top of itself. These axioms were unnecessary since they could be deduced from other axioms that were already included in the encoding. The authors claim that this performance increase was only observed when using local search SAT solvers. When a DPLL method was used to solve the encodings, the extra axioms failed to cause an improvement in performance; however, they only present data that shows an improvement in local search and omit data for the DPLL case.

Nevertheless, extra axioms may help to guide a local search procedure towards a solution and to avoid local minima. Since DPLL is complete, it will eventually find a truth assignment if one exists; however, it is somewhat surprising that adding extra axioms, that have the potential to allow shortcuts in unit propagation, does not improve the performance of DPLL. Modern SAT solvers with clause learning have the potential to deduce these type of clauses automatically.

Rintanen (2008) showed that fact mutexes in planning graphs can be derived from a parallel encoding of planning problems when using a unit propagation with look-ahead[3] and a 2-literal clause learning procedure. They also show how londex constraints can be derived using unit propagation and so do not add any extra information or prune the search space; therefore, the performance advantage offered by londex constraints must

[3]Look-ahead is similar to probing. It iterates through unassigned literals, running unit propagation under the condition that the literal is set to true. If unit propagation produces an empty clause then we can conclude that its negation must be true if there is a chance of finding a satisfiable assignment.

come from some other effect, such as changing the evaluation of heuristic functions or offering shortcuts in the inference procedure.

Sideris and Dimopoulos (2010) conduct a systematic review of encoding schemes for planning as satisfiability. For a given londex constraint $(\neg p(t+k) \vee \neg q(t))$, which states that literals corresponding to two different time points $t+k$ and t cannot both be true, they say that the clause is forward redundant if $\neg p(t+k)$ can be derived from unit propagation when $q(t)$ is true, and it is backward redundant if $\neg q(t)$ can be derived by unit propagation when $p(t+k)$ is true. Sideris and Dimopoulos show that londex constraints are forward redundant for the SATPLAN06 encoding. Using this terminology, one realises that Rintanen has only shown that londex constraints are forward redundant for their parallel encoding. Thus, there remains the possibility – although we are not aware of a proof either way – that those encodings are not backward redundant and this may be why londex constraints improve performance for those encodings. Sideris and Dimopoulos present a new encoding for which londex clauses can be shown to be both forward and backward redundant with regard to the underlying encoding, and so londex constraints offer no extra information than what can be derived through unit propagation.

2.4 The maximum satisfiability problem

The Maximum Satisfiability (Max-SAT) problem is the optimisation variant of the traditional SAT problem. The objective is to find a truth assignment that minimises the number of violated constraints or clauses. If we associate with each clause a weight, then the *weighted Max-SAT* problem is to find a truth assignment that minimises the sum of weights of violated clauses. A weighted partial Max-SAT problem divides clauses into two classes: hard and soft. Any truth assignment that is a solution to a weighted partial Max-SAT problem must satisfy all hard clauses, whereas soft clauses can optionally be satisfied.

Formally, a *weighted* clause is a pair (C_i, w_i) where C_i is a propositional clause and $w_i \in \mathbb{N}_0 \cup \{\top\}$ is the weight of clause C_i. $\top$ is a special weight that denotes that the clause is hard. Sometimes it will be convenient for us to write a weighted clause $(\ell_1 \vee \cdots \vee \ell_q, w)$ as

$$\underbrace{(\ell_1 \vee \cdots \vee \ell_q)}_{w}. \tag{2.4}$$

A weighted partial Max-SAT (WPMax-SAT) formula over a set of propositional variables V is a set $F \subseteq \{(C, w) \mid C \in \mathcal{L}(V)$ is a clause and $w \in \mathbb{N}_0 \cup \{\top\}\}$. A truth assignment

$\mathcal{T}$ to the set of variables V satisfies a weighted partial Max-SAT formula F if and only if

$$\left[\!\!\left[\bigwedge_{\substack{(C_i,w_i)\in F \\ \text{s.t. } w_i=\top}} C_i \right]\!\!\right]_{\mathcal{T}} = \mathbf{t}. \tag{2.5}$$

The cost $\mathsf{Cost}(\mathcal{T}, F)$ of a truth assignment $\mathcal{T}$ to the set of variables V that satisfies a WPMax-SAT formula F is equal to

$$\sum_{\substack{(C_i,w_i)\in F \\ \text{s.t. } [\![C_i]\!]_{\mathcal{T}}=\mathbf{f}}} w_i. \tag{2.6}$$

We are now in a position to define the optimisation problem class WPMAXSATOPT.

Definition 2. WPMAXSATOPT*: Given a WPMax-SAT problem (V, F), where F is a WPMax-SAT formula, find a truth assignment $\mathcal{T}$ that satisfies F and for which there is no other truth assignment $\mathcal{T}'$ to the set of variables V such that $\mathsf{Cost}(\mathcal{T}', F) < \mathsf{Cost}(\mathcal{T}, F)$, or deduce that such a truth assignment does not exist.*

Like SAT solvers, Max-SAT solvers can be classified as either systematic or local search. Local search solvers aim to minimise the cost of a solution but cannot guarantee that a solution is optimal. In contrast, systematic solvers will prove that either a solution does not exist or return an optimal solution. Hence, we will be primarily interested in systematic solvers. These often follow a branch-and-bound search combined with a DPLL-style framework. A branch-and-bound algorithm keeps track of an upper bound, which corresponds to the cost of the best solution found so far, and a lower bound on the lowest cost that would be incurred by extending the current partial assignment to a total assignment.

Research in Max-SAT often focuses on the case where all clauses are soft with weight equal to 1. Algorithms specialised for WPMax-SAT can take advantage of knowing that all hard clauses must be satisfied, which allows the use of unit propagation. Also, soft clauses can be promoted to hard clauses when their weight exceeds the difference between the cost of the current partial assignment and the cost of the upper bound.

This dissertation explores representing planning problems within this framework. Using the optimisation features of this framework, it is possible to find provably optimal plans.

2.5 Other compilation approaches to planning

Following the success of compiling classical planning problems to satisfiability problems, similar approaches were attempted to compile to integer programming (Vossen et al. 1999) and constraint satisfaction (Do and Kambhampati 2001a) problems.

2.5.1 Integer programming

The canonical form of a *linear program* (LP) is written as

$$\begin{array}{ll} \text{maximise} & \mathbf{c}^{\mathsf{T}}\mathbf{x} \\ \text{such that} & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \forall i.x_i \geq 0. \end{array} \tag{2.7}$$

If the values of the variables x_i are constrained to be integers, then it is known as an *integer program* (IP). If only a subset of the variables x_i are constrained to be integers then it is known as a *mixed integer program* (MIP). Vossen et al. (1999) presented an encoding scheme for representing STRIPS planning problems as integer programs. Separate variables are created for each fact and action present at each time step in the planning graph. The encoding is similar to that used for satisfiability, for example, the mutex $(\neg x_1 \vee \neg x_2)$ can be written as the constraint $x_1 + x_2 \leq 1$.

One of the main advantages of using an integer programming representation is the ease with which numerical variables and constraints can be represented in the target language. Expressing the need for a minimal number of actions can be done by setting $c_i = 1$ for all i such that x_i encodes an action, and setting $c_i = 0$ everywhere else. Fixed-horizon cost-optimal planning can be performed by setting c_i to the cost of executing the corresponding action.

Vossen et al.'s empirical results suggest that BlackBox is quicker at solving SAT encodings of planning compared to applying a MIP solver to IP encodings of planning. Optiplan (van den Briel and Kambhampati 2005) improves upon Vossen et al.'s state-change encoding by only encoding facts and actions that are present in the planning graph. This reduces the number of variables and constraints in the encodings, leading to faster solution times. van den Briel et al. (2008) present a 'loosely-coupled' formulation of planning that improves upon Optiplan. Their state-change encodings are constructed from a multi-valued SAS$^+$ representation of the problem. It is likely that this translation step leads to more compact representations that are responsible, in part, for the improved performance over Optiplan. van den Briel et al.'s generalised one state change encoding adopts a $\exists$-step semantics (Rintanen et al. 2006), which further reduces the size of the

encodings required to solve a problem by admitting an increased level of parallelism, as was discussed in Section 2.3.3.

2.5.2 Constraint satisfaction

Planning graphs have also been compiled to constraint satisfaction problems (CSPs) (Do and Kambhampati 2001a). In this encoding the variables in the CSP correspond to propositions in the planning graph; the domain of a variable is its set of supporting actions at that level in the planning graph. The final results presented by Do and Kambhampati show that their solver, GP-CSP, is faster than Blackbox. The use of explanation based learning in the constraint solver is a large contributor to this good performance. The Relsat solver, which is one of the underlying SAT solvers used by Blackbox, also incorporates explanation based learning; however, there is not much evidence to suggest that its policy for retaining explanations for failure has been as optimised as that used in GP-CSP. Moreover, GP-CSP also incorporates variable and value ordering heuristics that use information from the planning graph.

The conclusion presented by Do and Kambhampati is that GP-CSP is faster than Blackbox using either Satz or Relsat. While their results confirm this, it is worth noting that they optimise the explanation based learning whereas this was not done for Relsat on planning problems.

2.6 Heuristic search approaches to planning

Planning as heuristic search has proven to be one of the most popular and successful approaches to STRIPS planning over the last decade. Hsp (Bonet and Geffner 2001) uses a planning graph as the basis for its heuristic computation, which defined the approach for many subsequent systems. Bonet and Geffner present two variations of heuristic search: Hsp and Hspr which correspond to progression (starting from the initial state and progressing towards a goal) and regression searches (starting from the goal propositions and regressing back towards the initial state), respectively.

Hsp's heuristic computation at a state s constructs a relaxed planning graph from s to a level at which all the goal propositions occur. Estimates of the cost of achieving each proposition are then computed using a dynamic programming approach over the relaxed planning graph. The results are used to estimate the cost of achieving the group of unachieved goal propositions. Since this requires computing a planning graph for every state visited, Hspr was proposed. This constructs a relaxed planning graph from the initial state and uses a similar cost computation used to estimate the number of actions

needed to move from the initial state to one that achieves a group of propositions. In this way, a regression planner is able to compute cost estimates once and reuse them many times during search.

The cost estimates of achieving a group of propositions C make assumptions about how independent the propositions in C are. Since one action can achieve and delete many propositions, a sequence of actions that achieves a proposition in C can, in addition, achieve other propositions in C; similarly, it can also delete propositions that are achieved by a parallel sequence of actions supporting another member of C. This is termed positive and negative interaction, respectively. Estimating the cost of a collection of propositions as the sum of their individual costs neglects both positive and negative interaction which, in the former case, can overestimate the true cost. Taking the estimate as that of the two hardest propositions to co-achieve will never overestimate the true cost but can be less informative because it neglects the cost of achieving the other propositions.

The AltAlt planner (Nguyen et al. 2002), which conducts a regression state-space search very similar to HSPR, attempts to correct estimates by considering the difference between the first level at which two propositions appear non-mutex in a non-relaxed planning graph and the level at which both propositions first appear in the relaxed planning graph. This estimates the cost of fixing interference between supporting actions. For a group of propositions C, the two propositions in C that have the largest fixing estimate are identified, and this fixing estimate is added to the length of the relaxed planning graph that contains all propositions C in its final layer, to produce an improved heuristic.

The FF planning system (Hoffmann and Nebel 2001) builds upon the main idea of HSP by conducting an enforced hill-climbing progression search with a relaxed planning graph heuristic. The heuristic is calculated by finding a relaxed plan, from the current state, that supports all goal propositions and then summing the number of actions in this relaxed plan to give the value of the heuristic. This takes into account positive interactions between actions in a way that HSP does not. The term 'enforced' refers to the part of the algorithm that at each state s, performs a breadth-first search until a state s' with $h(s') < h(s)$ is found; hence, we are always selecting a next state which estimates a better distance to the goal. If no such state can be found, the search terminates and reverts to the complete heuristic search algorithm.

The idea of using relaxed planning graphs in heuristic computation has been extended to handle planning with durative actions and metric resource constraints in the SAPA planner (Do and Kambhampati 2001b). A relaxed temporal planning graph is used to extract heuristics. Since the planner is metric, a new kind of relaxation can be performed that ignores effects that reduce the quantity of a resource. Since the planner is temporal, a new kind of heuristic can be computed that is based upon the sum of durations of actions in a relaxed plan. Observing that actions often consume resources rather than produce

them, heuristics can be improved by calculating, for each resource, the excess amount required by a relaxed plan and by identifying the maximal increase in each resource that is possible by executing a single action. Together, this can improve the heuristic estimate by calculating the minimum number of action executions needed to produce the resources required for a relaxed plan.

The *fast downward* planning system (Helmert 2006) uses a causal graph to compute heuristic estimates. This is constructed from a domain transition graph that uses a SAS^+ representation of the planning problem. The planning system combines multiple heuristics by maintaining an open-list for each heuristic. State expansions are then made from each open-list in a round-robin fashion, with expanded states being added to all open-lists.

The LAMA planning system, winner of the 2008 IPC sequential satisficing track, is built upon the *fast downward* planner (Richter and Westphal 2008). Its main contribution is to include a landmark counting heuristic. Landmarks, which are described in more detail in our future work (Chapter 6), are facts that must be true at some point in any valid plan. The landmark counting heuristic estimates the cost to a goal state my considering the number of landmarks that remain to be achieved. This is combined with other heuristics according to the fast downward planner's algorithm.

Rintanen (2010) has investigated the effect of incorporating a planning specific heuristic for selecting decision variables inside a SAT solver to be used on SAT encodings of planning problems. The heuristic is quite simple: choose an action that achieves an open (sub)goal at the earliest time possible. The use of this heuristic leads to an improvement in finding satisfiable assignments over the popular VSIDS heuristic and is competitive with the LAMA planning system, one of the most successful state-space heuristic search planners.

2.7 Applications

Before concluding this chapter, we briefly review some past and emerging applications of automated planning systems that motivate research in planning.

Mars Exploration Rover

NASA's Mars Exploration Rover is an example of an autonomous agent operating in an environment where communication with the agent is difficult. Each rover is only able to communicate with orbiting spacecraft for a limited time each day as an orbiter passes over the rover. The solution is to give each rover a plan of work to do for the period of time that it is unable to communicate with Earth. Advances in technology means that it is now possible for rovers to cover large distances while they are incommunicado. While a rover

is travelling it may encounter many objects of scientific interest. Given that scientists are unable to receive a live video feed, they cannot interrupt the rover to ask it to explore new areas. The AEGIS (Autonomous Exploration for Gathering Increased Science) system is used to identify targets of scientific interest from a rover's panoramic camera (Estlin et al. 2009). Once such objects are identified, they are passed onto a replanning system as additional goals. The replanning system must then calculate whether it is feasible to take scientific measurements in the new area given its other goals and resource constraints. If so, it must adjust the current plan to incorporate those experiments.

Contract bridge

Contract bridge is a game of incomplete information, which gives rise to a large branching factor. This limits the effectiveness of traditional adversarial search methods. This branching factor can be significantly reduced by observing that bridge players often engage in one of only a small number of standard tactical plays: for example, finessing, ruffing and crossruffing.

Bridge baron achieved success as a computer program for playing bridge by incorporating *hierarchical task networks* from planning research to automate play (Smith et al. 1998). Hierarchical task networks use *methods* that decompose tasks into sub-tasks. Eventually, by repeatedly applying methods, a plan consisting of entirely atomic tasks is derived, which can then be executed as a plan. In bridge, methods encapsulate particular tactical plays and describe the order of steps needed to complete the play. Using this approach, only successful strategies are considered, which allows search to progress to the leaf nodes and to propagate scores upwards through the tree to decide which move to make.

Semantic web

The *semantic web* is an umbrella term used to describe the objective of making the information on the Internet understandable by machines. Part of this goal is to provide a semantic annotation of Web services so that the discovery, execution and composition of Web services can be automated (McIlraith et al. 2001). The latter goal is of particular interest as a practical application of planning methods.

The hope is that eventually, users will be able to describe their need, and from this, a planning system can combine a collection of Web services, that have been identified as providing particular functions, such that the user's need is met. A example of where this might be useful is for the task of planning a holiday. There are various options for travel between locations, of varying costs and durations. A user is likely to have preferences over the time, cost and duration of travel. While on holiday he would like to

participate in a range of activities and he would like his time to be planned so that he can participate in as many activities as possible while minimising the cost and slack time between activities. An automated holiday planner could identify a possible itinerary for the user that attempts to closely match his preferences, thus allowing the user to refine the plan and decide whether to commit to it.

Recent research has attempted to combine this with planning techniques in pursuit of making such methods a reality. Examples include:

- a system that translates OWL-S – a language for describing the semantics of Web services – to PDDL before solving the planning problem using a FF style planner (Klusch et al. 2005).
- formulating the 'composition goal' and description of existing Web services as a planning problem, and then using a solution plan to produce a program in a low-level process language that describes how to execute existing Web services to achieve the composition goal (Traverso and Pistore 2004).

2.8 Summary

In this chapter we have introduced the STRIPS model of classical planning and described how planning problems represented in this language can be compiled to satisfiability problems. We have reviewed various aspects of the choice of encoding that is used to represent such problems. We have highlighted competing approaches to classical planning, such as compilation to other target languages (IP, CSP), and heuristic search methods. We introduced the optimisation problem called maximum satisfiability that will be used throughout the rest of this dissertation to direct plan search towards higher quality solutions. In the next chapter, we explore how to represent preferences in planning within this framework.

Chapter 3

Goal utility dependencies

In this chapter we describe the concept of goal utility dependencies in more detail. We describe how such preferences can be included in the net benefit quality metric. We present an encoding scheme in WPMax-SAT that can be used to solve this problems for a limited planning horizon.

3.1 Introduction

In a classical planning problem, valid plans must achieve every goal; however, when faced with limited resources, it is often useful to relax this constraint and allow the planner to make a trade-off between the benefit and cost of achieving a set of goals. Such planning problems are often called *partial satisfaction* or *oversubscription* problems. There is a growing body of research concerning how to utilize concepts of utility and preference from decision theory in the solution of such problems. This has produced a renewed interest in modeling problems as integer programs due to their facility for explicitly expressing an optimization function over solutions. Despite integer programming (IP) lagging behind satisfiability in performance (van den Briel et al. 2008), the possibility of a satisfiability approach to partial satisfaction problems in classical planning has largely been neglected.

In previous years, satisfiability approaches to planning have enjoyed much success: in the 4th and 5th International Planning Competition (IPC) Satplan (Kautz and Selman 1999; Kautz et al. 2006) achieved first and joint first prizes respectively for optimal planning in propositional domains. In this context, an optimal plan is one with the smallest number of discrete time steps, often referred to as the plan's *makespan*. This notion of optimality is not a particularly natural one; instead IPC-6 adopts a collection of different metrics for optimization: number of actions, total action cost and net benefit, where the net benefit of a plan is the total utility of the goals achieved minus the cost of the exe-

cuted actions. This raises the question of how the total utility of a set of goals should be defined.

There are scenarios where the utility of achieving a single goal depends upon which other goals are co-achieved. One such example arises in the area of autonomous scientific agents, which are tasked with performing a collection of experiments. For these systems, it seems desirable to achieve complementary goals, which give good scientific coverage. For example, the utility of taking an image and a rock sample from a single experiment site should be greater than the sum of the utilities of achieving either of those tasks in isolation.

Groups of goals do not always interact positively, and there are scenarios where jointly achieving two or more goals has a reduced utility over achieving any one of them in isolation. For a purchasing problem, the agent may be expected to choose a single item from a collection of similar products. Purchasing more than one item may provide little extra benefit and the agent should not incur the extra cost of doing so in such instances.

Do et al. (2007) described these types of problems as having *goal utility dependencies*, and they presented a systematic approach for handling them using the *Generalized Additive Independence* (GAI) model of utility and integer programming. Their IPUD planner finds solutions with maximum net benefit for a bounded makespan horizon, but plans may exist with larger makespans that have a greater net benefit.

Thus, it seems that integer programming has received renewed interest for this type of planning because of its facility for explicitly expressing an optimization function. There has been some work on handling preferences in SAT: SATPLAN has been extended to minimize the cost of executed actions in planning within a bounded horizon (Giunchiglia and Maratea 2007). In principle, it should be possible to extend this to solve PSP net benefit problems. However, maximum satisfiability (Max-SAT), in particular its weighted variants, also offers a facility for explicitly defining an optimization function in a similar manner to integer programming.

To our knowledge, we present the first system that handles planning problems with preferences using a general-purpose WPMax-SAT solver. It is also the first to explore a satisfiability approach to handling goal utility preferences. This chapter will present an encoding scheme for representing PSP net benefit problems with goal utility dependencies in WPMax-SAT together with a technical extension, which we call MSATPLAN, to SATPLAN that implements this using a general-purpose WPMax-SAT solver. The empirical evaluation of this system is presented in Chapter 5, where we find that MSATPLAN has competitive and often better performance than an IP counterpart.

3.2 Partial satisfaction planning

Each user has their own set of preferences, and a planning system should take a description of this set of preferences in order to return a plan that closely matches them. Utility provides one method of describing a user's set of preferences. Utility is a measure used to assess the degree of satisfaction that a user has with a particular outcome. For a particular user, this measure is described by a utility function, which maps each outcome to a number. This determines an ordering over outcomes such that those outcomes with higher utility are more preferable to the user.

In partial satisfaction planning (PSP), the *outcome* of a plan can be regarded as the subset of goals that are achieved. The PSP net benefit metric (Van Den Briel et al. 2004) assumes that each goal has an individual utility, and that the total utility of a set of goals achieved by a plan can be expressed simply as the sum of their individual utilities; however, it also assumes that each action has a cost that is incurred when executed. The utility and cost measures are assumed to have the same unit of measurement. This determines an ordering over valid plans: those with higher net benefit are more preferable to the user. The PSP model is worth studying because it is conceptually simple to state but can express a variety of interesting problems: the class of Simple Preferences from PDDL3 can be reduced to PSP (Benton et al. 2009).

Formally, a PSP planning problem is the extension $\langle \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A}, u, c \rangle$, where $\mathcal{F}$, $\mathcal{I}$, $\mathcal{G}$ and $\mathcal{A}$ are defined in the same way as for a classical planning problem $\langle \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ (see Section 2.1.1); the function $u \colon \mathcal{G} \to \mathbb{N}_0$ gives each goal $g \in \mathcal{G}$ a utility $u(g)$; and the function $c \colon \mathcal{A} \to \mathbb{N}_0$ gives each action $a \in \mathcal{A}$ a cost $c(a)$. A plan P of length m is an ordered sequence of actions $(a_1, \ldots, a_m)$ that achieves the set of goals $G = \mathit{Result}((a_1, \ldots, a_m), \mathcal{I}) \cap \mathcal{G}$. For such a problem, the net benefit of P is equal to

$$\sum_{g \in G} u(g) - \sum_{i=1}^{m} c(a_i). \tag{3.1}$$

An optimal solution to a PSP planning problem is a plan that has maximal net benefit over all other plans. Note that this definition of PSP net benefit assumes all goals to be soft. Hard goals can be incorporated by restricting valid plans to those P such that all hard goals are members of $\mathit{Result}(P, \mathcal{I})$.

One shortcoming of the PSP net benefit framework is that it is difficult to model situations where goals act as substitutes or complements for other goals. Consider an automated holiday planning system. The holiday destination might consist of many museums that are of interest to the user, but the user's holiday only lasts a small number of days and they wish to undertake other activities, such as visiting a spa, and do not want to spend

all their time and money visiting museums. If the utilities of each activity, and the costs of achieving them, are of comparable magnitudes, then net benefit planners may exhibit indifference between plans that visit only museums and ones that visit only spas. This is because the planner is indifferent between achieving goals that add the same net benefit to the plan, that is, their utility minus the cost of achieving them is identical. The planner is unable to consider that one goal might be more preferable because it provides the user with something different to that which has been currently achieved. For example, that the holiday plan so far consists of only visits to museums, so something other than an extra museum visit should be achieved.

One possible way around this problem is to enforce rules such as 'at least one spa visit should be made' or 'no more than two museum visits should be made'. This could be accomplished by introducing extra pseudo actions and auxiliary goals. However, this approach adds to the number of variables in the encoding, which increases the search space.

There are also scenarios where the utility of achieving a single goal depends upon which other goals are co-achieved. Consider an example based on the Mars Exploration Rover. A Rover must gather experimental data at a number of locations. There are different types of experiments to perform, such as taking images or rock samples, which each measure a particular aspect of the location. In such a scenario, it may be desirable to achieve complementary goals that give good scientific coverage. For example, the utility of taking an image and a rock sample from a single site should be greater than the sum of the utilities of achieving either of those tasks in isolation.

3.3 Goal utility dependencies

The underlying reason for the shortcomings described above is that the PSP net benefit definition has assumed something about the user's utility function: that the utility of achieving a group of goals is equal to the sum of the utilities of achieving each goal in isolation. When this is not true, we need to be able to model the user's utility function over the set of achieved goals.

To make this more concrete, let us start by defining some fundamentals. We work with a classical planning problem $\langle \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$. We will also use the set of binary values $\mathcal{B} = \{0, 1\}$ and the set of natural numbers $\mathbb{N}_0 = \{0, 1, 2, \dots\}$. For a nonempty set $G \subseteq \mathcal{G}$ of N propositional variables with indexed elements $\{g_1, \dots, g_N\}$ and a subset $S \subseteq G$ of goals

achieved by a particular plan, we define the characteristic vector $\mathbf{1}_S^G = (x_1, \ldots, x_N)$ where

$$x_i = \begin{cases} 1 & \text{if } g_i \in S \\ 0 & \text{otherwise.} \end{cases}$$

Thus, we can use $\mathbf{1}_S^G$ to describe which goals have been achieved from a particular subset of goals.

For two different outcomes of a plan $o_1, o_2 \in \mathcal{B}^{|\mathcal{G}|}$, we could represent our preference between the outcomes qualitatively using a partial order $\succeq$ by writing $o_1 \succeq o_2$ when o_1 is more preferred than o_2. Since there are $2^{|\mathcal{G}|}$ different outcomes, a naive representation would require the user to specify $O(2^{2|\mathcal{G}|})$ pairs in order to describe fully the user's preferences over choices between outcomes. While this number might be manageable for a computer to work with, since the number of goals will be comparatively small, it is unreasonable to expect users to specify this information even for a modest number of goals.

In practice, we can describe our preferences in a much more compact form because we do not often reason using total orders over all possible combinations. For instance, if we say that we prefer to go swimming rather than hiking during a holiday, we rarely mean that we prefer any holiday itinerary where swimming is included over any where hiking is included. Under this semantics, we would prefer a holiday in which we went swimming and did nothing else over a holiday where we were paid one million pounds to take part in a hiking trip. Instead, we often mean that we prefer to go swimming over hiking if all other parameters of the itinerary remain the same. This is referred to as *preferential independence* between attributes of outcomes. It is analogous to the idea of independence in probability, and by exploiting it where available, we can obtain more compact representations of utility functions. The following standard definitions describe preferential independence more precisely, see Keeney and Raiffa (1993) for a thorough treatment of the subject.

Preferential independence

If we partition the set of goals $\mathcal{G}$ into two sets $X, Y \subseteq \mathcal{G}$ such that $Y = \mathcal{G} \setminus X$. For any $\mathbf{x} \in \mathcal{B}^{|X|}$ and $\mathbf{y} \in \mathcal{B}^{|Y|}$, we use the shorthand of writing $\mathbf{xy}$ to denote the vector $\mathbf{1}_S^{\mathcal{G}}$ where $S = S_x \cup S_y$; S_x is the smallest set that satisfies $\mathbf{1}_{S_x}^X = \mathbf{x}$; and S_y is the smallest set that satisfies $\mathbf{1}_{S_y}^Y = \mathbf{y}$.

A subset of goals X is *preferentially independent* of its complement $Y = \mathcal{G} \setminus X$ if and only

if for all $\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{B}^{|X|}$ and all $\mathbf{y}_1, \mathbf{y}_2 \in \mathcal{B}^{|Y|}$ the following is true,

$$\mathbf{x}_1\mathbf{y}_1 \succeq \mathbf{x}_2\mathbf{y}_1 \text{ iff } \mathbf{x}_1\mathbf{y}_2 \succeq \mathbf{x}_2\mathbf{y}_2. \tag{3.2}$$

In other words, for a subset of goals X, we always prefer an outcome that achieves those goals in X that are indicated by $\mathbf{x}_1$ rather than those goals in X that are indicated by $\mathbf{x}_2$ if the set of goals that are achieved but are not in X is held to be the same in both instances. In this case, we say that assignment $\mathbf{x}_1$ is preferred to $\mathbf{x}_2$ *ceteris paribus* ('all other things being equal').

Conditional preferential independence

Let the set of goals be partitioned into three disjoint sets X, Y, Z such that $X \cup Y \cup Z = \mathcal{G}$. Similarly, for $\mathbf{x} \in \mathcal{B}^{|X|}$, $\mathbf{y} \in \mathcal{B}^{|Y|}$ and $\mathbf{z} \in \mathcal{B}^{|Z|}$, we write $\mathbf{xyz}$ to denote the vector $\mathbf{1}^{\mathcal{G}}_{S}$ where $S = S_x \cup S_y \cup S_z$; S_x is the smallest set that satisfies $\mathbf{1}^{X}_{S_x} = \mathbf{x}$; S_y is the smallest set that satisfies $\mathbf{1}^{\mathcal{G}}_{S_y} = \mathbf{y}$; and S_z is the smallest set that satisfies $\mathbf{1}^{\mathcal{G}}_{S_z} = \mathbf{z}$.

X is preferentially independent of Y given the assignment $\mathbf{z} \in \mathcal{B}^{|Z|}$ if for all assignments $\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{B}^{|X|}$ and assignments $\mathbf{y}_1, \mathbf{y}_2 \in \mathcal{B}^{|Y|}$, the following is true,

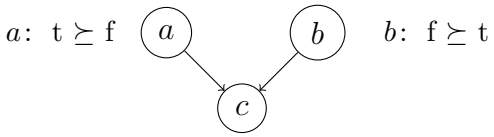
$$\mathbf{x}_1\mathbf{y}_1\mathbf{z} \succeq \mathbf{x}_2\mathbf{y}_1\mathbf{z} \text{ iff } \mathbf{x}_1\mathbf{y}_2\mathbf{z} \succeq \mathbf{x}_2\mathbf{y}_2\mathbf{z}. \tag{3.3}$$

If the above condition holds for all $\mathbf{z} \in \mathcal{B}^{|Z|}$, then we say that X is *conditionally preferentially independent* of Y given Z.

CP-nets

A conditional preference network, or CP-net, (Boutilier et al. 1999) is a graphical model used to represent a user's preferences over the values that a set of variables can take. A CP-net consists of a set of nodes which represent variables – for our purposes, $\mathcal{G}$ is the set of binary variables, and $g \in \mathcal{G}$ takes value 1 if the corresponding goal is part of the outcome and 0 otherwise – which are joined by directed edges to form a directed acyclic graph. The parents of variable X are contained in the set of variables $Parents(X)$ for which each element has an outgoing edge joined to X in the CP-net. Each variable node in a CP-net is annotated with a conditional preference table (CPT), which states for each assignment to its parents, the ordering over its values. For a CP-net to be valid, every variable X must be conditionally preferentially independent of $\mathcal{G} \setminus (X \cup Parents(X))$ given $Parents(X)$.

An example CP-net is shown in Figure 3.1 and the orderings between assignments that



a	b	c
t	t	t ≻ f
t	f	t ≻ f
f	t	f ≻ t
f	f	t ≻ f

Figure 3.1: An example CP-net.

it implies are shown in Figure 3.2. We can see that $a\bar{b}c$[1] is the most preferred assignment and $\bar{a}bc$ is the least preferred assignment according to the CP-net. However, the CP-net does not make any assertion about the preference between certain pairs of assignments, for example, $ab\bar{c}$ and $\bar{a}\bar{b}\bar{c}$. When it is not possible to achieve particular outcomes due to the set of actions available to the planning system, the CP-net representation can be problematic for describing preferences between outcomes of plans.

For example, consider a planning problem with three goals a, b, c where the preferences between outcomes is described by the above CP-net. Imagine that this planning problem only contains three different actions with add lists $\{a\}$, $\{a, b\}$ and $\{b, c\}$. Moreover, b does not appear in the delete list of any action; that is to say, the outcome $a\bar{b}c$ is not achievable by a valid plan. The CP-net does not make any assertion about which of the two possible outcomes abc and $a\bar{b}\bar{c}$ is more preferable. Hence, a planning system using this representation would be unable to decide between two plans with these respective outcomes. This problem can be overcome by using a quantitative ranking of outcomes.

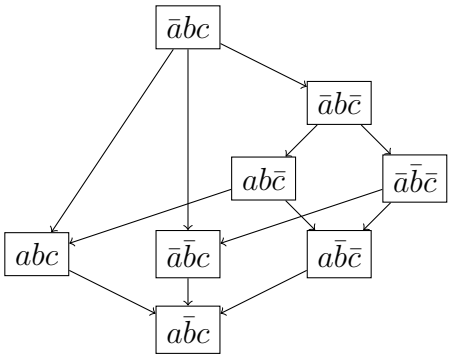


Figure 3.2: Orderings implied by the example CP-net.

[1] $\bar{x}$ indicates that variable x is assigned the value *False*.

3.3.1 Generalized additive independence

A utility function $u\colon \mathcal{B}^{|\mathcal{G}|} \to \mathbb{N}_0$ induces a preference order $\succeq$ over outcomes $o_1, o_2 \in \mathcal{B}^{|\mathcal{G}|}$ such that $o_1 \succeq o_2$ iff $u(o_1) \geq u(o_2)$. Having such a utility function would make it easy to select the most preferred outcome from any pair of options: apply u separately to both options and choose the outcome that leads to the higher value of u.

We assume that we are given this utility function by the user. In the worst case, to describe u will have a space requirement that is exponential in the number of goals; however, we can achieve a more compact representation if we assume that u can be factorized in some way. The *Generalized Additive Independence* (GAI) model (Bacchus and Grove 1995) provides one such method for decomposing a utility function and is used by Do et al. (2007) in their framework for handling goal utility dependencies. We still want to make use of concepts such as preferential independence to make it easy for a user to specify preferences in an intuitive manner.

Let us split $\mathcal{G}$ into k, not necessarily disjoint, nonempty subsets $G_1, \ldots, G_k$ such that $\bigcup_{i=1,\ldots,k} G_i = \mathcal{G}$. The utility function u has an *additive decomposition* over $G_1, \ldots, G_k$ if u can be expressed as

$$u(\mathbf{1}_S^{\mathcal{G}}) = \sum_{i=1}^{k} f_i(\mathbf{1}_{G_i \cap S}^{G_i}) \tag{3.4}$$

for a collection of k functions $f_i\colon \mathcal{B}^{|G_i|} \to \mathbb{N}_0$. Hopefully, k and the size of each G_i will be sufficiently small to allow us to represent the function using less space than a single tabular representation of u over $|\mathcal{G}|$, thus leading to a more practical encoding.

UCP-nets

UCP-nets are an extension to CP-nets that use an additive decomposition of a utility function (Boutilier et al. 2001). A UCP-net for the utility function u exists if a directed acyclic graph can be constructed such that u has an additive decomposition

$$u(g_1, \ldots, g_n) = \sum_{i=1}^{n} f_i(g_i, \mathit{Parents}(g_i)), \tag{3.5}$$

and the preference order $\succeq$ induced by u is such that each variable X, in the UCP-net, is conditionally preferentially independent of $\mathcal{G} \setminus (X \cup \mathit{Parents}(X))$ given $\mathit{Parents}(X)$. Not all utility functions with additive decompositions have a UCP-net representation due to the acyclicity and conditional preferential independence restrictions; however, all UCP-nets must have an additive decomposition. Consequently, we will assume that u is specified using an additive decomposition as described above. This will allow us to handle user preferences that are specified by UCP-nets.

Net benefit

The utility of a set of goals achieved by a plan is only one aspect of its quality; there are often many different plans that will achieve a particular set of goals. In order to distinguish between them, we should include their execution cost in our measure of plan quality. For each action $a \in \mathcal{A}$ we associate a cost $c(a) \in \mathbb{N}_0$ for executing that action. By writing the set of goals achieved by a plan P as $\mathsf{Goals}(P) = \mathit{Result}(P, \mathcal{I}) \cap \mathcal{G}$, we can now precisely define the most preferred plan $P^\star$ as the plan that maximizes the *net benefit*:

$$P^\star = \underset{\substack{\text{valid plans} \\ P=(a_1,\ldots,a_m)}}{\operatorname{argmax}} \left(u\left(\mathbf{1}^{\mathcal{G}}_{\mathsf{Goals}(P)}\right) - \sum_{i=1}^{m} c(a_i) \right). \tag{3.6}$$

3.3.2 Plan Encoding

We will encode a planning problem as a WPMax-SAT formula (see Section 2.4). We extend the 'thin-gp' encoding from SATPLAN (Kautz et al. 2006) to create a formula φ_h^T in propositional logic with the property that a satisfying assignment to it allows us to extract a plan with a makespan of at most T from the truth assignment; however, since all goals are soft, we do not include the clauses that force each goal to be true at level T, and we add additional axioms, not included in the original 'thin-gp' encoding, that ensure a goal is not achieved by the extracted plan if it is false at level T.

A summary of the steps involved in producing this encoding can be found in Algorithm 3. Lines 3–19 are from the original 'thin-gp' encoding; we write $\mathcal{A}'$ to denote $\mathcal{A}$ extended to include NOOP actions for each fact. For more details the reader should refer to Kautz et al. (2006).

A planning graph (Blum and Furst 1997) of makespan T is built from domain and problem PDDL files. For $0 \leq t \leq T$, a binary variable $V_{\mathit{pred}}(\rho, t)$ is created for each fact ρ that is reachable at time step t. For $0 \leq t \leq T-1$, a binary variable $V_{\mathit{act}}(a, t)$ is created for each action a that has reachable and non-mutex preconditions at level t.

A satisfying truth assignment, $\mathcal{S}$, to the variables in φ_h^T corresponds to a valid plan $\mathsf{Plan}(\mathcal{S})$; however, it is no longer necessary for a valid plan to achieve all the goals in $\mathcal{G}$. For each goal fact $\rho_i \in \mathcal{G}$, if ρ_i is in the planning graph at level T then the variable $V_{\mathit{pred}}(\rho_i, T)$ occurs in φ_h^T and we say that ρ_i *is coded for as a goal in* φ_h^T. If ρ_i has been coded for as a goal then ρ_i is achieved by executing $\mathsf{Plan}(\mathcal{S})$ from the initial state iff $\mathcal{S}[V_{\mathit{pred}}(\rho_i, T)]$ is true; in all other cases, including those for which ρ_i has not been coded for as a goal, $\mathsf{Plan}(\mathcal{S})$ does not achieve ρ_i. The 'only if' condition is ensured by the

addition of the following axiom to the encoding:

$$\neg V_{pred}(g,t) \Rightarrow \left(\neg V_{pred}(g,t-1) \vee \bigvee_{\substack{a \in \mathcal{A} \\ \text{s.t. } g \in Del(a)}} V_{act}(a,t-1) \right) \wedge \bigwedge_{\substack{a \in \mathcal{A} \\ \text{s.t. } g \in Add(a)}} \neg V_{act}(a,t-1), \ (\forall g \in \mathcal{G}, 1 \leq t \leq T), \tag{3.7}$$

which produces the clauses as described in lines 20–27 in Algorithm 3.

Algorithm 3: Encoding the planning graph (φ_h^T).

Result: φ_h^T encoding the planning problem.

begin
  $\varphi \leftarrow$ **true**
  foreach $\rho \in \mathcal{I}$ **do**
    $\varphi \leftarrow \varphi \wedge (V_{pred}(\rho, 0))$
  for $t \leftarrow 1$ **to** T **do**
    foreach $a \in \mathcal{A}'$ *applicable at level* t **do**
      foreach $\rho \in Pre(a)$ **do**
        $\varphi \leftarrow \varphi \wedge (\neg V_{act}(a, t-1) \vee V_{pred}(\rho, t-1))$
    foreach $\rho \in \mathcal{F}$ *reachable at level* t **do**
      $C \leftarrow \neg V_{pred}(\rho, t)$
      foreach $a \in \mathcal{A}'$ *applicable at level* $t-1$ *such that* $\rho \in Add(a)$ **do**
        $C \leftarrow C \vee V_{act}(a, t-1)$
      $\varphi \leftarrow \varphi \wedge C$
    foreach $a_1, a_2 \in \mathcal{A}'$ *applicable at level* $t-1$ **do**
      if `Mutex`$(a_1, a_2, t-1)$ **then**
        $\varphi \leftarrow \varphi \wedge (\neg V_{act}(a_1, t-1) \vee \neg V_{act}(a_2, t-1))$
    foreach $\rho_1, \rho_2 \in \mathcal{F}$ *reachable at level* t **do**
      if `Mutex`(ρ_1, ρ_2, t) **then**
        $\varphi \leftarrow \varphi \wedge (\neg V_{pred}(\rho_1, t) \vee \neg V_{pred}(\rho_2, t))$
    foreach $g \in \mathcal{G}$ *reachable at level* t **do**
      foreach $a \in \mathcal{A}'$ *applicable at level* $t-1$ *such that* $g \in Add(a)$ **do**
        $\varphi \leftarrow \varphi \wedge (V_{pred}(g, t) \vee \neg V_{act}(a, t-1))$
      if g *is reachable at level* $t-1$ **then**
        $C \leftarrow V_{pred}(g, t) \vee \neg V_{pred}(g, t-1)$
        foreach $a \in \mathcal{A}'$ *applicable at level* $t-1$ *such that* $g \in Del(a)$ **do**
          $C \leftarrow C \vee V_{act}(a, t-1)$
        $\varphi \leftarrow \varphi \wedge C$
  $\varphi_h^T \leftarrow \varphi$
end

3.4 Optimization function

So far, we have presented a method for creating a clausal formula φ_h^T in propositional logic, consisting only of hard clauses, from which we can extract a valid plan from any satisfiable truth assignment to that formula. In order to guide the search procedure to plans of high net benefit, we need to specify an optimization function over solutions. Our approach is to construct a clausal formula φ_s^T in propositional logic, consisting only of weighted soft clauses, with the property that an optimal satisfiable truth assignment $\mathcal{S}^\star$ to the WPMax-SAT formula $\Phi^T = \varphi_h^T \wedge \varphi_s^T$ gives a plan $\mathsf{Plan}(\mathcal{S}^\star)$ of maximum net benefit over all possible plans of makespan less than or equal to T.

We assume that our utility function u has an additive decomposition over $G_1, \ldots, G_k$ as given in Equation 3.4. We introduce a measure which we call the *residual utility* resulting from a truth assignment to the arguments of a function f_i in the additive decomposition of u. The residual utility is the amount of utility that we failed to secure by choosing this truth assignment over one that would maximize the utility for this factor. More precisely, let the maximum of the function be

$$\bar{f}_i = \max_{\mathbf{v} \in \mathcal{B}^{|G_i|}} f_i(\mathbf{v}). \tag{3.8}$$

Define the function $r_i \colon \mathcal{B}^{|G_i|} \to \mathbb{N}_0$ that calculates the residual utility of a truth assignment $\mathbf{v} \in \mathcal{B}^{|G_i|}$ to the arguments of f_i as

$$r_i(\mathbf{v}) = \bar{f}_i - f_i(\mathbf{v}). \tag{3.9}$$

Using this measure, and assuming that all facts in $\mathcal{G}$ are coded for as goals in φ_h^T, we construct φ_s^T such that it satisfies the property that for every complete satisfiable assignment, $\mathcal{S}$ to Φ^T, the following holds:

1. For each action a that is executed in $\mathsf{Plan}(\mathcal{S})$, a unique clause is violated in φ_s^T with weight $c(a)$.
2. For each G_i in the additive decomposition of u, a unique clause is violated in φ_s^T with weight $r_i(\mathbf{1}^{G_i}_{G_i \cap S})$ where S is the set of goals achieved by $\mathsf{Plan}(\mathcal{S})$.
3. No other clauses are violated.

If φ_s^T satisfies this property, then the sum of weights of violated clauses for such a truth assignment $\mathcal{S}$ will be given by

$$\sum_{i=1}^{k} r_i\Big(\mathbf{1}^{G_i}_{G_i \cap \mathsf{Goals}(\mathsf{Plan}(\mathcal{S}))}\Big) + \sum_{i=1}^{m} c(a_i), \tag{3.10}$$

where $\mathsf{Plan}(\mathcal{S}) = (a_1, \ldots, a_m)$. An optimal WPMax-SAT solver applied to Φ^T will find the truth assignment $\mathcal{S}^\star$ that minimizes this quantity which is equivalent to maximizing its negative. Therefore, $\mathcal{S}^\star$ is given by

$$\mathcal{S}^\star = \underset{\substack{\mathcal{S} \text{ s.t.} \\ [\![\varphi_h^T]\!]_{\mathcal{S}} = \mathbf{t}}}{\operatorname{argmax}} \left[\sum_{i=1}^{k} f_i\Big(\mathbf{1}^{G_i}_{G_i \cap \mathsf{Goals}(\mathsf{Plan}(\mathcal{S}))}\Big) - \sum_{i=1}^{m} c(a_i) \right], \tag{3.11}$$

where the a_i and m are dependent on $\mathcal{S}$ such that $\mathsf{Plan}(\mathcal{S}) = (a_1, \ldots, a_m)$. Note that the quantity being maximized is a form of Equation 3.6; thus, a WPMax-SAT solver applied to Φ^T will find a valid plan that is optimal up to the makespan T with regard to maximizing the net benefit metric.

3.4.1 Encoding the optimization function

We have seen that if φ_s^T satisfies the property we outlined above, then the plans produced maximize net benefit for a fixed makespan. Now we discuss the details of how such a formula is constructed according to our procedure shown in Algorithm 4.

The first part of the property is encoded in lines 4–6 where a clause is added to φ_s^T for each action that is applicable at each level up to the makespan T. If an action a is executed at level t in a plan extracted from a truth assignment, then $V_{act}(a, t)$ is necessarily true from the definition of φ_h^T. Consequently, the clause $(\neg V_{act}(a, t))$ with weight $c(a)$ is violated and $c(a)$ is added to the cost of the truth assignment. If the action is not executed, then $V_{act}(a, t)$ is false and its corresponding clause is satisfied and makes no contribution to the cost of the assignment.

The second part of the property is ensured by lines 7–17; the objective is to produce, for each G_i and each truth assignment to facts in G_i, a soft clause, weighted by the residual utility, that is violated iff the facts in G_i take on that truth assignment. Our procedure is made more complicated by accounting for situations where one or more facts in $\mathcal{G}$ are not coded for as goals in φ_h^T.

If a particular G_i is being processed, for each truth assignment, the facts in G_i are split into two sets, π^+ and π^-, depending on whether the fact is assigned true or false respectively (lines 9–10). We then check to see if the truth assignment might be possible on line 11 by checking if any pair of facts in π^+ is known to be mutex at the final level of the plan. If this is true then the truth assignment will never satisfy φ_h^T so there is no need to add a clause for this particular truth assignment to G_i to φ_s^T.

If at least one fact in π^+ is not coded for as a goal, then this tells us that this fact cannot be achieved by any plan of makespan less than or equal to T; thus, this truth assignment

Algorithm 4: Encoding the optimization function (φ_s^T).

Result: φ_s^T encoding the optimization function.

1 **begin**
2 $\quad \Omega_T \leftarrow \{\, \rho_j \in \mathcal{G} \mid \rho_j$ is coded for as a goal in $\varphi_h^T \,\}$
3 $\quad \varphi \leftarrow$ **true**
4 $\quad$ **for** $t \leftarrow 0$ **to** $T-1$ **do**
5 $\quad\quad$ **foreach** *action* $a \in \mathcal{A}$ *applicable at level* t **do**
$\quad\quad\quad \varphi \leftarrow \varphi \wedge \underbrace{(\neg V_{act}(a,t))}_{c(a)}$
6
7 $\quad$ **for** $i \leftarrow 1$ **to** k **do**
8 $\quad\quad$ **foreach** $\mathbf{v} \in \mathcal{B}^{|G_i|}$ **do**
9 $\quad\quad\quad \pi^+ \leftarrow \{\rho_j \in G_i \mid v_j = 1\}$
10 $\quad\quad\quad \pi^- \leftarrow \{\rho_j \in G_i \mid v_j = 0\}$
11 $\quad\quad\quad$ **if** `MutexFree(`π^+`,`T`)` **and** $\pi^+ \subseteq \Omega_T$ **then**
12 $\quad\quad\quad\quad L \leftarrow \{\neg V_{pred}(\rho, T) \mid \rho \in \pi^+\}$
13 $\quad\quad\quad\quad L \leftarrow L \cup \{V_{pred}(\rho, T) \mid \rho \in \pi^- \cap \Omega_T\}$
14 $\quad\quad\quad\quad$ **if** $L \neq \emptyset$ **then**
$\quad\quad\quad\quad\quad \varphi \leftarrow \varphi \wedge \underbrace{\left(\bigvee_{\ell \in L} \ell\right)}_{r_i(\mathbf{v})}$
15

16 $\quad \varphi_s^T \leftarrow \varphi$
17 **end**

and its corresponding clause should be ignored. This is the reason for the check $\pi^+ \subseteq \Omega_T$ on line 11.

At lines 12 and 13 we gather the set of literals for the clause. We negate variables corresponding to facts in π^+ and leave as positive literals the variables corresponding to facts in $\pi^- \cap \Omega^T$. If the truth assignment is made, then all literals will be false and the clause will be violated. Notice how we exclude any facts that are in $\pi^- \setminus \Omega_T$ because they are unreachable at the final level and cannot be achieved by any plan of makespan less than or equal to T, consequently they are fixed to false. The check at line 14 handles the special case where the truth assignment assigns false to all facts in G_i and none of these facts are coded for as goals in φ_h^T. This results in an empty clause that is always violated; therefore, we need not include it in the encoding since it will not affect the minimization. Finally, at line 15 the clause is added with weight set to the residual utility of the truth assignment to the facts in G_i.

We implemented this procedure on top of the SATPLAN06 system. We modified the parser and lexer to read in a specification of action costs and a description of the utility function.

To represent the utility function we use a UCP-net[2] (Boutilier et al. 2001). As discussed in Section 3.3.1, each node in the UCP-net representation corresponds to the value of a single goal fact from $\mathcal{G}$. Each node X contains a *conditional preference table* which we specialize to a tabular representation of a pseudo-Boolean function $f_X \colon \mathcal{B}^{|Parents(X)|+1} \to \mathbb{N}_0$ since all variables in the UCP-net are Boolean-valued. f_X represents X's contribution to the utility of a plan dependent on its value and those of its parents.

3.5 Related work

Using Max-SAT to model hard and soft constraints in optimisation problems has been studied for Steiner trees (Jiang et al. 1995), although this did not use the weighted partial Max-SAT variant, so hard constraints were modelled with very high numeric weights.

Although preferences are receiving an increasing amount of attention from the planning community, there has been little work examining how the planning as satisfiability paradigm can handle preferences. PLAN-A is a SAT-based fixed-horizon cost-optimal planner that uses a custom DPLL procedure that adds blocking clauses to avoid revisiting satisfiable assignments (Chen et al. 2008). This allows the full collection of valid plans to be systematically considered; however, it does not seem necessary to add these blocking clauses as the search can be made systematic in the implementation of the backtracking algorithm. At each stage, the best solution is retained, and pruning occurs when the current cost of a partial assignment exceeds the best solution found so far.

SATPLAN(P) handles quantitative and qualitative preferences using a custom DPLL solver that branches according to a preference order (Giunchiglia and Maratea 2007). In particular, for a problem with quantitative preferences, the value of the optimisation function is encoded as a sequence of bits, and the preference order prefers higher/lower order bits to be set depending on whether the optimisation function is to be maximised/minimised. Their experimental results only cover the cases where either (1) each goal is soft with a utility of 1 and there are no action costs or (2) all goals are hard and all actions have cost 1. Thus, it remains to be seen how their approach scales for optimisation functions that are more flexible.

GAMER solves PSP net benefit problems with a breadth-first style search that uses a binary decision diagram (BDD) to compactly represent sets of states (Edelkamp and Kissmann 2009). The search progresses in layers of increasing cost, with each layer representing a collection of states that are reachable with the same cost. A separate BDD is used to represent the set of states contained in each layer. The search maintains the

[2]We ignore the *ceteris paribus* condition that ensures the dominance property at each node because we compile the function to WPMax-SAT where it is not exploited by the solver.

best solution found so far and terminates when the cost exceeds this or all preferences are satisfied. Since the latter case has highest utility and the search progresses through states of increasing cost, the net benefit can only decrease from that point onwards so the search can be terminated.

The heuristic search planners AltAlt and Sapa have been extended to solve PSP problems (Van Den Briel et al. 2004); the former heuristically selects a set of goals to plan for, and Sapa uses an A* search with a heuristic that estimates the extra net benefit available from extending the current partial plan. Sapa has also been extended to handle goal utility dependencies (Do et al. 2007). Its heuristic calculation first greedily constructs a relaxed plan that supports all reachable goals; it then encodes a problem in IP to find the most beneficial plan contained in the relaxed plan.

TCP-nets (tradeoff-enhanced CP-nets) extend the CP-net model to allow the user to optionally specify when certain preferences are more important than others (Brafman et al. 2002). This model also allows the importance ordering to be conditional on the values of particular outcomes. Using TCP-nets to model conditional qualitative goal preferences has been implemented in a CSP-based planner (Brafman and Chernyavsky 2005). The solution search procedure is altered to respect the TCP-net by first assigning to variables that appear as goals in the final layer of the planning graph and have no parents in the TCP-net that describes the user's preferences. The value ordering heuristic is altered to respect the TCP-net by trying more preferred values first.

An alternative method for selecting goals to plan for is to represent an abstracted part of the planning problem as an orienteering problem (Smith 2004). This is motivated by oversubscription planning problems relevant to the Mars rover where the cost of achieving goals depends strongly on the order in which they are achieved. The aim is to model the cost dependencies between achieving goals, but this ignores the idea of goal utility dependencies. If such goal dependencies are sufficiently localised so that none exist between 'cities' in the orienteering graph, then a system such as iPUD or MSatplan might find use in producing reward estimates for each city provided that the subproblems are small enough and the computation time constraints are sufficiently generous.

3.5.1 Cost-optimal planning as satisfiability

One of the disadvantages of a compilation approach to planning has been that the planning horizon, or *makespan*, must be specified in advance. Without this, the planner must repeatedly construct and solve encodings for increasing makespans until a plan can be found. While this approach is not ideal, it maintains completeness and optimality with regard to finding the plan of shortest makespan; however, when considering cost-optimal planning, if we have found a solution of makespan n, it is possible that a plan of lower cost

exists at a makespan greater than n. Hence, a cost-optimal, SAT-based planner following this strategy, would not know when to terminate because it is not equipped with the power to deduce that lower cost plans of higher makespans are not possible.

The first system to satisfactorily address this shortcoming of planning as SAT is Cos-P (Robinson et al. 2010), which uses a WPMax-SAT encoding of STRIPS planning problems with action costs. Cos-P encodes a planning problem of makespan n as expected but adds a relaxed suffix to the encoding based upon a delete relaxation of the planning problem. In the delete relaxation, actions have no delete effects, so the cost of optimal plans in the delete relaxation are lower bounds to the costs of plans that start from the same state and achieve the same goals in the non-relaxed problem. The relaxed suffix also includes Boolean variables for a collection of causal links, which were introduced in SAT encodings by Kautz et al. (1996). A causal link is used to specify that a proposition supports another proposition, i.e. for propositions p_1, p_2 such that there is an action a where p_1 is a precondition of a and p_2 is in the add list of a, then there is a causal link from p_1 to p_2.

Causal links were originally introduced by McAllester and Rosenblitt (1991), where they were used as constraints to indicate that a step achieves a precondition for another step and that any other step that deletes that precondition cannot possibly interleave these two steps. In Cos-P, causal links are only used in the delete relaxation where there can be no interference between actions. Instead, causal links are used in Cos-P to express how propositions are achieved and to prevent a proposition supporting itself in the relaxed plan. In effect, they are used in a collection of axioms that together enforce the property that a proposition is true in the relaxed plan if and only if there is a supporting chain from propositions that are true in the final layer of the prefix.

Cos-P has two encodings: Variant I which is a standard encoding for makespan n and Variant II which is Variant I with the added relaxed suffix. Since Cos-P uses an optimal WPMax-SAT solver, an optimal solution to the Variant II encoding will yield the lowest cost method of achieving some (possibly empty) subset of the goals by level n and then achieving the remaining goals in the relaxed suffix. This is a lower bound to the actual lowest cost of achieving all goals. A solution to Variant I will yield the lowest cost method of achieving all goals by level n. Since both encodings force at least one proper action to occur at each step up to n, there will eventually be some n where all goals are achieved by level n in Variant II with lowest cost. Hence, if the optimal solutions to the Variant I and Variant II encodings are equal then we know that we cannot do better than the lowest cost plan we have found of makespan n. Otherwise, we repeatedly increase n by one, solving at each step until the cost of optimal solutions to Variant I and Variant II are equal.

Keyder and Geffner (2009) show that STRIPS planning with soft goals and action costs

can be compiled to STRIPS planning with only action costs. They report experiments that show that sequential optimising planners operating on such compiled problems can outperform their net benefit optimising variants; however, the data is not particularly convincing: the MIPS-XXL planner solves more problems expressed as net benefit problems than when they are compiled to cost-only problems; the GAMER planner repeats this pattern for two domains and solves more problems when they are compiled to cost-only problems rather than expressed as net-benefit for one domain. The only planner in their data that shows the trend of solving more problems using the cost-only compilation is for the HSP* planner; which ironically, already solves net benefit problems by reducing them to a cost-only representation before running a heuristic regression search similar to that used on the sequential optimal encoding.

Keyder and Geffner also discusses how this compilation step can be extended to compile utilities defined on formulas of fluents, which is relevant to goal utility dependencies, but they do not present an empirical study of such a compilation. We believe that it is premature to abandon research in this area as it has received only a small amount of attention so far – one set of benchmarks designed for a single competition.

3.6 Summary

We have reviewed the concept of utility dependence and existing representations of user's preferences. This motivated using an additive decomposition to represent the user's utility function so that it is possible to accept UCP-net descriptions of preferences for our encoding. We presented an encoding scheme that represents planning problems of a fixed horizon as a weighted partial Max-SAT formula. The optimal solution of this formula corresponds to a plan of optimal net benefit for that horizon.

Chapter 4

Survey propagation

In this chapter we review the survey propagation method and extend its derivation to handle WPMax-SAT formulae resulting in a new set of message passing equations. We then present a method for using information obtained from fixed points of these equations to make branching decisions in a WPMax-SAT solver.

4.1 Introduction

In the last chapter we presented an encoding scheme for solving planning problems that consist of utility dependencies between the goals achieved by a plan. We did this by representing the problem as a weighted partial Max-SAT formula, which is then solved by a general purpose solver. Much work needs to be done to increase the effectiveness of such solvers on large problems that have practical applications. Moreover, in the presence of many valid solutions, these solvers will require new heuristics to guide them towards the best solutions. This chapter will explore a scheme for incorporating a method known as survey propagation, which has been successful at solving large hard k-SAT problems, to provide heuristic guidance in a weighted partial Max-SAT solver.

The k-SAT problem is a instance of Boolean satisfiability where each clause contains exactly k literals. The study of randomly generated k-SAT problems is an area of significant research interest that aims to understand why some problems are harder than others for algorithms to solve. For a randomly sampled k-SAT formula with M clauses and N variables, there is strong empirical evidence to suggest that the satisfiability of the formula is determined by its clause to variable ratio, $\alpha = M/N$, in the limit $N \to \infty$. Moreover, for each $k \geq 2$, experiments suggest that a phase transition occurs about a unique ratio α_k^t where random k-SAT formulas with $\alpha < \alpha_k^t$ are almost always satisfiable and those with $\alpha > \alpha_k^t$ are almost always unsatisfiable.

Intuitively, one can view these two regions as being over- or under-constrained, respectively. The threshold α_k^t at which the switch from being under-constrained to over-constrained is referred to as the satisfiability phase transition. Deciding the satisfiability of a random k-SAT formula appears to be hardest – those problems exhibit the longest runtimes – for formulas with α in the region of this phase transition ratio (Cheeseman et al. 1991; Mitchell et al. 1992). The skewness in running times for 3-SAT appears to be solver dependent and for some solvers can vary significantly with α (Coarfa et al. 2000).

Survey propagation (Mézard et al. 2002; Braunstein et al. 2005) is a technique developed within the statistical physics community, which has led to an algorithm that can solve a high proportion of random k-SAT formulas close to the phase transition ratio α_k^t. It has also helped to develop new hypotheses for why problems with clause-to-variable ratios close to the phase transition are particularly difficult to solve by current algorithms.

Survey propagation operates on a graphical representation of a SAT formula, known as a factor graph. This representation consists of separate nodes for each variable and clause that occurs in the SAT formula, with edges connecting variable nodes to clause nodes to indicate when a variable appears in a clause. A message passing procedure then takes place where information is exchanged along each edge in the factor graph. The purpose of this message passing is to estimate for each variable, the probabilities of it taking the value *True*, taking the value *False*, or being unconstrained in a randomly chosen satisfying assignment.

A message passed from a clause to a variable estimates the chance of that variable having to take the value that satisfies that clause. A message passed from a variable to a clause estimates the chance of that variable taking a value that will not satisfy that clause. These messages are computed according to the survey propagation equations. The computation of each outgoing message from a node only requires the knowledge of incoming messages to that node, so the messages are said to be locally computed.

Once this collection of probabilities has been estimated, one can calculate the bias of a variable, which is the difference between the estimated probability of that variable taking the value *True* compared to the probability of it taking the value *False*. One can then identify the subset of variables that have a strong bias towards taking a particular value. By fixing a percentage of these strongly biased variables, the problem can be simplified by removing satisfied clauses, fixing pure literals and performing unit propagation where possible. We can then apply another round of message passing using the survey propagation equations to arrive at new bias estimates for this simplified problem and set any highly biased variables. This process is known as *decimation*, and it continues until survey propagation no longer identifies any strongly biased variables. Hopefully, this happens when the problem has been simplified enough that the application of a local search solver will find a satisfying assignment to the remaining unassigned variables.

The underlying techniques of survey propagation can be applied to solve the Max-SAT problem. This variant of survey propagation is known as the SP(y) algorithm (Battaglia et al. 2004). An function is used to describe the cost of an assignment to variables. For a given assignment, each unsatisfied clause contributes an increase to the energy associated with that assignment. In the classical Max-SAT problem, we search for an assignment to variables that minimises the number of unsatisfied clauses, which is equivalent to seeking an assignment of minimum energy. This chapter will review the ideas behind the survey propagation method and will show how they can be adapted to derive a new set of message passing equations that can be used on WPMax-SAT problems. We will conclude by describing how this can be incorporated into a WPMax-SAT solver to provide heuristic guidance early on in the search.

4.2 Applications to random k-SAT

One can generate a random k-SAT problem by sampling with replacement from all possible clauses of k literals that do not contain duplicate variables. For a fixed number of variables N, one needs to generate $M = \alpha N$ such clauses to generate a random k-SAT problem for a fixed density α. If we take a collection of randomly generated k-SAT problems with density α, we can calculate the percentage of problems that are found to be satisfiable after applying a local search solver to them. This leads to an empirical estimate for the probability that a randomly generated k-SAT problem of density α will be satisfiable.

Recent theoretical results have placed the following lower and upper bounds on the satisfiability phase transition α_3^t for 3-SAT: $3.52 \leq \alpha_3^t$ (Kaporis et al. 2006) and $\alpha_3^t \leq 4.4898$ (Díaz et al. 2009). Figure 4.1 shows the above empirical estimates obtained for several densities between the theoretical bounds for two different approaches: applying WalkSAT alone and running survey inspired decimation before running WalkSAT. One can see that by using survey inspired decimation, WalkSAT is able to find more satisfiable assignments closer to the phase transition, which is expected to be around 4.27 (Mézard and Zecchina 2002). One can also observe that the known theoretical bounds are quite loose compared to the empirical observations.

Rintanen et al. (2004) examined the phase transition in solubility for randomly generated classical planning problems. Operators are generated randomly, each with 3 preconditions and 2 effects, by selecting from the set of state variables. Further restrictions are placed on the set of operators to avoid the case where a problem is insoluble simply because a goal does not appear in the effect of any operator. The ratio of number of operators to number of state variables is varied and a transition in solubility takes place such that as the ratio increases, a randomly generated problem with that ratio is more likely to have

a solution. The phase transition appears to take place over a larger range in the ratio than is observed for 3-SAT. Although this may sharpen when the number of variables is increased. Rintanen et al. also presents results that show a peak in difficulty for problems with certain ratios. This suggests a method for creating inherently difficult problems for evaluating planning algorithms.

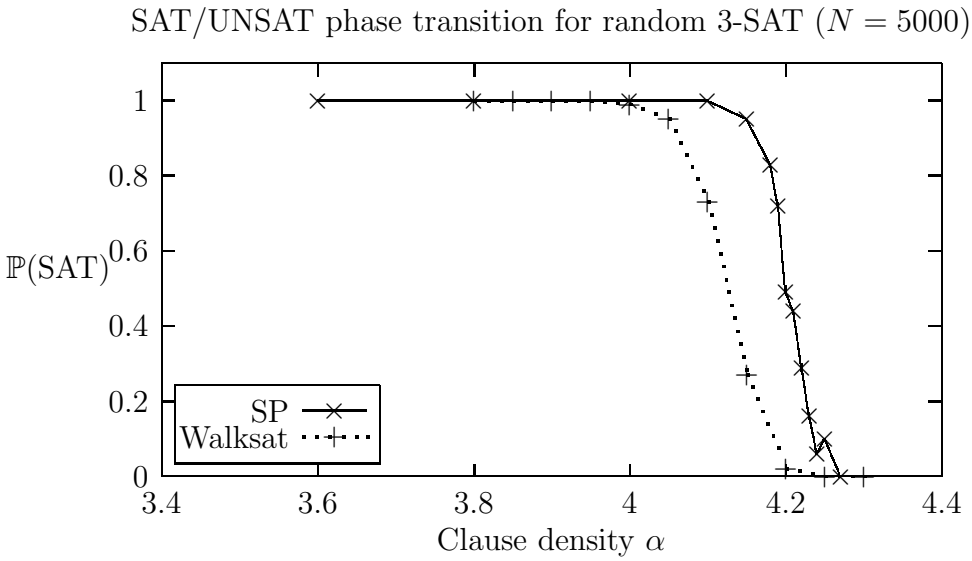


Figure 4.1: Empirically observed chance of finding a satisfiable assignment to a random 3-SAT problem with $N = 5000$ variables for different clause densities ($\alpha = M/N$ for M clauses). We compared running Walksat (cutoff $= 1 \times 10^6$) against running survey propagation based decimation to set a maximum of 2000 variables before running Walksat with the same cutoff. 100 trials were conducted for each α/solver pair to arrive at the probability estimates.

The survey propagation equations can be derived by applying belief propagation to compute marginals on an extended space, where variables can take a 'joker' value when they are not constrained by clauses and other assignments to take the value *True* or *False* (Braunstein and Zecchina 2004; Maneva et al. 2007). This leads to the idea of a *cover* (Achlioptas and Ricci-Tersenghi 2006), which is a total assignment to variables involving the values $\{0, 1, *\}$, where $*$ denotes the 'joker' state. For a particular partial assignment, a *supported* variable X_i is one that is assigned the value 0 or 1 with the constraint that there is a clause C such that a literal containing X_i satisfies C and all other literals in C evaluate to *False* under the assignment. A cover is a total assignment of values $\{0, 1, *\}$ for which every clause contains either at least two literals that are assigned the value $*$, or a literal that satisfies the clause; and every variable that is assigned 0 or 1 is a supported variable. The trivial cover which assigns $*$ to all variables exists for all formulae, even those that are unsatisfiable. A *true* cover is one which can be extended to a satisfiable total assignment.

Maneva et al. (2007) shows that for a particular initial assignment to messages, survey

propagation performs a peeling procedure that sets unconstrained variables to the joker state until no unconstrained variables are left in the assignment, which identifies a cover. Empirical evidence suggests that non-trivial covers can be found by applying this peeling procedure to solutions of random 3-SAT problems, although the fraction of solutions that lead to non-trivial covers becomes vanishingly small as the number of variables increase (Kroc et al. 2007).

Chieu and Lee (2009) extend Maneva et al.'s markov random field (MRV) formulation of survey propagation to handle the weighted Max-SAT problem. A weighted Max-SAT formula is represented as a MRV. Belief propagation is then applied to the MRV to produce a set of message passing equations. These can be understood as estimating marginals over min-covers – an extension of the idea of a cover to Max-SAT. Their approach shows good performance compared to SP(y) on random Max-3-SAT problems. Adapting this MRV formulation to handle WPMax-SAT would be a natural extension to the work we present.

4.3 Belief propagation

Belief propagation (Pearl 1982) is intimately related to the technique of survey propagation. Belief propagation (BP) is a technique for computing marginals on graph-based representations (graphical models) of probability distributions. The marginals it calculates are exact when the graphical model is a tree. In other cases, the marginals it yields may still be close to the exact marginals; consequently, it has been applied to many non-tree graphical models.

Let us use an indexing set $I_n \equiv \{1, \ldots, n\}$ to label the family of variables $\{X_i\}_{i \in I_n}$ where each X_i has the same finite domain $\mathcal{B} = \{0, 1\}$. For this family of variables, consider a full joint probability distribution $p\colon \mathcal{B}^n \to [0, 1]$. We will use lowercase letters x_i as placeholders for the value that a variable X_i takes and adopt $p(x_1, \ldots, x_n)$ as shorthand for $p(X_1 = x_1, \ldots, X_n = x_n)$, for p and other similar functions.

For a subset $S = \{i_1, \ldots, i_k\} \subseteq I_n$ we denote a vector of variables $\mathbf{X}_S \equiv (X_{i_1}, \ldots, X_{i_k})$ and a vector of values $\mathbf{x}_S \equiv (x_{i_1}, \ldots, x_{i_k})$ where $i_1 < i_2 \wedge \cdots \wedge i_{k-1} < i_k$. In such cases, $p(\mathbf{x}_S)$ is shorthand for $p(X_{i_1} = x_{i_1}, \ldots, X_{i_k} = x_{i_k})$. The marginal distribution p_S for a subset of variables $S \subseteq I_n$ is

$$p_S(\mathbf{x}_S) = \sum_{\mathbf{x}_{I_n \setminus S}} p(\mathbf{x}_{I_n}), \tag{4.1}$$

where the sum is over all possible assignments to the variables $\mathbf{X}_{I_n \setminus S}$. Belief propagation is used to calculate the marginal distribution for single variables.

4.3.1 Factor graphs

A *factor graph* (Kschischang et al. 2001) is a graphical representation of a function that is expressed as a product of other simpler functions, known as *factors*. Given a subset $S \subseteq I_n$, a *potential function* over S is written as $\psi_S \colon \mathcal{B}^{|S|} \to \mathbb{R}$. We will assume that each factor is expressed as a potential function. Let $A \subset \mathcal{P}(I_n)$ be the set of arguments that the potential functions take, where $\mathcal{P}(I_n)$ is the powerset of I_n. For each $a \in A$, there is a factor expressed as a potential function ψ_a. In summary, to represent a function g over n variables using a factor graph, we will assume that g is expressed as a product of factors:

$$g(\mathbf{x}_{I_n}) = \prod_{a \in A} \psi_a(\mathbf{x}_a). \tag{4.2}$$

Although two factors operating on the same set of arguments can be combined to form a single factor, it will be convenient for us to allow multiple different factors that operate on the same set of arguments to simplify the implementation. The factor graph representation of g has a separate node for each variable X_i, a separate node for each factor ψ_a and edges that connect any variable node for X_i to a factor node for ψ_a if $i \in a$.

A factor graph is a triple (V, F, E) where V is a set of variable nodes, F is a set of factor nodes, and E is a set of undirected edges. For any $i \in V$ and $a \in F$, E contains the edge (i, a) if and only if i represents a variable X_p and a represents a potential function ψ_Q and $p \in Q$. We write the set of neighbours of $i \in V$ as $\partial i = \{a \in F \mid (i, a) \in E\}$ and the set of neighbours of $a \in F$ as $\partial a = \{i \in V \mid (i, a) \in E\}$. Notice that two variable nodes or two factor nodes are never joined together; hence, the factor graph is bipartite. To distinguish between the two classes of nodes in drawings, factor nodes are drawn as squares and variable nodes are drawn as circles.

Throughout the rest of this chapter we will often refer to the variable nodes using placeholders i, j and k; similarly, we will refer to factor nodes using placeholders a, b and c. To simplify the notation we will write x_i for $i \in V$ to denote a value taken by the variable corresponding to node i. Also, for $a \in F$, we will write $\psi_a(\mathbf{x}_{\partial a})$ to denote the application of the factor represented by node a to an assignment of values $\mathbf{x}_{\partial a}$ to its arguments.

Satisfiability example

The satisfiability of a propositional formula consisting of variables $\{X_i\}_{I_n}$ and expressed in conjunctive normal form with clauses $C_1, \ldots, C_m$ can be represented as a product of functions. For each clause C_i, let V_i be the set of indices of variables appearing in that

$$\boxed{\{\neg x_1, \neg x_2\}\ \{\neg x_4, x_2\}\ \{\neg x_4, x_3\}\ \{\neg x_5, x_1\}\ \{\neg x_6, x_5, x_4\}}$$

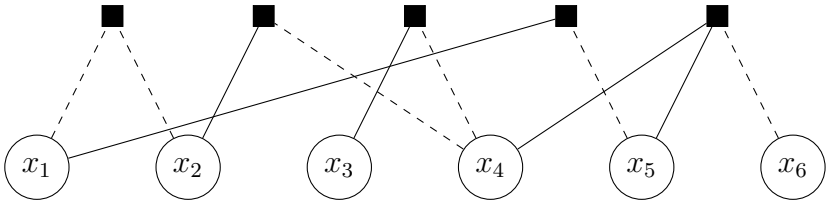


Figure 4.2: An example factor graph representation of a propositional formula shown above. Dotted lines indicate a negative literal and solid lines indicate a positive literal.

clause. For $i = 1, \ldots, m$ define the function ϕ_i as

$$\phi_i(\mathbf{x}_{V_i}) = \begin{cases} 1 & \text{if } C_i \text{ is true for the values } \mathbf{x}_{V_i} \text{ assigned to variables } \{X_j\}_{j \in V_i}, \\ 0 & \text{otherwise.} \end{cases} \tag{4.3}$$

Note that two different clauses may contain the same set of variables, and so our factorisation may consist of multiple factors that have the same set of arguments. Define the function Φ as

$$\Phi(\mathbf{x}_{I_n}) = \prod_{i=1}^{m} \phi_i(\mathbf{x}_{V_i}). \tag{4.4}$$

Then $\Phi(\mathbf{x}_{I_n})$ is equal to 1 if the assignment to variables satisfies all clauses and 0 if at least one clause is violated. Figure 4.2 shows an example factor graph representation for the function Φ. Normalising Φ would give a uniform probability distribution over satisfying assignments to the formula, and we could apply belief propagation to the above graph to estimate marginal probabilities for each variable.

4.3.2 The belief propagation equations

If we are given a factor graph (V, F, E), for each edge $(i, a) \in E$, two messages $\nu_{i \to a}$ and $\hat{\nu}_{a \to i}$ are sent. The subscript of the message indicates the direction the message is travelling. In the case of binary valued variables, each message is a two-vector, and $\nu_{i \to a}(x_i)$ is a real number for the case that the variable, represented by node $i \in V$, takes the value x_i. The belief propagation equations are:

$$\nu_{i \to a}^{(t+1)}(x_i) = \alpha_{ia} \prod_{b \in \partial i \setminus a} \hat{\nu}_{b \to i}^{(t)}(x_i), \tag{4.5}$$

$$\hat{\nu}_{a \to i}^{(t+1)}(x_i) = \sum_{\mathbf{x}_{\partial a \setminus i}} \psi_a(\mathbf{x}_{\partial a}) \prod_{j \in \partial a \setminus i} \nu_{j \to a}^{(t)}(x_j). \tag{4.6}$$

$\nu^{(k)}_{i\to a}(x_i)$ and $\hat{\nu}^{(k)}_{a\to i}(x_i)$ are the messages at iteration k. We use the notational convention of placing a circumflex above a message that is sent from a function node to a variable node. α_{ia} is a normalisation constant so that $\nu_{i\to a}(1) + \nu_{i\to a}(0) = 1$. When $\partial i \setminus a = \emptyset$, $\nu_{i\to a}(x_i) = 0.5$. When $\partial a \setminus i = \emptyset$, $\hat{\nu}_{a\to i}(x_i) = \psi_a(\mathbf{x}_{\partial a})$.

The equations can be understood as the graphical equivalent of variable elimination or 'pushing a summation' inside a product of factors, for example,

$$\sum_{x_1,x_2,x_3} f_1(x_1)f_2(x_1,x_2)f_3(x_3) = \sum_{x_1} f_1(x_1) \sum_{x_2} f_2(x_1,x_2) \sum_{x_3} f_3(x_3).$$

The belief propagation algorithm iteratively applies Equations 4.5 and 4.6 until convergence. In practice, a convergence precision ϵ is used so that the iterative process ends when no message changes by more than ϵ during an iteration. Unless otherwise stated, we use a precision of $\epsilon = 0.001$ in our work. Although we may talk about finding fixed-points of message-passing equations, when we use ϵ to determine convergence, we are in fact only returning approximate fixed-points and so results derived from these approximate fixed-points are also only approximations.

If convergence is reached, an estimate of the marginal probability for a variable can be found by calculating

$$\alpha \prod_{a\in\partial i} \hat{\nu}^{*}_{a\to i}(x_i), \tag{4.7}$$

where α is a normalisation constant. Here we have used the notation of adding an asterisk as a superscript to messages that belong to an approximate fixed-point of the message-passing equations, and this notation will be used later in this section.

For a tree-structured factor graph, it is easy to see that belief propagation is guaranteed to converge and will calculate exact marginals. By rooting the tree at any variable node and applying the BP equations from the leaves, working upwards, the correct marginal estimate is made for the root. After a number of iterations equal to the length of the longest path in the factor graph, all information will have propagated throughout the graph, and there will be no further change in the messages.

4.3.3 Related work

Belief propagation has found a successful application in the decoding of low density parity check codes, or Gallager codes, which achieve performance close to the Shannon limit (MacKay and Neal 1996). The decoding problem involves computing marginal posterior probabilities for each bit of the transmitted codeword given the received codeword, the parity-check matrix and an assumption about the corrupting properties of the channel.

In practice, the decoding problem can be successfully solved by applying belief propagation to compute these marginals despite the factor graph representation of the posterior containing many loops.

Murphy et al. investigated whether this good performance was transferable to a wider collection of problems other than error correcting codes (Murphy et al. 1999). Their empirical data showed that belief propagation converged to accurate marginals for several types of problems that contained many small loops without highly peaked posteriors; however, there were also groups of problems that produced oscillations in message passing and failed to converge. This could be partly mitigated by applying an exponential moving average to message values, but while this increased the chance of convergence, the marginals were sometimes found to be inaccurate.

The main criterion for success in decoding is the identification of the correct codeword; it is not important that the marginals we use to do this decoding are not entirely accurate. This is similar to the process of solving satisfiability problems, where we are usually interested in finding a satisfiable assignment rather than computing accurate marginals. This is why we do not test the accuracy of the bias estimates obtained by survey propagation and its related methods in our work.

One might wonder why loopy belief propagation has any success at all. One possible reason is that fixed points of the belief propagation equations correspond to stationary points of the Bethe free energy (Yedidia et al. 2003). This provides us with an alternative when belief propagation does not converge: minimise the Bethe free energy which is guaranteed to terminate. Empirical data has suggested that this approach produces similar results to belief propagation (Welling and Teh 2001).

This link between belief propagation and the Bethe approximation to the free energy has inspired propagation methods based on increasingly accurate approximations to the free energy, known as Kikuchi approximations; these methods are often collectively referred to as *generalised belief propagation* (Yedidia et al. 2000). These approximations are obtained by grouping variables into *regions* and summing the free energies of each region, then subtracting the free energies of over-counted intersections of regions, then adding over-counted free energies of intersections of intersections of regions, and so on. Messages are then passed from regions to their subregions. The accuracy improves with the size of the regions chosen to the point where the approximation is exact when a region surrounds the entire graph; however, as one might expect from a method with such a property, the complexity grows exponentially with the size of the regions. The practical applications of generalised belief propagation are for graphs with many small loops where most of the error that belief propagation generates can be eliminated by considering many small regions that surround those loops, which results in only a small increase in computational complexity.

The Expectation Maximisation Belief Propagation (EMBP) algorithm is an attempt to adapt the ideas of belief propagation, when applied to satisfiability, to guarantee convergence (Hsu and McIlraith 2006). The idea is to parameterise the problem through a vector θ that contains an estimate, for each variable in the SAT formula, of that variable being set to *True*. We then seek the maximum likelihood value of θ (that which maximises $P(\text{SAT}|\theta)$). A collection of latent binary variables is assumed $\{s_{c,v}\}$ where $s_{c,v}$ is set to *True* if variable v is the sole support of clause c (the only variable that satisfies clause c). A distribution over these latent variables is defined in terms of the parameters θ, and by applying the expectation maximisation algorithm, an update rule for θ can be derived. In Hsu and McIlraith's experimental results, EMBP behaves significantly worse compared to BP when applied to random 3-SAT problems, which would lead one to conclude that EMBP is not simply a version of BP that is guaranteed to converge. It would be interesting to investigate an approach that minimises the Bethe free energy directly for random 3-SAT problems to check whether the behaviour is the same as BP.

4.3.4 Maximum marginals

A variant of belief propagation is the max-product algorithm (Kschischang et al. 2001). When applied to a factor graph that represents a function μ over variables $X_1, \ldots, X_n$, the max-product algorithm aims to find the maximum marginal $\mu_i^{\max}$ for each variable X_i as given by

$$\mu_i^{\max}(x_i) = \max_{\substack{x_1,\ldots,x_{i-1},\\ x_{i+1},\ldots,x_n}} \mu(x_1, \ldots, x_n). \tag{4.8}$$

We are interested in computing this property because when solving WPMax-SAT problems, we want to find a truth assignment to variables of lowest cost. If we can find a suitable distribution such that an assignment to variables that maximises μ is also an assignment of minimum cost, we can apply the max-product equations to find such an assignment. The max-product equations are given as

$$\nu_{i\to a}^{(t+1)}(x_i) = \alpha_{ia} \prod_{b\in\partial i\setminus a} \hat{\nu}_{b\to i}^{(t)}(x_i), \tag{4.9}$$

$$\hat{\nu}_{a\to i}^{(t+1)}(x_i) = \alpha_{ai} \max_{\mathbf{x}_{\partial a\setminus i}} \left[\psi_a(\mathbf{x}_{\partial a}) \prod_{j\in\partial a\setminus i} \nu_{j\to a}^{(t)}(x_j) \right]. \tag{4.10}$$

α_{ia} and α_{ai} are normalisation constants that do not affect the maximisation but add to stability and avoid numerical underflow. One can see that this set of equations is identical to the belief propagation equations with the exception that the summation is replaced by a maximisation operator in the second equation. This is a result of trying to find the most likely assignment rather than marginalising over other variables.

Given an approximate fixed-point of the max-product equations, an estimate of the max-marginal for variable X_i is

$$\mu_i^{\max}(x_i) = \alpha \prod_{a \in \partial i} \hat{\nu}^*_{a \to i}(x_i), \tag{4.11}$$

where α is a normalisation constant. Like belief propagation, if the max-product algorithm is applied to a tree-structured factor graph, it will converge to a fixed-point and will compute exact maximum marginals. If μ is the full joint distribution, then the maximum marginal for each variable can be used to find the most likely assignment to variables on a tree-structured factor graph. This is done by first running the max-product algorithm to convergence, and then setting the variable X_i with the highest maximum marginal to the value that maximises $\mu_i^{\max}$. A new factor graph is constructed to represent μ with X_i's value fixed, and the procedure is repeated until all variables are set.

4.4 Survey propagation

In this section we review the derivation of the optimisation variant of the survey propagation algorithm, often referred to as the SP(y) algorithm (Battaglia et al. 2004). Survey propagation is a special case of the SP(y) algorithm that arises in the limit $y \to \infty$. This corresponds to the requirement that all clauses must be satisfied.

The differences between the derivations of SP and SP(y) are found only in the simplifying assumptions about the problem that lead to an efficient set of equations: they both use the same core method, which is referred to as the 1RSB (Replica Symmetry Breaking) *cavity method*. The 1RSB cavity method makes assumptions about how the probability distribution over variable configurations decomposes and on this basis, it is used to make quantitative predictions about the large-scale properties of the system. Mézard and Zecchina (2002) use the 1RSB cavity method to predict the satisfiability phase transition for 3-SAT as $\alpha_3^t = 4.267$, which coincides with empirical measurements and suggests that the set of assumptions is reasonable for large-scale 3-SAT problems.

There is much that remains to be understood in this area, and we will not dwell on the nature of these assumptions. We do not attempt to calculate quantities such as α_3^t, and so the formal statement of the underlying assumptions is less important. We aim for a practical application of this method where our success is judged by whether the heuristic guidance that is extracted from the results of message passing yields a significant improvement. If it does, then this indicates that the assumptions may be appropriate for the problem and warrant further investigation.

We justify that our message passing equations fit within the cavity method approach. We do this by starting from an intuitive setting that is easy for someone unfamiliar with the

1RSB cavity method. We then summarise the arguments that lead to the 1RSB cavity equations. The reason for this is twofold: it provides an introduction to the method for those unfamiliar with it, and it establishes a vocabulary for describing our contribution.

A full technical derivation of the survey propagation method is presented by Mézard and Zecchina (2002). We summarise the material presented by Mézard and Montanari (2009) that leads to the 1RSB cavity equations, using most of their notation, in Sections 4.4.1, 4.4.2 and 4.4.3.

After the presentation of the 1RSB cavity equations, our work takes a different direction to previous works. Our contributions are as follows. In Section 4.4.4 we present the form that the message passing equations take for weighted partial Max-SAT formulae; we call this set of equations weighted partial Max-SAT survey propagation, or WPSP(y). These are different to the SP(y) equations in that they allow clauses to be specified as hard, and soft clauses can have any positive integer weight rather than being restricted to unit weights. In Section 4.4.7 we present how the variable biases can be computed from a fixed point of the WPSP(y) equations. This can then be used with the normal survey inspired decimation procedure used in SP(y) and SP to set variables and simplify problems.

An overview of how the cavity equations are derived is as follows. They are derived by first defining a Gibbs probability distribution over assignments to variables. We then study the form that the max-product equations take under this distribution and show how the max-product equations simplify to a set of equations called the min-sum equations. Following this, we apply the cavity method to estimate distributions over messages that are exchanged in the min-sum algorithm. Using these distribution estimates, we arrive at a decimation algorithm for deciding how to make assignments to variables in order to simplify the problem.

4.4.1 A factor graph representation of a WPMax-SAT formula

Consider a WPMax-SAT problem over variables $X_1, \ldots, X_n$ with a set of m weighted clauses $\{(C_i, w_i)\}_{i \in I_m}$ as discussed in Section 2.4. The variables take values in $\mathcal{B} = \{0, 1\}$ where a 1 indicates the value *True*, and 0 indicates the value *False*. Here we will define a probability distribution over truth assignments to these variables that has the property that maximums of the distribution correspond to truth assignments of minimum cost. This will then be represented as a factor graph, which we will apply the max-product equations to.

The WPMax-SAT problem described above, already defines the structure of our factor graph (V, F, E) representation, where there is a separate variable node for each variable and a separate factor node for each weighted clause in the WPMax-SAT problem. In order

to describe the distribution over truth assignments, we will adopt the terminology used in survey propagation, which refers to the *energy* of a truth assignment for a particular problem. For each weighted clause (C_a, w_a) in the problem, there is a unique factor $a \in F$. Each weighted clause contributes an energy $E_a(\mathbf{x}_{\partial a})$ for an assignment $\mathbf{x}_{\partial a}$ to the variables appearing in C_a defined as

$$E_a(\mathbf{x}_{\partial a}) = \begin{cases} 0 & \text{if } \mathbf{x}_{\partial a} \text{ satisfies clause } C_a \\ \infty & \text{if } \mathbf{x}_{\partial a} \text{ violates clause } C_a \text{ and } w_a = \top, \\ w_a & \text{otherwise.} \end{cases} \tag{4.12}$$

We can express the total energy $E(\mathbf{x}_{I_n})$ of a truth assignment $\mathbf{x}_{I_n}$ as

$$E(\mathbf{x}_{I_n}) = \sum_{a \in F} E_a(\mathbf{x}_{\partial a}). \tag{4.13}$$

Note that if any hard clause is violated by an assignment, the energy of the whole assignment is infinite. Let us now consider a Gibbs measure over truth assignments to variables $X_1, \ldots, X_n$ as given by

$$\mu(x_1, \ldots, x_n) = \frac{1}{Z} \exp\left(-\beta E(x_1, \ldots, x_n)\right), \tag{4.14}$$

where $Z = \sum_{\mathbf{x}_{I_n}} \exp(-\beta E(x_1, \ldots, x_n))$ is a normalisation constant and β is an inverse temperature parameter. This distribution can be written in the factorised form

$$\mu(\mathbf{x}_{I_n}) = \frac{1}{Z} \prod_{a \in F} \exp(-\beta E_a(\mathbf{x}_{\partial a})). \tag{4.15}$$

It is this factorised form that our factor graph (V, F, E) represents, that is, each $a \in F$ represents a factor $\psi_a(\mathbf{x}_{\partial a}) = \exp(-\beta E_a(\mathbf{x}_{\partial a}))$. Note that we do not include the term $1/Z$ in the factor graph as we will normalise variable-to-factor messages using on-the-fly normalisation as described by MacKay (2003).

We can apply the max-product algorithm to this distribution in order to compute the maximum-marginals for each variable, and in so doing let us take logarithms of both sides of Equations 4.9 and then multiply by $-1/\beta$ to obtain the following variation of the max-product equations:

$$\frac{-1}{\beta} \log\left(\nu_{i \to a}(x_i)\right) = \sum_{b \in \partial i \setminus a} \frac{-1}{\beta} \log\left(\hat{\nu}_{b \to i}(x_i)\right) + \kappa_{ia} \tag{4.16}$$

$$\frac{-1}{\beta} \log\left(\hat{\nu}_{a \to i}(x_i)\right) = \min_{\mathbf{x}_{\partial a \setminus i}} \left[E_a(\mathbf{x}_{\partial a}) - \frac{1}{\beta} \sum_{j \in \partial a \setminus i} \log(\nu_{j \to a}(x_j)) \right] + \kappa_{ai}, \tag{4.17}$$

where κ_{ia} and κ_{ai} are a result of taking logarithms of the normalisation constants in the max-product equations. We are able to take the logarithm inside the maximisation because it is a monotonically increasing function. Taking the minus sign inside the maximisation turns it into a minimisation. Since β is held constant, we can define

$$\eta_{i\to a}(x_i) = \frac{-1}{\beta}\log(\nu_{i\to a}(x_i)), \qquad \hat{\eta}_{a\to i}(x_i) = \frac{-1}{\beta}\log(\hat{\nu}_{a\to i}(x_i)), \tag{4.18}$$

and rewrite the max-product equations as

$$\eta_{i\to a}(x_i) = \sum_{b\in\partial i\setminus a} \hat{\eta}_{b\to i}(x_i) + \kappa_{ia}, \tag{4.19}$$

$$\hat{\eta}_{a\to i}(x_i) = \min_{\mathbf{x}_{\partial a\setminus i}} \left[E_a(\mathbf{x}_{\partial a}) + \sum_{j\in\partial a\setminus i} \eta_{j\to a}(x_j) \right] + \kappa_{ai}. \tag{4.20}$$

These equations are known as the min-sum equations. In a tree-structured factor graph, the message $\eta_{i\to a}(x_i)$ is the minimum total energy achievable by a truth assignment consistent with $X_i = x_i$ in the subtree rooted at variable X_i. Similarly, the message $\hat{\eta}_{a\to i}(x_i)$ is the minimum total energy achievable by a truth assignment consistent with $X_i = x_i$ in the subtree rooted at factor node a. These equations can be used to find the energy minimum-marginal for each variable, which is given as

$$E_i^{\min}(x_i) = \kappa + \sum_{a\in\partial i} \hat{\eta}_{a\to i}(x_i). \tag{4.21}$$

We can now write the maximum-marginal estimate given in Equation 4.11 as

$$\mu_i^{\max}(x_i) = \frac{1}{Z}\exp\left(-\beta \sum_{a\in\partial i} \hat{\eta}_{a\to i}(x_i)\right). \tag{4.22}$$

Observe that $\mu_i^{\max}(x_i)$ takes its maximum value at the same x_i for which $E_i^{\min}(x_i)$ takes its minimum value.

4.4.2 Energy of a truth assignment

We want to find a truth assignment with minimum energy, and we will attempt to do this by applying the min-sum equations. Since the structure of our factor graph is determined by the WPMax-SAT formula, we expect the factor graph to contain loops, and so the min-sum equations are not guaranteed to converge and the minimum marginals will only be approximations. The energy of a total assignment is calculated according to Equation 4.13. If we do find a fixed point of the min-sum equations, we can estimate the

quantity $\min_{\mathbf{x}_{I_n \setminus \partial a}} E(\mathbf{x})$ at each factor $a \in F$ by evaluating

$$E_a(\mathbf{x}_{\partial a}) + \sum_{i \in \partial a} \eta^*_{i \to a}(x_i). \tag{4.23}$$

In order to minimise the energy of an assignment we should start by assigning to the neighbouring variables $\{X_i\}_{i \in \partial a}$ the assignment $\mathbf{x}^*_{\partial a}$ given as

$$\mathbf{x}^*_{\partial a} = \arg\min_{\mathbf{x}_{\partial a}} \left[E_a(\mathbf{x}_{\partial a}) + \sum_{j \in \partial a} \eta^*_{j \to a}(x_j) \right]. \tag{4.24}$$

Note that it may not be possible to achieve a total assignment that is consistent with $\mathbf{x}^*_{\partial a}$ for each $a \in F$. Nevertheless we will use the quantity

$$U^*(\{\eta^*_{i \to a}, \hat{\eta}^*_{a \to i}\}_{(i,a) \in E}) = \sum_{a \in F} E_a(\mathbf{x}^*_{\partial a}), \tag{4.25}$$

to estimate a lower bound to the minimum energy achieveable. We now show that $U^*(\{\eta^*_{i \to a}, \hat{\eta}^*_{a \to i}\}_{(i,a) \in E})$ can be simply expressed as a sum involving the following terms:

$$\mathbb{F}_{(i,a)}(\eta^*_{i \to a}, \hat{\eta}^*_{a \to i}) = \min_{x_i} \left[\eta^*_{i \to a}(x_i) + \hat{\eta}^*_{a \to i}(x_i) \right], \tag{4.26}$$

$$\mathbb{F}_i(\{\hat{\eta}^*_{b \to i}\}_{b \in \partial i}) = \min_{x_i} \left[\sum_{a \in \partial i} \hat{\eta}^*_{a \to i}(x_i) \right], \tag{4.27}$$

$$\mathbb{F}_a(\{\eta^*_{j \to a}\}_{j \in \partial a}) = \min_{\mathbf{x}_{\partial a}} \left[E_a(\mathbf{x}_{\partial a}) + \sum_{j \in \partial a} \eta^*_{j \to a}(x_j) \right]. \tag{4.28}$$

We start by defining

$$x^*_i(a) = \arg\min_{x_i} \left[\eta^*_{i \to a}(x_i) + \hat{\eta}^*_{a \to i}(x_i) \right]. \tag{4.29}$$

Note that $x^*_i(a)$ and $\mathbf{x}^*_{\partial a}$ both depend on a set of fixed-point messages that are a solution to the min-sum equations. We have not made this explicit to avoid cluttering the notation. The reader should keep in mind that these values are all dependent on a single set of fixed point messages that we are implicitly considering. The following three lemmas are presented as exercises without proof in Mézard and Montanari (2009).

Lemma 1. $\mathbb{F}_{(i,a)}(\eta^*_{i \to a}, \hat{\eta}^*_{a \to i}) = \eta^*_{i \to a}(x^*_i(a)) + \hat{\eta}^*_{a \to i}(x^*_i(a))$.

Proof. By definition of $x^*_i(a)$. □

Lemma 2. $\mathbb{F}_i(\{\eta^*_{i \to b}, \hat{\eta}^*_{b \to i}\}_{b \in \partial i}) = \sum_{a \in \partial i} \hat{\eta}^*_{a \to i}(x^*_i(a))$.

Proof. Since $\{\eta^*_{i \to b}, \hat{\eta}^*_{b \to i}\}_{b \in \partial i}$ belong to a fixed-point of the min-sum equations, for $a \in \partial i$,

we can use the expression for $\eta^*_{i\to a}(x_i)$ from Equation 4.19 to write Equation 4.29 as

$$\begin{aligned} x^*_i(a) &= \arg\min_{x_i} \left[\eta^*_{i\to a}(x_i) + \hat{\eta}^*_{a\to i}(x_i)\right] \\ &= \arg\min_{x_i} \left[\sum_{b\in\partial i} \hat{\eta}^*_{b\to i}(x_i) + \kappa_{ia}\right] \\ &= \arg\min_{x_i} \left[\sum_{b\in\partial i} \hat{\eta}^*_{b\to i}(x_i)\right]. \end{aligned}$$

□

Recall the definition of $\mathbf{x}^*_{\partial a}$ in Equation 4.24.

Lemma 3. $\mathbb{F}_a(\{\eta^*_{j\to a}, \hat{\eta}^*_{a\to j}\}_{j\in\partial a}) = E_a(\mathbf{x}^*_{\partial a}) + \sum_{j\in\partial a} \eta^*_{j\to a}(x^*_j(a))$.

Proof. By contradiction. Instead, assume that $\mathbf{x}'_{\partial a} = (x'_{i_1}(a), \ldots, x'_{i_k}(a))$ minimises the right hand side of Equation 4.28. Then there must be at least one i_z such that $x'_{i_z}(a) \neq x^*_{i_z}(a)$. We can write $x'_{i_z}(a)$ as

$$\begin{aligned} x'_{i_z}(a) &= \arg\min_{x_{i_z}} \left[\min_{\mathbf{x}_{\partial a\setminus i_z}} \left\{E_a(\mathbf{x}_{\partial a}) + \sum_{j\in\partial a} \eta^*_{j\to a}(x_j)\right\}\right] \\ &= \arg\min_{x_{i_z}} \left[\eta^*_{i_z\to a}(x_{i_z}) + \min_{\mathbf{x}_{\partial a\setminus i_z}} \left\{E_a(\mathbf{x}_{\partial a}) + \sum_{j\in\partial a\setminus i_z} \eta^*_{j\to a}(x_j)\right\}\right] \\ &= \arg\min_{x_{i_z}} \left[\eta^*_{i_z\to a}(x_{i_z}) + \hat{\eta}^*_{a\to i_z}(x_{i_z}) - \kappa_{ai}\right] \\ &= \arg\min_{x_{i_z}} \left[\eta^*_{i_z\to a}(x_{i_z}) + \hat{\eta}^*_{a\to i_z}(x_{i_z})\right] \\ &= x^*_{i_z}(a), \end{aligned}$$

which contradicts our assumption that $x'_{i_z}(a) \neq x^*_{i_z}(a)$. □

Using the above results, it is easy to verify that

$$\begin{aligned} U^*(\{\eta^*_{i\to a}, \hat{\eta}^*_{a\to i}\}_{(i,a)\in E}) \;=\; & \sum_{a\in F} \mathbb{F}_a(\{\eta^*_{j\to a}\}_{j\in\partial a}) + \sum_{i\in V} \mathbb{F}_i(\{\hat{\eta}^*_{b\to i}\}_{b\in\partial i}) \\ & - \sum_{(i,a)\in E} \mathbb{F}_{(i,a)}(\eta^*_{i\to a}, \hat{\eta}^*_{a\to i}). \end{aligned} \tag{4.30}$$

4.4.3 The cavity method

We assume that there will be multiple fixed point solutions to the min-sum equations for our problems. We want to find the fixed point solution that has the lowest energy

estimate, U^*. The cavity method considers a Gibbs distribution over fixed point solutions to the min-sum equations, constructs a factor graph representation of this distribution and uses belief propagation to estimate the distribution of messages in the min-sum equations.

Auxiliary graph

In this new factor graph, which we refer to as the auxiliary graph, there are $|E|$ variables, one for each edge $(i,a) \in E$ in the factor graph representation of μ described in Section 4.4.1. For each edge $(i,a) \in E$ we index the variable associated with that edge as $(i \mid a)$. We write an assignment to the variable indexed by $(i \mid a)$ as $\mathbf{m}_{(i|a)}$ which is a tuple $(\mathbf{m}^L_{(i|a)}, \mathbf{m}^R_{(i|a)})$, where $\mathbf{m}^L_{(i|a)}$ is a possible message sent from i to a and $\mathbf{m}^R_{(i|a)}$ is a possible message sent from a to i in the factor graph (V, F, E).

For any variable i and the subset of factors S, where S is either ∂i or $\partial i \setminus a$ for some $a \in \partial i$, write the contents of S explicitly as $S = \{a_1, \ldots, a_k\} \subseteq \partial i$. We then define the following:

- $\mathbf{M}_S = (\mathbf{m}_{(i|a_1)}, \ldots, \mathbf{m}_{(i|a_k)})$,
- $\mathbf{M}^L_S = (\mathbf{m}^L_{(i|a_1)}, \ldots, \mathbf{m}^L_{(i|a_k)})$,
- $\mathbf{M}^R_S = (\mathbf{m}^R_{(i|a_1)}, \ldots, \mathbf{m}^R_{(i|a_k)})$.

For example, $\mathbf{M}_{\partial i \setminus a} = (\mathbf{m}_{(i|b_1)}, \ldots, \mathbf{m}_{(i|b_k)})$ where $\partial i \setminus a = \{b_1, \ldots, b_k\}$.

Similarly, for any factor a and the subset of variables V, where V is either ∂a or $\partial a \setminus i$ for some $i \in \partial a$, we write the contents of V explicitly as $V = \{i_1, \ldots, i_k\} \subseteq \partial a$. We then define the following:

- $\mathbf{M}_V = (\mathbf{m}_{(i_1|a)}, \ldots, \mathbf{m}_{(i_k|a)})$,
- $\mathbf{M}^L_V = (\mathbf{m}^L_{(i_1|a)}, \ldots, \mathbf{m}^L_{(i_k|a)})$,
- $\mathbf{M}^R_V = (\mathbf{m}^R_{(i_1|a)}, \ldots, \mathbf{m}^R_{(i_k|a)})$,

For example, $\mathbf{M}^L_{\partial a \setminus i} = (\mathbf{m}^L_{(j_1|a)}, \ldots, \mathbf{m}^L_{(j_k|a)})$ where $\partial a \setminus i = \{j_1, \ldots, j_k\}$.

We consider a Gibbs distribution over solutions to the min-sum equations using the approximation U^* given in Equation 4.30 to the minimum energy.

$$\Psi(\{\eta^*_{i\to a}, \hat{\eta}^*_{a\to i}\}_{(i,a)\in E}) = \frac{1}{Z} \exp\left(-yU^*(\{\eta^*_{i\to a}, \hat{\eta}^*_{a\to i}\}_{(i,a)\in E})\right) \tag{4.31}$$

$$= \frac{1}{Z} \prod_{a\in F} \exp(-y\mathbb{F}_a(\{\eta^*_{j\to a}\}_{j\in\partial a})) \prod_{i\in V} \exp(-y\mathbb{F}_i(\{\hat{\eta}^*_{b\to i}\}_{b\in\partial i})) \times \prod_{(i,a)\in E} \exp(y\mathbb{F}_{(i,a)}(\eta^*_{i\to a}, \hat{\eta}^*_{a\to i})). \tag{4.32}$$

where Z is a normalisation constant and y is an inverse temperature parameter. Again, the factor $1/Z$ will not appear in our factor graph as we will use on-the-fly normalisation. We only want to assign non-zero probabilities to assignments that correspond to solutions of the min-sum equations. We make use of two message transformation functions, T and $\hat{\mathrm{T}}$, that compute the min-sum equations

$$\eta_{i\to a} = \mathrm{T}_{i\to a}(\{\hat{\eta}_{b\to i}\}_{b\in\partial i\setminus a}), \qquad \hat{\eta}_{a\to i} = \hat{\mathrm{T}}_{a\to i}(\{\eta_{j\to a}\}_{j\in\partial a\setminus i}). \tag{4.33}$$

Using the above functions we define the following factors

$$\Psi_i(\mathbf{M}_{\partial i}) = \prod_{a\in\partial i} \mathbb{I}(\mathbf{m}^L_{(i|a)} = \mathrm{T}_{i\to a}(\mathbf{M}^R_{\partial i\setminus a}))\exp(-y\mathbb{F}_i(\mathbf{M}^R_{\partial i})), \tag{4.34}$$

$$\Psi_a(\mathbf{M}_{\partial a}) = \prod_{i\in\partial a} \mathbb{I}(\mathbf{m}^R_{(i|a)} = \hat{\mathrm{T}}_{a\to i}(\mathbf{M}^L_{\partial a\setminus i}))\exp(-y\mathbb{F}_a(\mathbf{M}^L_{\partial a})), \tag{4.35}$$

$$\Psi_{(i,a)}(\mathbf{m}_{(i|a)}) = \exp(y\mathbb{F}_{(i,a)}(\mathbf{m}^L_{(i|a)}, \mathbf{m}^R_{(i|a)})), \tag{4.36}$$

where $\mathbb{I}$ is the indicator function which returns 1 if its argument evaluates to *True* and 0 otherwise. Thus, if for a particular assignment, there is a variable that is assigned a pair of messages that do not follow from the min-sum equations together with the assignments to that variable's neighbours, the whole assignment will have a probability equal to zero. Given the above factorisation of the distribution, the auxiliary graph is the factor graph (V', F', E') that represents this distribution, where V', F' and E' are defined as follows:

$$V' = \{(i \mid a) \mid (i,a) \in E\}$$
$$F' = \{\Psi_a \mid a \in F\} \cup \{\Psi_i \mid i \in V\} \cup \{\Psi_{(i,a)} \mid (i,a) \in E\}$$
$$E' = \{((i \mid a), \Psi_a) \mid (i,a) \in E\} \cup \{((i \mid a), \Psi_i) \mid (i,a) \in E\} \cup \{((i \mid a), \Psi_{(i,a)}) \mid (i,a) \in E\}.$$

Applying belief propagation

To estimate the marginal distribution of messages according to the above Gibbs distribution, we apply the belief propagation algorithm. Note that each variable $(i \mid a)$ in the auxiliary graph is connected to only three factor nodes: Ψ_a, Ψ_i and $\Psi_{(i,a)}$. Moreover, the node $(i \mid a)$ need not pass a message to the factor $\Psi_{(i,a)}$ since that factor is not connected to any other nodes, and $(i \mid a)$ will always receive the same message from $\Psi_{(i,a)}$.

Therefore the belief propagation equations, given in Equations 4.5 and 4.6 simplify into

two cases:

$$\nu_{(i|a)\to\Psi_a}(\mathbf{m}_{(i|a)}) \cong \Psi_{(i,a)}(\mathbf{m}_{(i|a)}) \sum_{\mathbf{M}_{\partial i\setminus a}} \Psi_i(\mathbf{M}_{\partial i}) \prod_{b\in\partial i\setminus a} \nu_{(i|b)\to\Psi_i}(\mathbf{m}_{(i|b)}), \tag{4.37}$$

$$\nu_{(i|a)\to\Psi_i}(\mathbf{m}_{(i|a)}) \cong \Psi_{(i,a)}(\mathbf{m}_{(i|a)}) \sum_{\mathbf{M}_{\partial a\setminus i}} \Psi_a(\mathbf{M}_{\partial a}) \prod_{j\in\partial a\setminus i} \nu_{(j|a)\to\Psi_a}(\mathbf{m}_{(j|a)}). \tag{4.38}$$

$\cong$ indicates equality up to a normalisation constant, as we are using on-the-fly normalisation. Using Lemmas 1, 2 and 3 from the previous section, we can express the following products of factors as

$$\begin{aligned}\Psi_{(i,a)}(\mathbf{m}_{(i|a)})\Psi_i(\mathbf{M}_{\partial i}) &= e^{y\mathbf{m}^L_{(i|a)}(x_i^*(a))} \prod_{b\in\partial i} \mathbb{I}(\mathbf{m}^L_{(i|b)} = \mathrm{T}_{i\to b}(\mathbf{M}^R_{\partial i\setminus b})) \\ &\quad \times \exp\left(-y \sum_{c\in\partial i\setminus a} \mathbf{m}^R_{(i|c)}(x_i^*(c))\right), \end{aligned}\tag{4.39}$$

$$\begin{aligned}\Psi_{(i,a)}(\mathbf{m}_{(i|a)})\Psi_a(\mathbf{M}_{\partial i}) &= e^{y\mathbf{m}^R_{(i|a)}(x_i^*(a))} \prod_{j\in\partial a} \mathbb{I}(\mathbf{m}^R_{(j|a)} = \hat{\mathrm{T}}_{a\to j}(\mathbf{M}^L_{\partial a\setminus j})) \\ &\quad \times \exp\left(-y \sum_{k\in\partial a\setminus i} \mathbf{m}^L_{(k|a)}(x_k^*(a))\right) e^{-yE_a(\mathbf{x}^*_{\partial a})}. \end{aligned}\tag{4.40}$$

When Equations 4.39 and 4.40 are substituted into Equations 4.37 and 4.38, the first term $e^{y\mathbf{m}^L_{(i|a)}(x_i^*(a))}$ of $\nu_{(i|a)\to\Psi_a}(\mathbf{m}_{(i|a)})$ will cancel with the term $e^{-y\mathbf{m}^L_{(i|a)}(x_i^*(a))}$ in the sum wherever $\nu_{(i|a)\to\Psi_a}(\mathbf{m}_{(i|a)})$ is used in Equation 4.38. Similarly, the first term $e^{y\mathbf{m}^R_{(i|a)}(x_i^*(a))}$ of $\nu_{(i|a)\to\Psi_i}(\mathbf{m}_{(i|a)})$ will cancel with the term $e^{-y\mathbf{m}^R_{(i|c)}(x_i^*(c))}$ in the sum wherever $\nu_{(i|a)\to\Psi_i}(\mathbf{m}_{(i|a)})$ is used in Equation 4.37. Therefore, by making the following substitution,

$$\zeta_{i\to a}(\mathbf{m}_{(i|a)}) = \nu_{(i|a)\to\Psi_a}(\mathbf{m}_{(i|a)}) \exp(-y\mathbf{m}^L_{(i|a)}(x_i^*(a))) \tag{4.41}$$

$$\hat{\zeta}_{a\to i}(\mathbf{m}_{(i|a)}) = \nu_{(i|a)\to\Psi_i}(\mathbf{m}_{(i|a)}) \exp(-y\mathbf{m}^R_{(i|a)}(x_i^*(a))), \tag{4.42}$$

we can simplify Equations 4.37 and 4.38 to

$$\zeta_{i\to a}(\mathbf{m}_{(i|a)}) \cong \sum_{\mathbf{M}_{\partial i\setminus a}} \prod_{b\in\partial i} \mathbb{I}(\mathbf{m}^L_{(i|b)} = \mathrm{T}_{i\to b}(\mathbf{M}^R_{\partial i\setminus b})) \prod_{c\in\partial i\setminus a} \hat{\zeta}_{c\to i}(\mathbf{m}_{(i|c)}), \tag{4.43}$$

$$\hat{\zeta}_{a\to i}(\mathbf{m}_{(i|a)}) \cong \sum_{\mathbf{M}_{\partial a\setminus i}} \prod_{j\in\partial a} \mathbb{I}(\mathbf{m}^R_{(j|a)} = \hat{\mathrm{T}}_{a\to j}(\mathbf{M}^L_{\partial a\setminus j})) \prod_{k\in\partial a\setminus i} \zeta_{k\to a}(\mathbf{m}_{(k|a)}) e^{-yE_a(\mathbf{x}^*_{\partial a})} \tag{4.44}$$

The above equations can be written in another form by noting that $\mathbf{m}^L_{(i|a)}$ only depends

on $\mathbf{m}^R_{(i|b)}$ for $b \in \partial i \setminus a$ and $\mathbf{m}^R_{(i|a)}$ only depends on $\mathbf{m}^L_{(j|a)}$ for $j \in \partial a \setminus i$. By writing

$$Q_{i\to a}(x) = \sum_{\mathbf{m}_{(i|a)}} \mathbb{I}(\mathbf{m}^L_{(i|a)} = x)\zeta_{i\to a}(\mathbf{m}_{(i|a)}), \tag{4.45}$$

$$\hat{Q}_{a\to i}(x) = \sum_{\mathbf{m}_{(i|a)}} \mathbb{I}(\mathbf{m}^R_{(i|a)} = x)\hat{\zeta}_{a\to i}(\mathbf{m}_{(i|a)}), \tag{4.46}$$

we can seek a solution to the simpler equations

$$Q_{i\to a}(\mathbf{m}^L_{(i|a)}) \cong \sum_{\mathbf{M}^R_{\partial i\setminus a}} \mathbb{I}(\mathbf{m}^L_{(i|a)} = \mathrm{T}_{i\to a}(\mathbf{M}^R_{\partial i\setminus a})) \prod_{b\in\partial i\setminus a} \hat{Q}_{b\to i}(\mathbf{m}^R_{(i|b)}), \tag{4.47}$$

$$\hat{Q}_{a\to i}(\mathbf{m}^R_{(i|a)}) \cong \sum_{\mathbf{M}^L_{\partial a\setminus i}} \mathbb{I}(\mathbf{m}^R_{(i|a)} = \hat{\mathrm{T}}_{a\to i}(\mathbf{M}^L_{\partial a\setminus i})) \prod_{j\in\partial a\setminus i} Q_{j\to a}(\mathbf{m}^L_{(j|a)})e^{-yE_a(\mathbf{x}^*_{\partial a})}. \tag{4.48}$$

These are known as the 1RSB cavity equations. We will now look at how to find an approximate fixed point of these equations efficiently for the case of a WPMax-SAT formula.

4.4.4 The WPSP(y) message passing equations

The messages $\mathbf{m}^L_{(i|a)}$ and $\mathbf{m}^R_{(i|a)}$ each have two values: a value for $X_i = 1$ and a value for $X_i = 0$. In a tree structured factor graph, each value is the lowest energy achievable in the subtree rooted at a from (i, a) for the respective assignment to X_i. As we discussed earlier, the auxiliary graph is constructed in such a way as to limit the assignments that have non-zero probability to those that correspond to a set of messages that are a fixed point of the min-sum equations of the original factor graph. Therefore $\mathbf{m}^R_{(i|a)}$ is calculated using the equation

$$\mathbf{m}^R_{(i|a)}(x_i) = \min_{\mathbf{x}_{\partial a\setminus i}} \left[E_a(\mathbf{x}_{\partial a}) + \sum_{j\in\partial a\setminus i} \mathbf{m}^L_{(j|a)}(x_j) \right] + \kappa_{ai}. \tag{4.49}$$

According to this equation, the values that $\mathbf{m}^R_{(i|a)}$ can take can be divided into two classes: $\mathbf{m}^R_{(i|a)}(0) = \mathbf{m}^R_{(i|a)}(1)$ and $\mathbf{m}^R_{(i|a)}(0) \neq \mathbf{m}^R_{(i|a)}(1)$. Values that belong to the first class are produced when the values of x_j that minimise the sum $\sum_{j\in\partial a\setminus i} \mathbf{m}^L_{(j|a)}(x_j)$ also satisfy the clause a, which means that the minimum energy achievable in this subtree is not dependent on the value chosen for X_i. In all other cases, values belong to the second class. We know that values $\mathbf{m}^R_{(i|a)}(x_i)$ in this second class have their smallest value for the x_i that satisfies a, and $\mathbf{m}^R_{(i|a)}(0)$ and $\mathbf{m}^R_{(i|a)}(1)$ cannot differ by more than the weight w_a of the clause a.

This means that we can determine the value that X_i should take in order to minimise the energy in that subtree without having to consider all possible values that the message $\mathbf{m}^R_{(i|a)}$ could take. In other words, if $\mathbf{m}^R_{(i|a)}(x_i)$ differs in its value between $x_i = 0$ and $x_i = 1$, then we interpret this as a *warning* that X_i should take the value that satisfies clause a in order to minimise the energy in that subtree. This is an extension of the idea of a warning from Braunstein et al. (2005).

A variable X_i can receive conflicting warnings from its neighbouring factors, which means that X_i is being separately advised to take the value *True* and also the value *False*. If we only consider satisfiable assignments then conflicting warnings from hard clauses will not occur; however, we must still resolve conflicting warnings from soft clauses. We will want to choose the value for X_i that results in the smallest overall energy across the graph. If this value does not satisfy a neighbouring clause a, then i sends a warning to a that it will not be able to satisfy it.

Let us first split the neighbours of variable i into four sets that are a function of a clause a connected to i:

1. $H^u_i(a)$ – the set of hard clauses in F where variable X_i appears with an opposite sign to that which it takes in clause a.

2. $H^s_i(a)$ – the set of hard clauses in F, excluding a, where variable X_i appears with the same sign to that which it takes in clause a.

3. $S^u_i(a)$ – the set of soft clauses in F where variable X_i appears with an opposite sign to that which it takes in clause a.

4. $S^s_i(a)$ – the set of soft clauses in F, excluding a, where variable X_i appears with the same sign to that which it takes in clause a.

Each message $\mathbf{m}^L_{(i|a)}$ is calculated using the equation

$$\mathbf{m}^L_{(i|a)}(x_i) = \sum_{b \in \partial i \setminus a} \mathbf{m}^R_{(i|b)}(x_i) + \kappa_{ia}. \tag{4.50}$$

Without loss of generality, let us assume that clause a contains the literal X_j. Then $\mathbf{m}^L_{(j|a)}(x_j)$ takes its lowest value at $x_j = 0$ – the value that does not satisfy a – if

$$\sum_{b \in S^u_j(a) \cup H^u_j(a)} \mathbf{m}^R_{(j|b)}(1) - \mathbf{m}^R_{(j|b)}(0) > \sum_{b \in S^s_j(a) \cup H^s_j(a)} \mathbf{m}^R_{(j|b)}(0) - \mathbf{m}^R_{(j|b)}(1). \tag{4.51}$$

That is to say, the energy would be increased by a larger amount if X_j took the value that could violate some clauses in $S^u_i(a) \cup H^u_i(a)$ rather than the value that could violate some clauses in $S^s_i(a) \cup H^s_i(a)$.

To make further progress, we will assume that differences between values such as $\mathbf{m}^R_{(j|b)}(0)$ and $\mathbf{m}^R_{(j|b)}(1)$ are exactly equal to the weight w_b of the clause b from which the message is sent. As we have discussed above, these values cannot differ by more than the weight w_b, so this is a worst case assumption. It also results in a loss of accuracy but simplifies the computation of whether a warning is sent from a variable to a clause. Under this scheme, for a given set of warnings that X_j is receiving, it should sum the weights of those clauses that contain the literal X_j and are sending a warning, call this sum w^+, and sum the weights of those clauses that contain the literal $\neg X_j$ and are sending a warning, call this sum w^-. If $w^+ > w^-$, then $X_i = 1$ (*True*) results in a lower total energy across those subtrees, otherwise $X_i = 0$ (*False*) does. This assumption of the worst case can lead us to erroneous conclusions as is illustrated in Example 1.

Example 1. Consider a clause $\{\neg X_i, X_j\}$ represented by the factor a and the message $\mathbf{m}^R_{(i|a)}(x_i)$. Let the clause a have a weight of 10 and consider an incoming message from j which has the values 4 and 5 for $X_j = 0$ and $X_j = 1$, respectively. According to Equation 4.49, this gives values of $\mathbf{m}^R_{(i|a)}(0) = 4 + \kappa_{ai}$ and $\mathbf{m}^R_{(i|a)}(1) = 5 + \kappa_{ai}$. Hence, variable X_i is receiving a warning from clause a suggesting that X_i be set to 0 (*False*) to satisfy a; however, if X_i does not assume such a value, the cost increases by 1 rather than the weight of a, since X_j can change its assignment to satisfy clause a.

Surveys

The above outlines a scheme for computing whether a warning is sent from variable i to clause a. A clause a sends a warning to a neighbouring variable i if a is receiving warnings from all $j \in \partial a \setminus i$. Let us return our attention to Equations 4.47 and 4.48, which show message passing in terms of $Q_{i\to a}(\mathbf{m}^L_{(i|a)})$ and $\hat{Q}_{a\to i}(\mathbf{m}^R_{(i|a)})$. We have outlined why we only need to consider warning messages in order to find a truth assignment of minimum energy. Let us now consider the probability of a warning message being exchanged between variables and factors.

We assume the messages $Q_{i\to a}$ and $\hat{Q}_{a\to i}$ have been normalised. Define $\omega_{i\to a}(1)$ as the probability of a warning being sent from variable i to a and $\hat{\omega}_{a\to i}(1)$ as the probability of a warning being sent from clause a to variable i. $\omega_{i\to a}(0)$ and $\hat{\omega}_{a\to i}(0)$ are the probabilities of those warnings not being sent.

$$\omega_{i\to a}(1) = \sum_{\substack{\mathbf{m}^L_{(i|a)} \text{ such that} \\ \mathbf{m}^L_{(i|a)}(x) < \mathbf{m}^L_{(i|a)}(1-x) \\ \text{and } X_i = x \text{ violates clause } a}} Q_{i\to a}(\mathbf{m}^L_{(i|a)}), \tag{4.52}$$

$$\hat{\omega}_{a\to i}(1) = \sum_{\substack{\mathbf{m}^R_{(i|a)} \text{ such that} \\ \mathbf{m}^R_{(i|a)}(0) \neq \mathbf{m}^R_{(i|a)}(1)}} \hat{Q}_{a\to i}(\mathbf{m}^R_{(i|a)}). \tag{4.53}$$

As a result of normalisation, we can write $\omega_{i\to a}(0) = 1-\omega_{i\to a}(1)$ and $\hat{\omega}_{a\to i}(0) = 1-\hat{\omega}_{a\to i}(1)$. The probability of a warning being sent is referred to as a *survey*.

Since a hard clause carries an infinite weight, according to Equation 4.51, if a warning is received from a hard clause then one side of the inequality will be infinite. Hence, a warning from a clause $b \in \partial i \setminus a$ must be obeyed and determines whether a warning is sent from the variable i to clause a.

The probability of variable X_i receiving a warning from a clause in $H_i^u(a)$, a clause in $H_i^s(a)$, and no clauses in $H_i^u(a) \cup H_i^s(a)$ is proportional to the terms $\mathcal{H}_i^u(a)$, $\mathcal{H}_i^s(a)$ and $\mathcal{H}_i^0(a)$, respectively:

$$\mathcal{H}_i^u(a) = \left[1 - \prod_{b\in H_i^u(a)} (1-\hat{\omega}_{b\to i}(1))\right] \prod_{b\in H_i^s(a)} (1-\hat{\omega}_{b\to i}(1)), \tag{4.54}$$

$$\mathcal{H}_i^s(a) = \left[1 - \prod_{b\in H_i^s(a)} (1-\hat{\omega}_{b\to i}(1))\right] \prod_{b\in H_i^u(a)} (1-\hat{\omega}_{b\to i}(1)), \tag{4.55}$$

$$\mathcal{H}_i^0(a) = \prod_{b\in \mathcal{H}_i^u(a)\cup\mathcal{H}_i^s(a)} (1-\hat{\omega}_{b\to i}(1)). \tag{4.56}$$

The terms are normalised by the sum $\mathcal{H}_i^u(a) + \mathcal{H}_i^s(a) + \mathcal{H}_i^0(a)$ since we stipulate that receiving conflicting warnings from hard clauses is not possible. These equations are the same as the survey propagation equations for the SAT problem (Braunstein et al. 2005).

In the absence of receiving warnings from neighbouring hard clauses a variable that is connected to soft clauses must resolve any conflicting warnings from these clauses. By rewriting Equation 4.51 for only soft clauses,

$$\sum_{b\in S_j^u(a)} \mathbf{m}_{(j|b)}^R(1) - \mathbf{m}_{(j|b)}^R(0) - \sum_{b\in S_j^s(a)} \mathbf{m}_{(j|b)}^R(0) - \mathbf{m}_{(j|b)}^R(1) > 0,$$

we can see that we need to compute the sum h of weights of soft clauses in $S_i^u(a) \cup S_i^s(a)$ that are sending a warning to i, where a warning from a clause $b \in S_i^u(a)$ contributes $+w_b$ and a warning from a clause $c \in S_i^s(a)$ contributes $-w_c$ to the sum. If $h > 0$ then a warning is sent from variable i to clause a. Let $P_{i,a}(h)$ be the probability that the set of incoming warnings to i from clauses $b \in \partial i \setminus a$ is such that this sum is equal to h.

In addition, define the two quantities

$$\mathcal{S}_i^u(a) = \sum_{h=1}^{\infty} P_{i,a}(h), \qquad \mathcal{S}_i^s(a) = \sum_{h=-1}^{-\infty} P_{i,a}(h). \tag{4.57}$$

Therefore, $\mathcal{S}_i^u(a)$ is the probability that a warning is sent from i to a given that there are

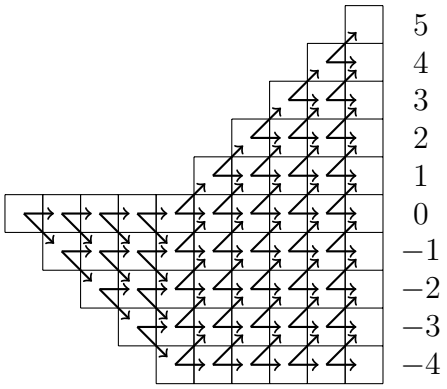


Figure 4.3: Calculating $P_{i,a}(h)$ if all soft clauses have weight 1. For each soft clause, we either receive no warning and move horizontally, or we receive a warning and move up or down depending on the sign that the variable takes in the clause that is sending the warning.

no warnings incoming to i from hard clauses.

We can now write the message passing equations for what we call *weighted partial Max-SAT survey propagation*, or WPSP(y). They are as follows:

$$\omega_{i\to a}(1) = \frac{\mathcal{H}_i^u(a) + \mathcal{H}_i^0(a)\mathcal{S}_i^u(a)}{\mathcal{H}_i^u(a) + \mathcal{H}_i^s(a) + \mathcal{H}_i^0(a)} \tag{4.58}$$

$$\hat{\omega}_{a\to i}(1) = \prod_{j\in\partial a\setminus i} \omega_{i\to a}(1) \tag{4.59}$$

In summary, variable i sends a warning to clause a if it receives a warning from a hard clause to take a value that would not satisfy a or it receives no warnings from hard clauses but a net warning from soft clauses to take a value that would not satisfy a.

4.4.5 Calculating $P_{i,a}(h)$

In order to compute $P_{i,a}(h)$, we do not need to sum over all possible assignments. We can use dynamic programming to construct the distribution. This has a simple form if all soft clauses have weight equal to 1, as would be the case in the unweighted Max-SAT problem (see Figure 4.3). Without loss of generality, we can consider all clauses in $S_i^s(a)$ first and then consider all clauses in $S_i^u(a)$. This is why Figure 4.3 shows only horizontal or downward movements initially and then switches to only horizontal and upwards movements.

Let us consider the term $e^{-yE_a(\mathbf{x}^*(a))}$ for the message $\hat{Q}_{a\to i}(\mathbf{m}^R_{(i|a)})$ in Equation 4.48. As we have already discussed, when $\mathbf{m}^R_{(i|a)}(0) = \mathbf{m}^R_{(i|a)}(1)$, $\mathbf{x}^*(a)$ will satisfy clause a regardless of the value chosen for X_i; in which case, $E_a(\mathbf{x}^*(a)) = 0$ and the term disappears. Otherwise, in the case of $\mathbf{m}^R_{(i|a)}(0) \neq \mathbf{m}^R_{(i|a)}(1)$, we have assumed that the messages differ by w_a. In

Algorithm 5: Calculating $P_{i,a}(h)$

Result: The distribution $P_{i,a}(h)$ of the net sum of weights h of soft clauses in $S_i^s(a) \cup S_i^u(a)$ sending warnings.

1 **begin**
2 $\quad t \leftarrow 1$
3 $\quad \tilde{P}_{i,a}^{(1)}(0) \leftarrow 1$
4 $\quad$ **for** $b \in S_i^s(a)$ **do**
5 $\quad\quad$ **for** $h \in \{x \mid \tilde{P}_{i,a}^{(t)}(x) \neq 0\}$ **do**
6 $\quad\quad\quad \tilde{P}_{i,a}^{(t+1)}(h) \leftarrow \tilde{P}_{i,a}^{(t+1)}(h) + (1 - \hat{\omega}_{b\to i}(1))\tilde{P}_{i,a}^{(t)}(h)$
7 $\quad\quad\quad \tilde{P}_{i,a}^{(t+1)}(h - w_b) \leftarrow \tilde{P}_{i,a}^{(t+1)}(h - w_b) + \hat{\omega}_{b\to i}(1)\tilde{P}_{i,a}^{(t)}(h)$
8 $\quad\quad t \leftarrow t + 1$
9 $\quad$ **for** $b \in S_i^u(a)$ **do**
10 $\quad\quad$ **for** $h \in \{x \mid \tilde{P}_{i,a}^{(t)}(x) \neq 0\}$ **do**
11 $\quad\quad\quad \tilde{P}_{i,a}^{(t+1)}(h) \leftarrow \tilde{P}_{i+1}^{(t+1)}(h) + (1 - \hat{\omega}_{b\to i}(1))\tilde{P}_{i,a}^{(t)}(h)$
12 $\quad\quad\quad \tilde{P}_{i,a}^{(t+1)}(h + w_b) \leftarrow \tilde{P}_{i,a}^{(t+1)}(h + w_b) + \hat{\omega}_{b\to i}(1)\tilde{P}_{i,a}^{(t)}(h)e^{-y\theta(h,w_b)}$
13 $\quad\quad t \leftarrow t + 1$
14 $\quad Z \leftarrow \sum_{h=-\infty}^{\infty} \tilde{P}_{i,a}^{(t)}(h)$
15 $\quad P_{i,a}(h) \leftarrow \frac{1}{Z}\tilde{P}_{i,a}^{(t)}(h)$
16 **end**

this case, the value of $E_a(\mathbf{x}^*(a))$ will depend upon the value assigned to X_i. If X_i's value does not satisfy clause a, then $E_a(\mathbf{x}^*(a))$ equals w_a, otherwise it equals 0 and the term disappears. Therefore, when a warning is being sent, we defer inclusion of the term $e^{-yE_a(\mathbf{x}^*(a))}$ in the calculation of $\hat{\omega}_{a\to i}(1)$ until we know the best assignment for X_i. That is to say, each value h determines the assignment of X_i, so the computation of $P_{i,a}(h)$ includes a term e^{-yw_a} for each clause a that took part in the sum that gave h and had its warning ignored.

Let us now look at how this affects the computation of $P_{i,a}(h)$, which is described in detail in Algorithm 5. Firstly, $\tilde{P}_{i,a}^{(t)}(x)$ is initialised to 1 for $t = 1, x = 0$ and to 0 for all other values of t and x. While we are in the $h < 0$ area, any warning from a clause $b \in S_i^s(a)$ decreases h by w_b (line 7) and if we remain in this area, then the assignment to X_i will eventually be such that clauses in $S_i^s(a)$ are satisfied. Thus, we should not be multiplying by the term $e^{-yE_a(x_i)}$. While we are in the $h < 0$ area, any warning from a clause in $b \in S_i^u(a)$ will increase h by w_b (line 12). If we are still in the $h < 0$ area after considering all clauses in $S_i^u(a)$, then the minimum energy will be achieved by assigning X_i to a value that violates clauses in $S_i^u(a)$. Therefore, when we receive a warning from a clause $b \in S_i^u(a)$ while in the region $h < 0$ we must multiply by e^{-yw_b}.

If we cross the boundary to the $h > 0$ region, then the preferred value for X_i is now the

one that satisfies clauses in $S_i^u(a)$. Then all the terms e^{-yw_b} that we included while we were in the $h < 0$ region no longer originate because of warnings received from clauses $b \in S_i^u(a)$, but instead they are attributed to clauses in $S_i^s(a)$. This is still correct because in order for an upwards movement in the $h < 0$ region to be made, a downward movement must have been made beforehand. While in the $h > 0$ region, we never include the term e^{-yw_b} when a warning is received from a clause $b \in S_i^u(a)$ because this region indicates that the warning clause will eventually be obeyed by the assignment to X_i.

The θ term found in line 12 is introduced to describe what happens when a transition is made from $h < 0$ to $h \geq 0$ by receiving a warning from a clause in $b \in S_i^u(a)$ with a weight w_b. Say we were at $h_1 < 0$ before receiving the warning and at $h_2 = h_1 + w_b \geq 0$ after receiving the warning. Then we should include the term e^{yh_1}. We do not include the full weight e^{-yw_b} because on crossing the boundary we switch the assignment that should be made to X_i to one that will satisfy clauses in $S_i^u(a)$ and so the penalties are now attributed to clauses from $S_i^s(a)$ and e^{yh_1} is the remaining unaccounted for penalty term. Consequently, θ is defined as

$$\theta(h, w) = \begin{cases} 0 & \text{if } h > 0, \\ w & \text{if } h + w < 0, \\ -h & \text{otherwise.} \end{cases} \tag{4.60}$$

Figure 4.4 illustrates an example of the possible paths that lead to different values of h for incoming warnings from three soft clauses.

4.4.6 Finding a fixed-point of the WPSP(y) equations

Before applying the survey propagation technique to a WPMax-SAT formula we preprocess the formula so that it contains no unit hard clauses or pure literals. As described in Section 2.3.2, we do this by performing unit propagation on hard unit clauses and setting pure literals to *True*. We also remove trivially satisfied clauses such as those that contain both a literal ℓ and $\neg\ell$.

The procedure for finding a fixed-point of the WPSP(y) equations is given in Algorithm 6. The procedure starts by initialising all surveys $\{\hat{\omega}_{a\to i}(1)\}_{(i,a)\in E}$ in the graph to random values in the range $[0, 1]$. To avoid repeating calculations, we cache the values $\omega_{i\to a}(1)$ and recompute them from $\{\hat{\omega}_{b\to i}(1)\}_{b\in\partial i\setminus a}$ only when it is possible that they might have changed. Surveys are iteratively updated in place by applying the WPSP(y) equations. In each iteration the order in which clauses are visited is randomised. For each clause that we visit we update each outgoing survey. We repeat this procedure until no survey changes by more than the convergence precision $\epsilon = 0.001$ during a single iteration or the

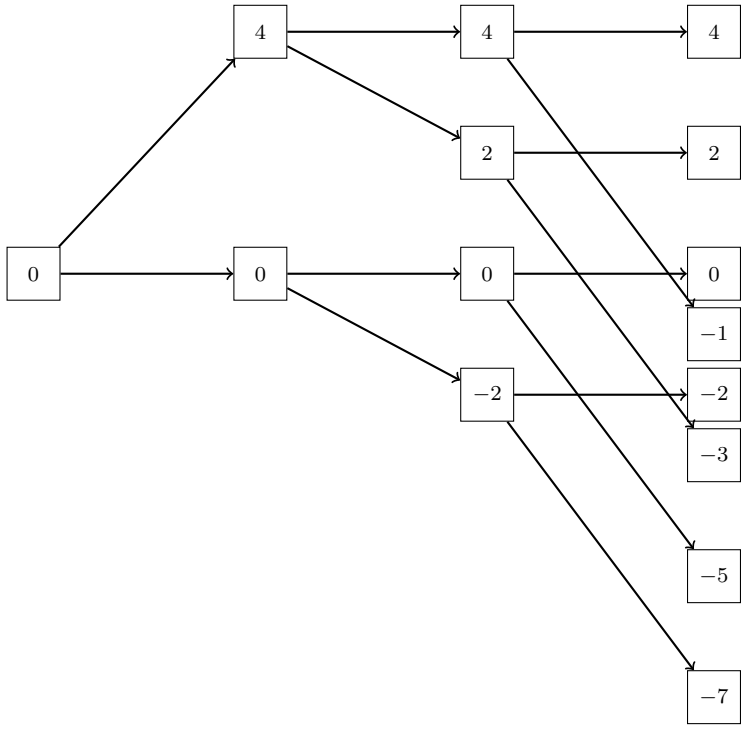


Figure 4.4: An illustration of the different energy levels that are possible from incoming warnings for a variable X_i sending a message to clause a when $S_i^u(a)$ contains a single clause with weight 4 and $S_i^s(a)$ contains two clauses with weight 2 and 5, respectively.

number of iterations exceeds the upper limit t_{lim}. The return value indicates whether the procedure has been successful in finding a fixed point of the equations.

Algorithm 6: Finding a fixed-point of the WPSP(y) equations

begin
  Initialise all surveys $\{\hat{\omega}_{a\to i}(1)\}_{(i,a)\in E}$ to random numbers in the range $[0,1]$.
  for $t \leftarrow 1$ **to** t_{lim} **do**
    Let π be a randomly selected permutation of $\{1,\ldots,m\}$.
    hasChanged $\leftarrow$ **false**
    for $p \leftarrow 1$ **to** m **do**
      for $i \in \partial\pi(p)$ **do**
        $\hat{\omega}'_{\pi(p)\to i}(1) \leftarrow \hat{\mathrm{T}}_{\pi(p)\to i}(\{\omega_{j\to a}(1)\}_{j\in\partial\pi(p)\setminus i})$
        if $\left|\hat{\omega}'_{\pi(p)\to i}(1) - \hat{\omega}_{\pi(p)\to i}(1)\right| > \epsilon$ **then**
          $\hat{\omega}_{\pi(p)\to i}(1) \leftarrow \hat{\omega}'_{\pi(p)\to i}(1)$
          hasChanged $\leftarrow$ **true**
    if $\neg$hasChanged **then return** Success
  return Failure
end

4.4.7 Calculating bias estimates

Once an approximate fixed-point is found, we must use the surveys to decide which variable to assign to next and which value should be chosen for that assignment. Earlier, in Equation 4.22, we gave the form that the max-marginal takes when using the min-sum equations. We noted that, for variable X_i, the max-marginal $\mu_i^{\max}(x_i)$ takes its maximum value at the same x_i that minimises the sum,

$$\sum_{a \in \partial i} \hat{\eta}_{a \to i}(x_i), \tag{4.61}$$

of incoming messages in the min-sum message passing scheme. Deciding how to minimise the above equation is similar to how we decided to minimise Equation 4.50 earlier. Instead of the summation being over $b \in \partial i \setminus a$, it is over $a \in \partial i$.

In the same manner with which we defined the terms in Section 4.4.4 we now define the following:

1. H_i^+ – the set of hard clauses in F that contain the literal X_i.
2. H_i^- – the set of hard clauses in F that contain the literal $\neg X_i$.
3. S_i^+ – the set of soft clauses in F that contain the literal X_i.
4. S_i^- – the set of soft clauses in F that contain the literal $\neg X_i$.

The probability of variable X_i receiving a warning from a clause in H_i^+, a clause in H_i^-, and no clauses in $H_i^+ \cup H_i^-$ is proportional to the terms $\mathcal{H}_i^+$, $\mathcal{H}_i^-$ and $\mathcal{H}_i^0$, respectively:

$$\mathcal{H}_i^+ = \left[1 - \prod_{b \in H_i^+} (1 - \hat{\omega}_{b \to i}(1))\right] \prod_{b \in H_i^-} (1 - \hat{\omega}_{b \to i}(1)), \tag{4.62}$$

$$\mathcal{H}_i^- = \left[1 - \prod_{b \in H_i^-} (1 - \hat{\omega}_{b \to i}(1))\right] \prod_{b \in H_i^+} (1 - \hat{\omega}_{b \to i}(1)), \tag{4.63}$$

$$\mathcal{H}_i^0 = \prod_{b \in H_i^+ \cup H_i^-} (1 - \hat{\omega}_{b \to i}(1)). \tag{4.64}$$

Let $\tilde{P}_i(h)$ be proportional to the probability of variable X_i receiving a set of warnings from soft clauses such that the weight of those clauses sums to h, where the weights of soft clauses found in S_i^+ and S_i^- contribute positively and negatively to the sum, respectively.

The normalised distribution is written as $P_i(h) = \frac{1}{Z_i}\tilde{P}_i(h)$, where $Z_i = \sum_{h=-\infty}^{\infty} \tilde{P}_i(h)$. We define $\mathcal{S}_i^+$ as the probability of receiving a set of warnings from soft clauses that leads to the aforementioned sum of weights equalling a value greater than zero. Similarly, we define $\mathcal{S}_i^-$ to be equal to the probability that the sum is less than zero, as follows:

$$\mathcal{S}_i^+ = \sum_{h=1}^{\infty} P_i(h), \qquad \mathcal{S}_i^- = \sum_{h=-1}^{-\infty} P_i(h). \tag{4.65}$$

$P_i(h)$ can be calculated by a simple modification to Algorithm 5. Line 4 should be changed to iterate over $b \in S_i^-$ and line 9 should be changed to iterate over $b \in S_i^+$.

X_i should take the value *True* if either it receives a warning from a clause in H_i^+ or it receives no warnings from hard clauses and the sum of weights of clauses in S_i^+ that it receives warnings from is greater than the sum of weights of clauses in S_i^- that it receives warnings from. Since these two events are mutually exclusive, we can write them as a sum of two probabilities. The case when X_i should take the value *False* is the same but with the $+$ and $-$ roles switched. We define the probability that X_i should take the values *True* and *False* as $\mathcal{B}_i^+$ and $\mathcal{B}_i^-$, respectively. Assuming that messages from neigbouring nodes are sent independently, $\mathcal{B}_i^+$ and $\mathcal{B}_i^-$ can be written as

$$\mathcal{B}_i^+ = \frac{\mathcal{H}_i^+ + \mathcal{H}_i^0\mathcal{S}_i^+}{\mathcal{H}_i^+ + \mathcal{H}_i^- + \mathcal{H}_i^0}, \qquad \mathcal{B}_i^- = \frac{\mathcal{H}_i^- + \mathcal{H}_i^0\mathcal{S}_i^-}{\mathcal{H}_i^+ + \mathcal{H}_i^- + \mathcal{H}_i^0}. \tag{4.66}$$

$$Bias(i) = \mathcal{B}_i^+ - \mathcal{B}_i^-. \tag{4.67}$$

The bias of variable X_i is equal to *Bias(i)*.

When a variable is selected for assignment, it should be assigned to *True* if $\mathcal{B}_i^+ - \mathcal{B}_i^- > 0$ and *False* otherwise. If no variable's absolute bias exceeds a threshold in between 0 and 1, then this might indicate that further assignments using the bias estimates may not be worthwhile and we should switch to a different search strategy.

Figures 4.5 and 4.6 show the measured biases for all unassigned variables, calculated from four fixed points of the survey propagation equations we obtained by running Algorithm 6 to convergence on problems p18 from Rovers and p12 from DriverLog (see Chapter 5 for information on these problems). One can see that most variables either have a bias of $+1$, -1 or a value close to 0. Also, note that many action variables have a bias of -1. This is not so surprising since we expect only a small number of actions to be executed; however, setting an action variable to *False* makes an early commitment to not execute that action, but does not make any real progress towards forming a plan that achieves a set of goals. Consequently, we decided to pursue a strategy that uses survey propagation to identify variables that should be set to *True*.

Three methods for choosing the next variable to assign to, taken from the survey inspired decimation algorithm (Braunstein et al. 2005), are:

1. Select the variable with the highest absolute bias $|\mathcal{B}_i^+ - \mathcal{B}_i^-|$,
2. Select with uniform probability from all variables that have an absolute bias exceeding a certain threshold.
3. Set multiple variables with a high bias to their suggested value.

Our empirical results show that the biases tend to belong to one of three groups: either equal to $+1$, 0 or -1. This would make the first scheme awkward to implement as an extra criterion would be needed to choose between two variables that had a similar bias. The third scheme is not very conservative as it may be the case that two variables with a high positive bias are not simultaneously true in any complete truth assignment. Thus, it seems preferable to select with uniform probability from the variables highly biased towards a *True* value and it is this scheme that we adopt in our implementation. Variations on this scheme can be made by adjusting the threshold that constitutes a high bias, and it is not clear what is an appropriate value and whether this is problem dependent. We use the same value of 0.3 for our threshold across all problems in our implementation.

We also restricted our selections to fact variables as this appeared to give better results in our preliminary experimentation. This required each CNF file to be accompanied by an annotation file that describes the type of each variable. The WPMax-SAT solver we describe in the following sections uses this file to determine the type of each variable.

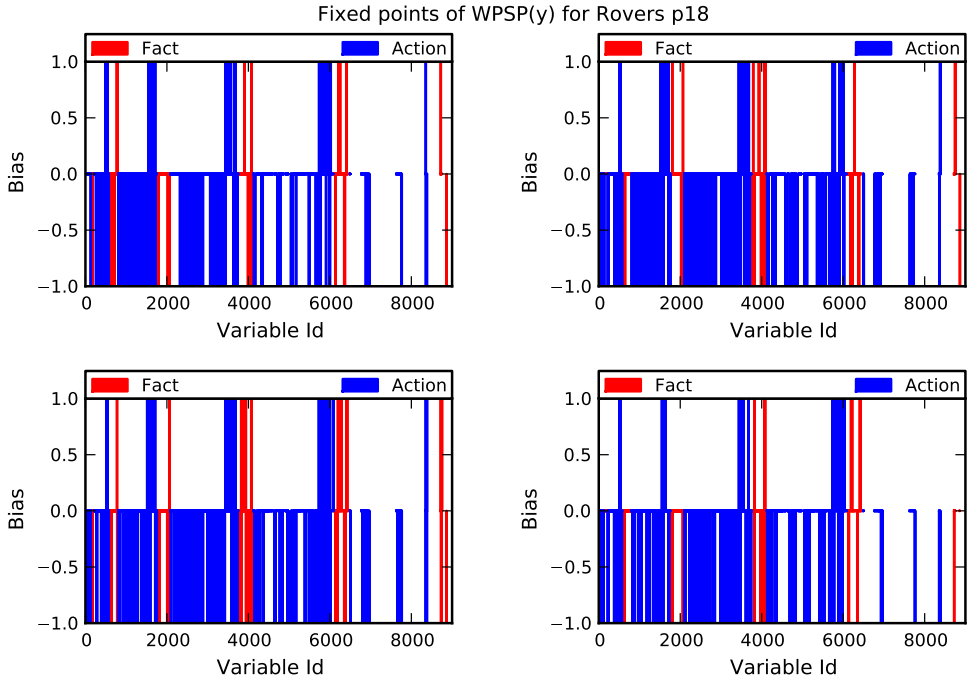


Figure 4.5: Bias of unassigned variables calculated from four fixed points of the survey propagation equations that were found for problem 18 of the Rovers domain.

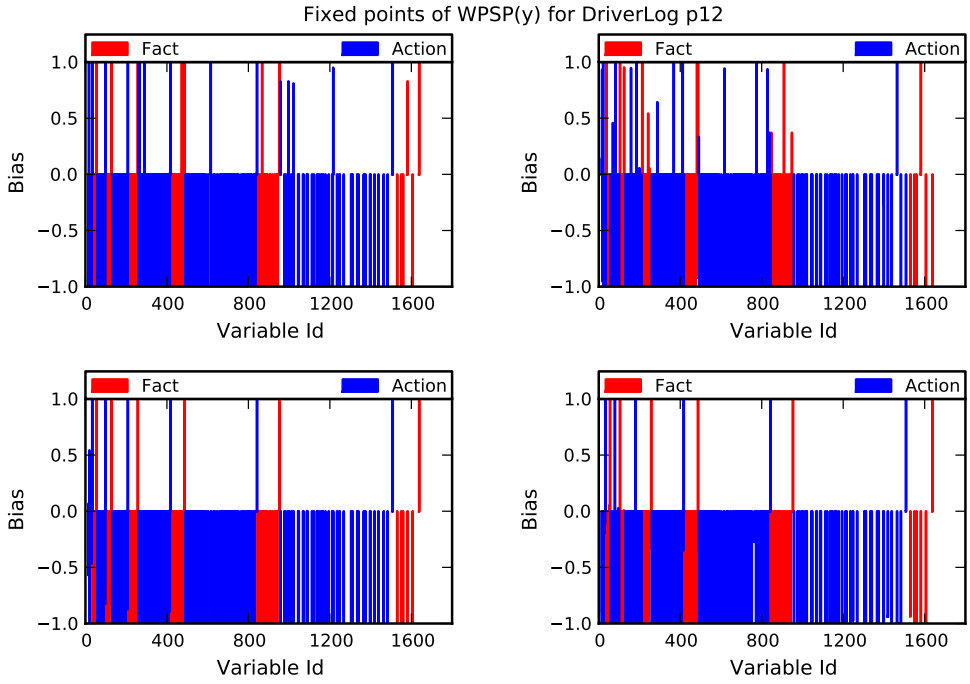


Figure 4.6: Bias of unassigned variables calculated from four fixed points of the survey propagation equations that were found for problem 12 of the DriverLog domain.

4.5 Restart strategy

A *Las Vegas* algorithm is one which always produces the correct answer if it terminates, but has a running time that is a random variable. If the distribution over running times is sufficiently wide then we may be able to find an answer more quickly by repeating runs with limits on the number of steps each run can take. A *restart strategy* $(t_1, t_2, , \dots)$ specifies that for a Las Vegas algorithm A, it should be first run for a maximum of t_1 steps; then A is started again and run for at most t_2 steps, and so forth until a run of A returns an answer. Each time A is run, its running time is randomly determined; by limiting the number of steps in each run we hope to terminate runs that have a long running time and retry to find a run that has a short running time, and by doing this reduce the total expected running time.

4.5.1 Luby strategy

If we know the distribution of running times, that is the probability $p(t)$ of A terminating after exactly t steps, then there is a known optimal restart strategy. However, typically we do not have such knowledge, so we must adopt a restart strategy that works well regardless of the underlying distribution on running times. The Luby strategy fulfills this requirement and can be shown to have an expected running time that is optimal amongst other restart strategies up to a constant factor (Luby et al. 1993).

The Luby strategy of run lengths is

$$(1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, ...).$$

Every element in the strategy is a power of two, starting with 1 for the first element. The sequence of run lengths can be generated by repeated applications of the following rule. Rewrite the sequence as two repetitions of the sequence generated so far, and then set the next element in the sequence to double the value of the largest element found so far.

Alternatively, the above strategy can be precisely described as $(luby(1), \dots, luby(i), \dots)$ where $luby(i)$ is defined as

$$luby(i) = \begin{cases} 2^{k-1} & \text{if } i = 2^k - 1, \\ luby(i - 2^{k-1} + 1) & \text{if } 2^{k-1} \leq i < 2^k - 1. \end{cases} \tag{4.68}$$

Restarts in message passing

Without the limit t_{lim}, Algorithm 6 can be classified as a Las Vegas algorithm. The objective is to find a fixed point of the WPSP(y) equations. Since the messages are

initialised to random values on each run, and the order in which the messages are updated is randomised, the number of iterations before a fixed point is reached is a random variable. Moreover, since we are not guaranteed to find a fixed point, the running time could be infinite.

We do not expect to have information about the distribution of the number of iterations required before reaching a fixed point of message passing. Therefore, it is sensible to adopt a Luby restart strategy to set the value t_{lim} of the maximum number of iterations to perform without finding a fixed point before restarting message passing from a new set of initial messages.

Restarts in SAT solvers

Gomes et al. (1997) showed empirically, for backtracking SAT solvers, that the erratic behaviour of the sample mean and variance, when measuring the number of backtracks before a solution is found, for increasing numbers of trials, were similar to what one might expect from *heavy-tailed* distributions. Informally, a heavy-tailed distribution is one which does not decay exponentially and so carries more probability weight in the tails than a normal distribution, for example. Such distributions can have infinite variance and mean, although for a SAT solver, the number of backtracks is bounded to a number that is exponential in the size of the input, and so we talk about *bounded heavy-tailed* distributions as having a mean or variance that is exponential in the size of the input. This heavy-tailed behaviour could be removed by incorporating randomised restarts into the solution procedure (Gomes et al. 1998). Chen et al. (2001) presented three models of backtracking search and showed how imbalanced trees could lead to heavy-tailed distributions; however, the authors found that balanced trees did not lead to such distributions and so would not benefit from restarts. This leads to the theory that SAT solvers which employ good heuristics lead to imbalanced trees which cause them to suffer from heavy-tailed distributions in their running times. This partly explains the prevalence of randomised restart strategies in successful SAT solvers.

4.6 Incorporating WPSP(y) into a WPMax-Sat solver

We now present a complete and optimal solver for weighted partial maximum satisfiability problems, based upon the MiniMaxSat solver (Heras and Larrosa 2006), that incorporates survey propagation as a variable/value selection heuristic. The MiniMaxSat solver is itself built upon MiniSat+ (Eén and Sörensson 2006). We build our implementation of MiniMaxSat on top of MiniSat2.2, which is a minor upgrade of the 2008 SAT-Race 'CNF sequential' track winner MiniSat2.1 (Sörensson and Eén 2009).

Survey propagation has been incorporated into a SAT solver in the VARSAT system (Hsu and McIlraith 2009). Our work is different to VARSAT in that we handle the optimisation variant incoporating the WPSP(y) equations, which we have developed, into a WPMax-SAT solver. We also make some different design choices to VARSAT; for example, VARSAT stops using survey propagation once its bias estimates do not exceed an activation threshold. We found that on finding such a fixed point, restarting the message passing procedure might lead to a different fixed point which did have a variable with a sufficiently high bias. Like VARSAT, we found that performing message passing is computationally intensive; hence, we limit its use to the initial decision levels.

4.6.1 MiniMaxSat's main search routine

Algorithm 7: Max-Sat main search routine

Result: the pair (b, u) where b is *True* iff the problem is satisfiable and u is the cost of the best solution found.

```
1  begin
2      while true do
3          C ← Propagate()
4          if C is HardConflict then
5              if decisionLevel = 0 then return (false,∞)
6              level ← Analyse(C)
7              Backjump(level)
8          else if C is SoftConflict then
9              success ← ChronologicalBacktrack()
10             if not success then return (true,upperBound)
11         else
12             ℓ ← PickBranchLiteral()
13             if ℓ is Undef then
                   // Found a model
14                 if lowerBound < upperBound then // If we've found a better solution
15                     upperBound ← lowerBound
16                 if upperBound = 0 then return (true,0)
17                 success ← ChronologicalBacktrack()
18                 if success then return (true,upperBound)
19             else
20                 NewDecisionLevel(ℓ)
21 end
```

In line 3 unit propagation is performed: any hard clauses that have become unit have their remaining literal added to the propagation queue, any soft clauses that have become

unit have their cost added to UnitWeight(ℓ). This propagation procedure is repeated until either a conflict is encountered and returned or there are no remaining literals in the propagation queue and no conflict is returned. The algorithm then procedes based on the type of returned conflict. Hard conflicts correspond to a hard clause becoming empty. Soft conflicts correspond to the lower bound exceeding the current upper bound.

If a hard conflict is encountered and the algorithm is at decision level 0, then this indicates that the problem has a subset of hard clauses that is unsatisfiable. Otherwise, in line 6 the conflict is analysed and the decision level to backtrack to is returned.

If a soft conflict is encountered then we attempt to perform chronological backtracking in line 9, which returns *True* if and only if the attempt was successful. Chronological backtracking unwinds the propagation queue to the first decision literal for which we have not tried its alternative value. If the attempt failed then we have exhausted all possible assignments and can return the best solution found as the solution with optimal cost.

If unit propagation did not encounter any conflicts, then we choose the next literal to branch on as in line 12. If the literal ℓ returned is undefined then this means that there are no remaining unassigned variables and we have arrived at a solution. The current lower bound is therefore an exact value for the cost of the solution. If the new solution has a lower cost than any other we have found, we update the upper bound to the current cost and save the solution. If the cost is 0 then we know it is optimal because we only allow non-negative weights for clauses, and so we return that solution. Otherwise, we attempt to chronologically backtrack to find other solutions. If this attempt fails then we have exhausted all possible assignments and can return the best solution so far as the optimal one.

Finally, if the literal ℓ was not undefined, we create a new decision level and add ℓ to the propagtion queue in preparation for the next iteration of the loop.

4.6.2 Lower bounding in MiniMaxSat

The first step in improving the lower bound in MiniMaxSat is to identify unit soft clauses $(\{\ell\}, u)$ and $(\{\neg\ell\}, v)$ that are present in the formula for any literal ℓ. Since either ℓ or $\neg\ell$ must be false in a total assignment, any total assignment will incur a cost of at least $m = \min(u, v)$ as a result of those two clauses being present in the formula. Therefore, those clauses can be changed to $(\{\ell\}, u - m)$ and $(\{\neg\ell\}, v - m)$, and the lower bound can be updated as $\mathsf{lowerBound} \leftarrow \mathsf{lowerBound} + m$.

The second technique to improve the lower bound is to perform a procedure called *Simulated Unit Propagation*. This assumes all soft clauses are in fact hard and then iterates

unit propagation under this assumption. If the empty clause is derived, then this indicates that a subset of hard and soft clauses are together unsatisfiable. A resolution refutation tree is built by unwinding the propagation queue. Each literal that is present on the propagation queue keeps track of the original clause that became unit and hence is the reason for that literal being added to the propagation queue. Start with the original clause C_1 that has become empty. The first literal ℓ_1 whose negation is present in C_1 has its reason clause resolved with C_1 to give a clause C_2. We then keep unwinding the propagation queue until a literal ℓ_2 is found whose negation is in C_2, and then the reason clause for ℓ_2 is resolved with C_2 to give a clause C_3. This process is continued until the empty clause is derived[1]. When the empty clause is derived, we are left with a list of clauses $C_1, \ldots, C_n$, for some n, that form a resolution refutation tree. We know that the clauses in this tree cannot be simultaneously satisfied; moreover, all hard clauses must be satisfied. Therefore, we can identify the minimum weight m of soft clauses in the tree and subtract that from all soft clauses in the tree while adding m to lowerBound.

MiniMaxSat can also perform Max-Sat resolution based lower bounding, which we do not implement. This applies a Max-SAT resolution transformation rule between clauses in the resolution refutation tree. Eventually, this results in a soft empty clause being derived with weight equal to the minimum weight amongst all clauses in the tree. This procedure transforms the formula and must be undone on backtracking. MiniMaxSat heuristically selects between Max-SAT resolution and substraction based lower bounding while our implementation uses substraction based lower bounding exclusively.

We do not implement a procedure called soft probing, which is used as a preprocessing step in MiniMaxSat. For each literal ℓ it assumes that ℓ is true and performs simulated unit propagation to identify whether this leads to an empty clause. If an empty clause is encountered, then either the resolution refutation tree consists entirely of hard clauses, in which case we can infer that $\neg\ell$ must be true, or we can take the minimum weight m of the soft clauses in the refutation tree and add the unit soft clause $(\{\neg\ell\}, m)$ at the same time as subtracting m from the weights of soft clauses in the tree. This can lead to improvements in the lower bound.

4.6.3 Selecting the branch literal

In line 12 of Algorithm 7 the function `PickBranchLiteral()` returns the next literal to branch upon. We now review the existing branching strategies of VSIDS, Jeroslow-Wang and weighted Jeroslow-Wang which are used in MiniMaxSat. We then discuss our implementation of `PickBranchLiteral()` that is different to MiniMaxSat in that

[1]The reason clause for a decision literal ℓ is taken to be the hard unit clause $\{\ell\}$

it incorporates a different strategy for the initial branching decisions and then reverts to that used by MiniMaxSat for the remaining branching decisions.

Marques-Silva (1999) argues that while the choice of branching strategy is important to the success of a DPLL propositional SAT solver, more significant gains are achieved by applying techniques such as clause learning and non-chronological backjumping to prune the search space. As we have said before, in a solver such as MiniMaxSat the branching strategy affects the ability to prune through lower bounding; so it is possible that the importance of the choice of branching strategy is increased for these types of algorithms.

VSIDS heuristic

The Variable State Independent Decaying Sum (VSIDS) branching heuristic (Moskewicz et al. 2001) favours choosing literals that have recently been involved in clause learning. The rationale is that solvers encounter and learn many hard conflicts during search on hard problems and decisions made by the solver should respect the most recent of these conflicts.

The heuristic is computed as follows. For each literal in the formula, an activity value is stored, which is initialised to 0. Each time a clause is added to the formula via conflict learning, each literal in that clause has its activity incremented by 1. The unassigned literal with the highest activity is selected to be assigned to *True* next. Periodically all activities are divided by a constant, which corresponds to the decaying part of the algorithm. This allows past activity to be forgotten and an emphasis to be placed on recently learned clauses.

The VSIDS-like heuristic adopted in MiniSat+ records an activity for each variable rather than each literal. If a variable v appears in the conflict clause, either as v or $\neg v$, then its activity is boosted by 1. The next decision literal chosen is the unassigned variable v with the highest activity. The literal v is then set to *True*. This is the default setup, but there are alternatives to this strategy such as randomly choosing between setting v or $\neg v$ to *True*.

Jeroslow-Wang heuristic

Let F be any propositional formula written in CNF, the Jeroslow-Wang heuristic (Jeroslow and Wang 1990) of literal ℓ is defined as

$$J(\ell) = \sum_{\substack{C \in F \\ \text{s.t. } \ell \in C}} 2^{-|C|}. \tag{4.69}$$

The rational for this expression is that for a formula with n variables, each clause of length p rules out 2^{n-p} of the 2^n possible truth assignments; of course, for multiple clauses, the assignments that are ruled out might intersect. Thus, $2^n J(\ell)$ is an upper bound to the number of truth assignments that are ruled out by the clauses that ℓ appears in. Setting ℓ to *True* satisfies those constraints and hopefully allows greater flexibility in the remaining 2^{n-1} assignments. Thus, the one-sided Jeroslow-Wang heuristic suggests that we set the currently unassigned literal ℓ that has the largest $J(\ell)$ value. A two-sided version of the Jeroslow-Wang heuristic selects the variable v that has maximum $J(v) + J(\neg v)$ over all other variables and sets v to *True* if $J(v) > J(\neg v)$ and *False* otherwise.

Weighted Jeroslow-Wang heuristic

A weighted version of the Jeroslow-Wang heuristic applicable to literals ℓ from a CNF formula F is given by Heras et al. (2008) as

$$J_w(\ell) = \sum_{\substack{(C,w)\in F \\ \text{s.t. } \ell \in C}} 2^{-|C|} w. \tag{4.70}$$

When $w = \top$, a value which is greater than the sum of weights of all soft clauses is used. This is provided by the DIMACS format but is set to one more than the sum of weights of all soft clauses in our generated problems. If a weight is large then this boosts the importance of satisfying that clause.

The selection algorithm

Our modification to `PickBranchLiteral()` is shown in Algorithm 8. In Section 4.6.4 we will discuss three strategies for selecting a decision variable: WPSP(y), Rand and Basic. The performance of these strategies are compared in Chapter 5. First, we will describe how these strategies are used in the solver.

The chosen strategy is assigned to selectionStrategy. The method `selectLiteral()` returns a decision variable according to the chosen strategy. For the first seven decision levels we attempt to select a literal using selectionStrategy. If the method `selectLiteral()` fails to select a variable in line 3, it will return Undef. In this case, `PickBranchLiteral()` returns a literal to branch on according to MiniMaxSat's normal routine described in lines 5–22.

Lines 5–22 choose between two heuristic methods: if there is an unassigned variable v that appears as a literal $\neg v$ in one unsatisfied hard clause and v in another unsatisfied hard

clause then the VSIDS-like heuristic is used, otherwise the one-sided[2] weighted Jeroslow-Wang heuristic is used. The function useJeroslow() in line 6 performs this test and returns *True* if and only if the weighted Jeroslow-Wang heuristic should be used according to the test above. Note that if this test returns *True*, then we have essentially found an assignment that satisfies all hard clauses, since we can set each remaining unassigned variable to a value that satisfies the clauses it appears in, and our only remaining consideration is how to minimise the cost of the total assignment.

The original MiniMaxSat system uses the two-sided weighted Jeroslow-Wang heuristic, which orders variables by $J_w(\ell) + J_w(\neg\ell)$; however, in our encodings all action variables only occur in their negated form in soft clauses so there is no difference between the two-sided and one-sided variants in those cases, whereas goal variables have both unnegated and negated occurrences in soft clauses. Thus, we decided to use the one-sided variant to reduce the effect of this bias. We expect most choices to be made with the the VSIDS-like heuristic, especially early on in the search, since the condition to use the weighted Jeroslow-Wang heuristic will fail until an assignment that satisfies all hard clauses is found.

Both heaps wjeroMaxHeap and vsidsMaxHeap are reconstructed before the procedure in Algorithm 8 is called. The heap wjeroMaxHeap is structured so that the top element is the variable v such that for either $\ell = v$ or $\ell = \neg v$, $J_w(\ell)$ has maximum value amongst all other variables in the formula. If the variable returned by the heap in line 10 is already assigned a value then we continue through the loop to select another variable from the heap. If at a particular iteration we find that the heap is empty, as is tested in line 8, then there can be no remaning unassigned variables to choose from so the procedure returns a special symbol Undef in line 22 to indicate this.

Similarly, the heap vsidsMaxHeap is structured so that the top element is the variable v with highest activity. Again, we loop through lines 16 to 21 until either an unassigned variable is popped from the heap, in which case we return this variable as an unnegated literal, or vsidsMaxHeap becomes empty, in which case we return Undef to indicate that a total assignment has been found.

4.6.4 Selection strategies

In line 3 of Algorithm 8, a selection strategy is used to choose the next literal to branch upon. We now outline three different strategies called WPSP(y), Rand and Basic, that we compare in our experimental results in Chapter 5. The WPSP(y) strategy uses survey propagation to identify an unassigned fact variable to set to *True*; the Rand strategy

[2]The original MiniMaxSat system uses the two-sided weighted Jeroslow-Wang heuristic.

Algorithm 8: PickBranchLiteral function

```
   Result: the literal to branch on
 1 begin
 2     if decisionLevel < 8 then
 3         ℓ ← selectionStrategy.selectLiteral()
 4         if ℓ ≠ Undef then return ℓ
       // Otherwise, choice is by VSIDS or weighted Jeroslow-Wang heuristic.
 5     v ← Undef
 6     if useJeroslow() then
 7         while v = Undef do
 8             if wjeroMaxHeap.isEmpty() then return Undef
 9             else
10                 v ← wjeroMaxHeap.removeMax()
11                 if ¬isAssigned(v) then
12                     if J_w(v) > J_w(¬v) then return v
13                     else return ¬v
14                 else v ← Undef
15     else
16         while v = Undef do
17             if vsidsMaxHeap.isEmpty() then return Undef
18             else
19                 v ← vsidsMaxHeap.removeMax()
20                 if ¬isAssigned(v) then return v
21                 else v ← Undef
22     return Undef
23 end
```

sets a randomly chosen unassigned fact variable to *True*; and the Basic strategy uses the normal procedure outlined in lines 5–22 of Algorithm 8.

The WPSP(y) selection strategy

This strategy is described in Algorithm 9. It attempts to run survey propagation until convergence is reached. If this is successful, it uses the bias estimates obtained from the fixed point messages to decide which literal to return. Our strategy only selects variables that encode facts from the planning graph.

The function `RunSP`, which is called in line 4, takes a single argument specifying the maximum number of iterations of message passing that can be performed before returning. t_{max} is held fixed throughout a run, but $luby(i)$ changes with each attempt according to the Luby strategy defined in Equation 4.68. This allows an increase in the maximum

Algorithm 9: Survey propagation high bias selection strategy.

```
1  begin
2      state ← Unconverged
3      for i ← 1 to 10 do
4          state ← RunSP(luby(i) × t_max)
5          if state = Converged then break
6      if state = Converged then
7          stack ← ∅
8          for v ← 1 to N do
9              if Fact(v) and Bias(v) > 0.3 and ¬Assigned(v) then stack.push(v)
10         k ← Random(0,stack.size())
11         if k ≥ 0 then return stack.getElement(k)
12     return Undef
13 end
```

number of iterations if we repeatedly fail to reach convergence. After ten attempts to find a fixed point of message passing, the strategy gives up and returns Undef to indicate that a variable has not been selected.

If convergence does occur, we iterate through each variable in line 8 and check if the following three conditions hold: the variable encodes a fact from the planning graph, it has a high bias (> 0.3) and is not currently assigned a value. If all three of these criteria are met, then that variable is added to a stack. In lines 10 and 11 we select randomly from amongst the variables that have been accumulated on stack. If there are no unassigned variables that meet the above criteria, the function returns Undef to indicate that a variable has not been selected. Note that if a variable is selected then this variable's value will be initially set to *True*.

The Rand selection strategy

This strategy is described in Algorithm 10. It selects uniformly from all unassigned variables that encode facts from the planning graph. If there are no remaining unassigned variables that correspond to facts in the planning graph, then the method returns Undef to indicate this. Note that if a variable is returned, it will initially be set to *True*.

A random selection strategy is not necessarily as bad as one first might suppose it to be. If it is combined with an aggressive restart policy, then it has the potential to abort early mistakes. For propositional satisfiability, Marques-Silva (1999) compared a random variable branching heuristic with other heuristics that selected based on certain problem statistics. For some problem domains, a random selection strategy outperformed all other heuristics, including one-sided and two-sided Jeroslow-Wang heuristics.

Algorithm 10: Random selection strategy.

```
1 begin
2 |  stack ← ∅
3 |  for v ← 1 to N do
4 |  |  if Fact(v) and ¬Assigned(v) then stack.push(v)
5 |  k ← Random(0,stack.size())
6 |  if k ≥ 0 then return stack.getElement(k)
7 |  return Undef
8 end
```

The Basic selection strategy

This strategy uses the normal procedure outlined in lines 5–22 of Algorithm 8. Hence, there is no change between the branching literal selection used in the first seven decision levels to that used for the remaining decision literals. This strategy is not randomised; however, this does not imply that there is no difference in the search path taken after a restart compared to the path taken before the restart. This is because a solution that is found before the restart could lower the upper bound which might allow pruning to occur after the restart where previously it could not have been done because the upper bound was higher. This would result in a different search path after the restart.

4.7 Summary

We have introduced and presented an extension of the SP(y) algorithm that can be applied to WPMax-SAT problems. Our extension handles the partial nature of the formula: it allows the specification of hard clauses that must be satisfied and soft clauses that describe an optimisation function. We allow soft clauses to have arbitrary positive integer weights whereas SP(y) restricts soft clauses to have a weight equal to 1 or 0. We refer to our equations as the WPSP(y) equations.

We also presented a method for incorporating bias estimates derived from a fixed point of the WPSP(y) equations in a WPMax-SAT solver based upon the architecture of MiniMaxSat. Our modifications preserve the optimal and complete properties of MiniMaxSat. We will use this system to investigate the effects of using such bias estimates to provide heuristic guidance in the solver. The theory predicts that if the approximations are reasonable, we can use these bias estimates to direct the search towards lower cost solutions.

Chapter 5

Results

In this chapter we present our experimental results. These are divided into two main sections. In Section 5.2 we study the performance of a general purpose WPMax-SAT solver when applied to planning problems represented in SAT using the encoding scheme presented in Chapter 3. This is compared against a general purpose mixed integer programming solver applied to an integer programming encoding devised by Do et al. (2007).

In Section 5.3 we evaluate how successful the WPSP(y) equations are in providing heuristic guidance within a WPMax-SAT solver by comparing the strategies described in Chapter 4. We then present results describing an approximate version of the WPSP(y) strategy that may be more suitable in practice.

5.1 Problem domains

Throughout this chapter, we use benchmark problems, taken from previous International Planning Competitions, as a starting point for our evaluation. We modified these problems to contain action costs and descriptions of goal utility dependencies, which are not present in past competitions. Below are brief descriptions of the domains that we used in our experimental analyses.

Depot

The Depot domain consists of crates, trucks, pallets, hoists and places. Crates can be lifted and dropped by hoists. When a crate is being held by a hoist, it can be loaded into a truck or released onto another crate or pallet. Hoists can also be used to unload crates from trucks. In order to execute these actions the relevant truck, crates, hoists and pallets must be co-located. Finally, trucks can transport crates between places.

DriverLog

The DriverLog domain consists of drivers, trucks, packages and locations. Objects must be loaded into trucks, which are then driven to locations where there exists a path between the two locations. Objects can then be unloaded from trucks at their target destinations. Drivers may need to walk from their current location to one where a truck is, and they can only walk between locations where there is a path connecting them.

ZenoTravel

The ZenoTravel domain consists of people, aircraft, cities and fuel-levels associated with each aircraft. There are two modes of flying: one which is slower but consumes less fuel; the other is faster but consumes more fuel. People can board and debark from aircraft at their location. Aircraft can transport people on board between cities. Aircraft must not run out of fuel though, so there is also a refueling action.

Rovers

The Rovers domain consists of vehicles that can navigate between waypoints; these vehicles are variously equipped with aparatus necessary to perform activities such as capturing images or taking rock or soil samples. The results of these analyses must then be communicated to the lander.

Truck

The Truck domain is another logistics domain where packages need to be moved between locations by trucks. The loading space in a truck is divided into areas, and packages can only be loaded onto an area if all areas between it and the truck door are empty. Some packages must be delivered to locations by a deadline.

Pathways

The Pathways domain is based on biochemical pathways from the field of molecular biology. Reactions that are likely to occur in the pathway are modelled by an action with preconditions representing reactants and effects that denote products of the reaction.

5.2 Goal Utility Dependencies

In this section we report experimental results concerning the use of the general purpose WPMax-SAT solver, MiniMaxSat1.0, to solve STRIPS based planning problems with goal utility dependencies. We compare this strategy with an integer programming based approach.

5.2.1 Experimental setup

At the time of writing, the iPUD system was the only other compilation-based approach to STRIPS planning with goal utility dependencies described in the literature. We compared our system MSatplan with iPUD using a *one state change* (1SC) encoding over a collection of problems derived from past International Planning Competition (IPC) benchmarks: *DriverLog*, *Depot*, *ZenoTravel* and *Rovers* from IPC3; and *Truck* and *Pathways* from IPC4. These benchmark problems did not have utility functions exhibiting goal utility dependencies attached to them; therefore, it was necessary to generate our own, which meant that we had to choose appropriate values for action costs that would be low enough to allow some goals to be achieved but high enough to introduce trade-offs. We wrote a Java program to parse untyped STRIPS problems and attach randomly generated action costs and utility functions over goals. This process is described as follows. For each action in a domain, a cost is generated randomly according to a discrete uniform distribution over the values $\{x \in \mathbb{N} \mid 1 \leq x \leq 30\}$. For each problem, a random utility function is generated in two stages. Firstly, a DAG with a restriction on heuristic induced width is randomly generated according to a method that is used to generate random Bayesian networks (Ide et al. 2004). Given this DAG, a conditional preference table (CPT) is generated for each node. For each truth assignment T for a node X and its parents, if $T[X]$ is false then an entry of 0 is made in the CPT for that truth assignment; otherwise, a value is randomly generated according to a discrete uniform distribution over the values $\{x \in \mathbb{N} \mid 100 \leq x \leq 200\}$, and this is entered in the CPT for the truth assignment T. The numbers in these ranges are somewhat arbitrary; they were selected by experimenting to find values that tended to allow valid plans of increasing net benefit as the makespan was increased from 1 to 10. If the action costs are too high or the utilities too low it precludes the existence of any nonempty plan with positive net benefit; the optimal plan would be empty in these cases.

We do not use the original iPUD system because it uses a commercial linear program solver called CPLEX 10.0[1], which we did not have access to; instead, we implemented

[1] http://www.ilog.com/products/cplex

iPUD's encoding scheme by extending the 1SC encoding[2] of IPPlan (van den Briel et al. 2008) and modifying it to use a mature open-source mixed integer program solver SYMPHONY 5.1 (Ralphs and Guzelsoy 2005). We refer to this implementation as iPUD/SYM to avoid ambiguity. We believe this remains a reasonably fair test since the algorithms for SYMPHONY are published and open for inspection. SYMPHONY uses a linear programming solver, CLP, that employs many of the same methods as CPLEX, such as simplex and interior point methods. MSatplan uses a non-commercial WPMax-SAT solver, MiniMaxSat 1.0 (Heras et al. 2008) to solve its encodings. MiniMaxSat was chosen because of its strong performance across domains in the Max-SAT-2008 evaluation.

Since all goals are soft, a (possibly empty) plan exists at every makespan, and consequently it is not very interesting to search for the plan with smallest makespan. Instead, we split each problem up into subproblems parameterized by a makespan variable d. Each subproblem is to find a plan of makespan d that solves the original problem with optimal net benefit over all other plans of makespan d. For each problem in a domain benchmark, we derive subproblems for $d = 1, \ldots, 10$. Each planning system is given 30 minutes to solve each subproblem, and is aborted if it fails to do so.

In summary, each run of our experiment consists of generating a random utility tree and action costs for each problem in each domain, splitting each problem into 10 subproblems and attempting to solve each subproblem with both solvers. We conduct five runs using randomly generated utility trees and action costs each time. Where appropriate, we compute the sample mean and sample variance over these different runs to investigate how the utility tree and action costs affect solution times. All experiments are conducted on a Linux machine with an Intel 2.4 GHz quad core CPU (although neither program is multithreaded) and 2 GB of memory; however, we limit the memory resource available to each program to 1.5 GB to reduce paging.

When examining these results, it should be noted that iPUD/SYM uses a translation step, taken from the *Fast Downward Planner* (Helmert 2006), that converts PDDL2.2 (Edelkamp and Hoffmann 2004) files to the SAS$^+$ formalism (Bäckström and Nebel 1995), which is used to represent multi-valued planning tasks. Inspection of several experiments revealed that for problems 06–30 from the Pathways domains, this translation step almost always failed to complete within 30 minutes. We found that this was also the case for the original translation tool applied to the original PDDL2.2 files taken from the Pathways domain from IPC-4; the reason for this failure remains unclear.

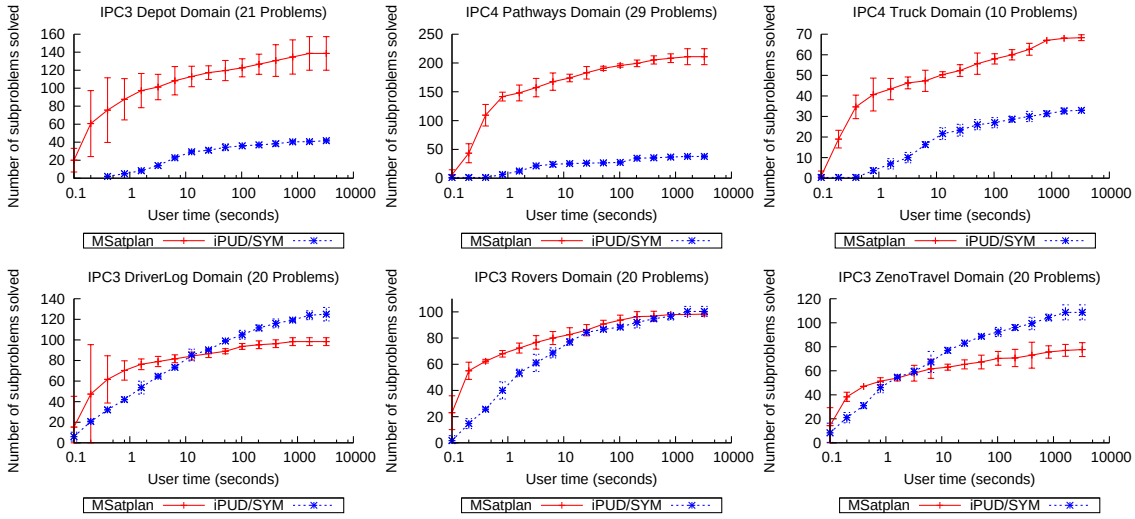


Figure 5.1: Comparison of MSATPLAN with iPUD/SYM on the number of subproblems solved as each system is given more time. Each point shows the mean number of subproblems solved within the corresponding time limit. Error bars show $\pm\frac{2.776}{\sqrt{5}}\sigma$, calculated over five runs. A lack of an error bar indicates that there was no observed variance in that measurement.

5.2.2 Results

For a collection of time limits, the data collected for each one of the five runs were used to calculate the number of subproblems solved within each time limit. The time limits were chosen to achieve equally spaced points on a logarithmic scale. The results for each time limit were then averaged over the five runs. Figure 5.1 shows a plot of the results. MSATPLAN shows a clear advantage over the 1SC encoding of iPUD/SYM for the Depot and Truck domains by solving $235 \pm 24\%$ and $109 \pm 8\%$ more subproblems respectively. MSATPLAN also performs very well on the Pathways domain; however, comparing this with iPUD/SYM is not possible because of a problem with the translation step as described above. iPUD/SYM solves $19 \pm 15\%$, $38 \pm 10\%$, and 3 ± 3 % more problems than MSATPLAN for the DriverLog, ZenoTravel and Rovers domains respectively. The larger variance in the results of MSATPLAN is in part due to the use of randomness in the local search solver, Walksat, that is used by MiniMaxSat to search for an initial satisfiable assignment. We believe that this is not much cause for concern as the time spent proving optimality typically dominates the running time.

How the two systems compare over the range of fixed makespans can be seen in Figure 5.2. The percentage of subproblems that are solved drops off with an increase in makespan, as one would expect; however, the gradient of this decrease differs quite substantially between domains. For the Depot and DriverLog domains, the decrease is smooth with a reasonably

[2]We use the 1SC encoding instead of the G1SC one used in Do et al.'s original paper because it allows the same amount of parallelism as the SAT-based encoding that MSATPLAN extends.

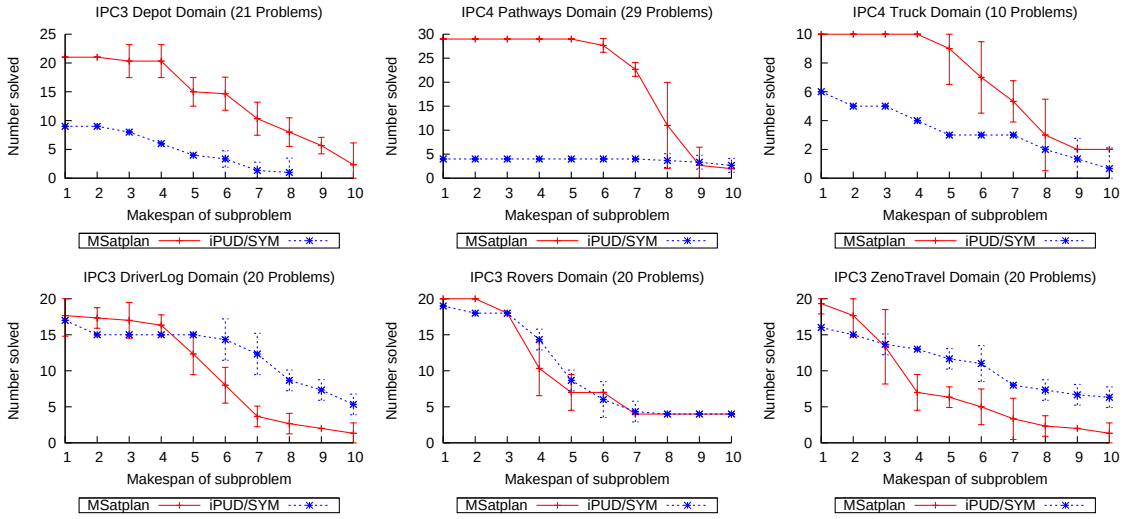


Figure 5.2: Comparison of the successful search depth between MSATPLAN and iPUD/SYM over six problem domains taken from IPC-3 and IPC-4. Each point shows the mean number of subproblems, of a particular makespan, that were solved by each system. Error bars show $\pm\frac{2.776}{\sqrt{5}}\sigma$, which is calculated over the five runs. A lack of an error bar indicates that there was no observed variance in that measurement.

consistent gradient for both systems. For the Truck and Pathways domains, there is very little decrease initially and almost all the decrease occurs within 2–4 makespans for MSATPLAN.

How each system performed on an individual problem can be seen in Figure 5.3. It is somewhat surprising that MSATPLAN's performance is maintained across the Pathways domain. For other domains the performance tends to degrade as the problem number increases, since these are considered harder problems. It is also surprising that MSATPLAN solves a large number of subproblems from the higher problem numbers in the DriverLog domain, but not for problem numbers 04–10, which would normally be considered easier to solve.

5.2.3 Discussion

The crossover that can be observed in the DriverLog and ZenoTravel domains in Figure 5.1 is probably due to the SAS$^+$ translation step that compresses the state representation (Edelkamp and Helmert 1999). In certain domains, this may help the iPUD/SYM solver to solve larger problems than MSATPLAN. Figure 5.2 shows that iPUD/SYM is able to solve more subproblems with makespans ≥ 5 for the DriverLog and ZenoTravel domains, which supports this idea. It would be interesting to integrate the SAS$^+$ translation step into a SAT-based planning system to try to reduce encoding sizes to test this hypothesis.

In Section 4.6.1 we discussed how MINIMAXSAT finds an optimal solution to a WPMax-

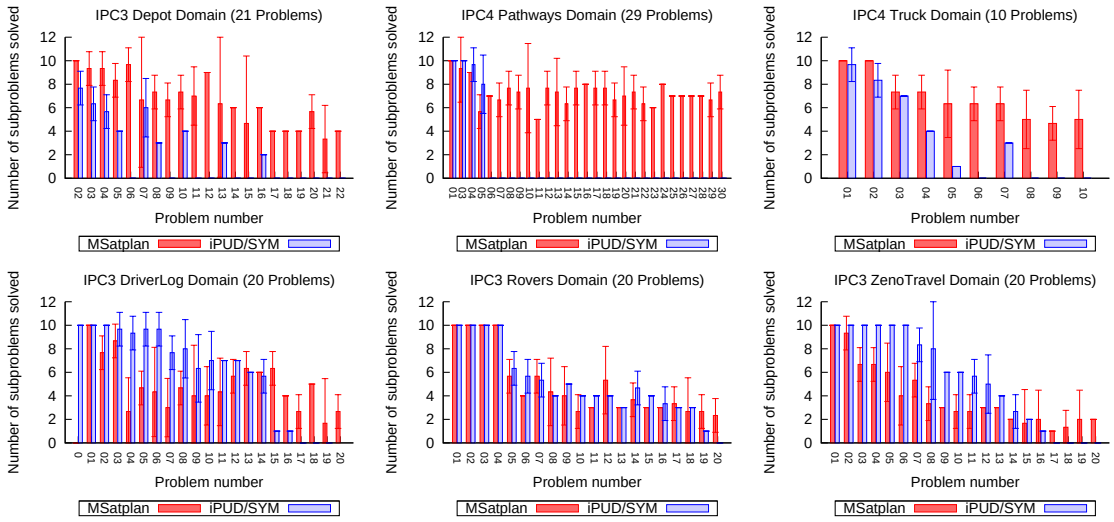


Figure 5.3: Comparison of MSATPLAN with iPUD/SYM on each problem over a total of six domains from IPC-3 and IPC-4. The height of a bar indicates the mean number of subproblems solved by that system for that problem. Error bars indicate $\pm\frac{2.776}{\sqrt{5}}\sigma$, calculated over five runs. A lack of an error bar for a column indicates that there was no observed variance in that measurement.

SAT formula using a branch-and-bound search. If we incrementally increase the plan makespan and find optimal solutions for each one, we can trivially extend an optimal solution for a makespan of $d-1$ to a solution with makespan d by executing appropriate NOOP actions between layers $d-1$ and d. From this, we can produce a nontrivial lower bound on the best net benefit obtainable at makespan d and thus produce a nontrivial value for ub for the problem at makespan d. Hopefully, this will increase the number of pruning and promotion events, as described above, that occur early on in the branch-and-bound search.

It is worth noting that these branch-and-bound searches keep track of the best solution encountered during search. This can produce a solution to a planning problem at any time – before the first non-trivial solution is found, this would return the empty plan. In our experiments we terminated searches that lasted for longer than 30 minutes and recorded no solution. Alternatively, after 30 minutes, we could have returned the best solution found so far in the branch-and-bound search and compared this to the results obtained by a heuristic search planner; however, MiniMaxSat did not support this feature.

5.3 Survey Propagation

In this section we report experimental results concerning the incorporation of survey propagation into a WPMax-SAT solver as a variable and value selection heuristic. The performance of this strategy is compared with the existing heuristic method used by

MiniMaxSat1.0 and another randomised selection strategy. The average cost of the best solution found by each strategy over a series of restarts is plotted to visualise the trajectory that each strategy takes. By examining these trajectories we can see how quickly each strategy improves upon its solutions with time. This is of interest because finding better solutions earlier in the search should lead to more pruning, which reduces the number of nodes that need to be expanded to find an optimal solution. Our results find that for many cases the strategy based upon survey propagation leads to statistically significant improvements in the solution trajectory. This section concludes by discussing an approximation to the WPSP(y) strategy that makes the method more practical.

5.3.1 Experimental setup

Our experimental evaluation is based upon STRIPS problems taken from the third and fourth International Planning Competitions (IPCs). These problems have been transformed to net benefit problems with goal utility dependencies according to the procedure described in Section 5.2. The domains that are investigated are *Depot, DriverLog, ZenoTravel* and *Rovers* from IPC3 and *Truck* and *Pathways* from IPC4.

The results presented in Figure 5.3 on page 106 effectively tell us, for each problem in each domain, the makespan for which MiniMaxSat1.0 could not solve the WPMaxSAT encoding of that planning problem in under thirty minutes. We used this data to identify a single makespan, for each domain, that would yield a high proportion of challenging problems that could not be optimally solved in under thirty minutes by the MiniMaxSat1.0 solver. For domains taken from IPC3, we chose makespans of 6, 6, 7 and 5 for the Rovers, DriverLog, Depot and ZenoTravel domains, respectively; for domains taken from IPC4, we chose makespans of 8 and 9 for the Pathways and Truck domains, respectively. Since each problem is different, it is unlikely that these makespans hold some special significance across all problems within a domain that would favour one strategy over another; hence, we do not vary the makespan dimension in our experimental results.

From these problems, CNF encodings are constructed for the makespans specified above using the scheme described in Chapter 3. Again, the physical memory used to generate these encodings is limited to 1.5 GB to restrict our attention to manageable encodings. If the memory limit is exceeded, the problem is removed from our results. This is why we only report results for problems 1 to 13 out of the possible 20 problems from the ZenoTravel domain. The encodings are also preprocessed by iterating unit propagation and pure literal elimination to remove unit clauses and pure literals from the problem.

Our results compare the performance of the three branching strategies (WPSP(y), Rand and Basic) outlined in Section 4.6.4. The experiments are run on a Condor 6.8.3 pool of virtual machines spread across several Xen-enabled servers. This enables more data

to be gathered at the expense of losing the ability to take accurate timed measurements. Hence, we do not report any timed measurements in our data. Instead we record the best solution found by each strategy at certain points during the search. By studying this data we can see if one strategy on average finds lower cost solutions earlier in the search.

We are primarily interested in whether WPSP(y) can be used to provide better heuristic guidance. By limiting each strategy to the same number of leaf nodes that it can encounter inbetween reporting results, we can test if the WPSP(y) strategy is leading us towards lower cost solutions more on average and how soon these are discovered. This will help us to determine if the application of the ideas underpinning survey propagation in this new setting is successful. If our results indicate an improvement, then we can focus on making the method more practical by developing a version that guarantees convergence.

Since the WPSP(y) and Rand strategies are randomised algorithms, we used the restart strategy from MiniSat2.2 that operates according to the Luby sequence described in Section 4.5.1. This is done to avoid heavy tailed behaviour as we described earlier and according to the work of Gomes et al. (1998).

The solver progresses in a sequence of *restart iterations*. Each restart iteration is given an index i starting from one and increasing by one each time a restart is made. A limit is placed on the number of hard and soft clauses that can be encountered in each restart iteration. If this limit is exceeded then a restart is made and the solver progresses to the next restart iteration. Upon a restart, the current partial assignment must be discarded, and search in the next restart iteration begins from the root. The current best upper bound on cost is remembered between restart iterations; hopefully, this will lead to more pruning in the subsequent search.

In the following experiments, the limit on hard and soft clauses for the ith restart iteration is equal to $100 \times \mathit{luby}(i)$. The factor of 100 comes from the MiniSat2.2 implementation, and we did not experiment with different values for it as the intention is to establish a limited resource and to compare different search strategies within this constraint. If one algorithm produces better solutions while encountering the same number of conflicts, one can regard it as being more efficient at finding good solutions; however, there remains the possibility that the very best solutions are found in subtrees which contain many dead-ends. We are not too dissuaded by this line of argument because we believe that this desire for efficiency is important for the class of branch-and-bound search algorithms. In branch-and-bound search, the earlier that good solutions are found, the more pruning can occur, which in turn leads to faster improvements upon these solutions; thus, it has the potential to result in a positive feedback loop, which it seems desirable to encourage.

The experiments look at the behaviour of the strategies across the first 31 restarts. This allowed the vast majority of jobs to complete within 24 hours. If a strategy reaches the

31st restart iteration, it can encounter a maximum of 1600 soft or hard conflicts during that iteration before it must terminate. The next higher limit is encountered at restart 63, where the limit is 3200 hard and soft conflicts. We decided not to gather data beyond 31 restarts in order to keep the time taken to perform the experiments manageable and we believed that 31 restarts gave us sufficient insight into the difference in performance of our new method.

Experiments are repeated with different random seeds where randomisation is used. The WPSP(y) and Rand strategies are run on each problem ten times; however, since the Basic strategy is not randomised, it is run on each problem only once. Each time a strategy is run on a problem a trace is recorded which details the lowest cost of a solution found at the end of each restart iteration. For a particular trace these costs should be monotonically decreasing as the best solution found so far is remembered between restarts.

The y value

In the following results, we state that we use a value of $y = 100$ for the WPSP(y) strategy. However, for each problem, this is divided by the sum of weights of soft clauses in that problem, before it is used in the message passing equations. For example, if the sum of weights of soft clauses for a problem equals 1000, then the value used in message passing is actually $y = 0.1$. This step was introduced as an attempt to derive suitable values of y for each problem in a consistent manner. Through experimentation, we found that changing y did not noticeably change the quality of the results. Two different values for y that are sufficiently large are indistinguishable in their behaviour because they cause the exponential terms in message passing to underflow.

Problem statistics

Table 5.1 shows the size of each CNF encoding used after pure literal elimination and unit propagation has been iterated until no pure literals or unit clauses remain. A low clause-to-variable ratio α indicates that the problem is under-constrained and potentially easier to find a satisfiable assignment for than problems with higher α values.

Problem	Variables (N)	Clauses (M)	$\alpha = M/N$
Rovers			
01	336	2660	7.9
02	331	2564	7.7
03	382	2362	6.2
04	457	3249	7.1
05	717	7991	11.1
06	809	10203	12.6
07	843	8122	9.6
08	1334	22404	16.8
09	1456	22626	15.5
10	1569	28284	18.0
11	1577	39145	24.8
12	1334	24012	18.0
13	2174	84648	38.9
14	1796	35479	19.8
15	2230	69922	31.4
16	1946	37155	19.1
17	3172	96123	30.3
18	4538	162790	35.9
19	6113	518033	84.7
20	7709	736012	95.5
DriverLog			
01	177	537	3.0
02	476	2705	5.7
04	705	5043	7.2
05	671	4578	6.8
06	923	6434	7.0
07	1067	7978	7.5
08	1050	8036	7.7
09	573	3634	6.3
10	1173	11586	9.9
11	768	6421	8.4
12	556	4022	7.2
13	840	8752	10.4
14	895	10718	12.0
15	498	3562	7.2
16	1969	31327	15.9
17	3530	70255	19.9
18	2159	34123	15.8
19	2261	37017	16.4
20	7761	238641	30.7
ZenoTravel			
01	231	2879	12.5
02	295	5383	18.2
03	815	22636	27.8
04	602	15589	25.9
05	594	20706	34.9
06	916	31283	34.2
07	569	8629	15.2
08	1814	105165	58.0
09	1571	83176	52.9
10	2064	105926	51.3
11	2954	238945	80.9
12	2406	160370	66.7
13	2986	252880	84.7

Problem	Variables (N)	Clauses (M)	$\alpha = M/N$
Depot			
02	656	4375	6.7
03	679	4531	6.7
04	677	4587	6.8
05	765	5337	7.0
06	579	3340	5.8
07	1040	11836	11.4
08	1512	27516	18.2
09	1613	28092	17.4
10	1416	14600	10.3
11	1394	14053	10.1
12	1234	10670	8.6
13	1381	14894	10.8
14	2198	36056	16.4
15	2651	53901	20.3
16	2750	69518	25.3
17	4916	254276	51.7
18	6427	444301	69.1
19	3827	74039	19.3
20	4272	89328	20.9
21	11351	517419	45.6
Pathways			
01	422	2964	7.0
03	529	4015	7.6
04	663	5410	8.2
05	840	7776	9.3
06	1086	17113	15.8
07	1270	21673	17.1
08	1337	22650	16.9
09	1505	30416	20.2
10	1092	20068	18.4
11	1601	29926	18.7
12	1135	19362	17.1
13	1498	30501	20.4
14	1678	26587	15.8
15	1503	27480	18.3
16	1429	37797	26.4
17	1577	44689	28.3
18	1614	29210	18.1
19	1958	41573	21.2
20	1738	35931	20.7
21	1812	47262	26.1
22	2394	49822	20.8
23	2471	51935	21.0
24	1673	39888	23.8
25	2302	63600	27.6
26	2222	57771	26.0
27	2128	62830	29.5
28	2749	61840	22.5
29	2243	41788	18.6
30	1713	42853	25.0
Truck			
01	717	8706	12.1
02	801	10064	12.6
03	1400	25449	18.2
04	1404	24801	17.7
05	2070	47185	22.8
06	2170	50329	23.2
07	1639	44658	27.2
08	2124	65801	31.0
09	2406	80886	33.6
10	2878	104932	36.5

Table 5.1: Size of CNF encodings of planning problems after unit propagation and pure literal elimination has been performed. Columns contain the number of variables, number of clauses and the clause to variable ratio α for each encoding.

5.3.2 Results

The results are presented in two parts. The first section summarises the data obtained from multiple runs of the WPSP(y) and Rand strategies and combines this with the data from the single runs of the Basic strategy. These summaries detail the best solution found amongst multiple runs and the earliest time that a solution of that quality is found. They also show whether optimal solutions were found for each problem. This data is useful for interpreting the trajectory plots that we present in the subsequent section. The trajectory plots show how the sample mean varies over restart iterations for each strategy and problem pair. This allows us to see if one strategy is directing the search towards better solutions earlier than other strategies on average.

Lowest cost solutions

For each problem, the ten traces produced by the WPSP(y) and Rand strategies are taken and the cost of the best solution found is recorded together with the earliest and latest restart iterations that a solution of this cost was observed. The same is done with the single trace produced by the Basic strategy. This data is presented in Tables 5.3 and 5.4; these results are summarised in Table 5.2, which details the number of times that each strategy produced the best result out of the traces gathered.

In Tables 5.3 and 5.4, the final column 'WPSP(y) improvement' indicates the percentage change of the best solution found by the WPSP(y) method compared to the best solution found by either the Rand or Basic strategies. If this value is negative, it indicates an improvement, if it is positive it indicates that it is worse; for example, if the best solution cost found by WPSP(y) for a problem is 90 and the other strategies best solutions were 100 and 120, then the improvement is -10%.

For many problems, the WPSP(y) improvement is 0.0%, but this does not imply that WPSP(y) offered no benefit over the other strategies. If it found a solution of equal quality to another strategy but at an earlier restart iteration, then this is advantageous.

To give some indication as to when this occurs, for each score, the entries in the columns labelled 'first' and 'last' indicate the earliest and latest restart iteration that a solution of that cost was found. If the value in the column labelled 'last' is less than 31, then it indicates at least one of two scenarios have occurred. Either at least one run failed to complete all 31 restart iterations before the 24 hour cutoff limit was applied, or at least one run proved that the solution with that cost was optimal and no better solution existed.

Determining which of the two scenarios occured can be done by looking at the value in the 'opt' column, which indicates whether a solution found was shown to be optimal by

at least one run of that strategy. An entry of '**t**' in that column indicates that this was the case whereas an entry of 'f' indicates that no run proved optimality. If the 'opt' column is set to **t** then we know that at least one run proved optimality within a number of restarts between the values in the columns labelled 'first' and 'last'. If the value in the 'opt' column is set to the value 'f' then we know that all runs that found that solution cost did not make it past the number of restart iterations equal to the entry in the column labelled 'last'.

Across all domains in our results, the Basic strategy is consistently outperformed by the WPSP(y) and Rand strategies. In all domains except Pathways, WPSP(y) found the greatest number of best solutions to problems; WPSP(y) also proved optimality for more problems than the other two strategies.

	No. best performing runs			
Domain	WPSP(y)	Rand	Basic	Total
Rovers	19	5	3	20
Depot	20	6	1	20
DriverLog	19	7	1	19
Pathways	3	29	2	29
Truck	9	6	1	10
ZenoTravel	11	8	2	13

Table 5.2: The number of best performing runs that belonged to each strategy. Note that more than one strategy could obtain the same best score. The final column shows the total number of problems in the domain.

Problem	WPSP(y) score	first	last	opt	Rand score	first	last	opt	Basic score	first	last	opt	WPSP(y) improvement
Rovers													
01	**150**	1	6	**t**	**150**	1	7	**t**	242	27	31	f	0.0%
02	**223**	1	29	**t**	**223**	3	31	**t**	**223**	6	31	f	0.0%
03	**246**	3	31	**t**	**246**	6	31	**t**	**246**	4	6	**t**	0.0%
04	**146**	2	15	**t**	**146**	4	11	**t**	**146**	14	14	**t**	0.0%
05	**546**	5	31	f	562	23	31	f	919	28	31	f	-2.8%
06	**1097**	15	31	f	1120	29	31	f	1411	8	31	f	-2.1%
07	**408**	18	31	f	416	21	31	f	702	31	31	f	-1.9%
08	**796**	3	31	f	893	10	31	f	986	13	31	f	-10.9%
09	886	27	31	f	**883**	31	31	f	927	13	31	f	+0.3%
10	**1124**	31	31	f	1210	30	31	f	1387	15	31	f	-7.1%
11	**1079**	21	31	f	1236	31	31	f	1356	14	31	f	-12.7%
12	**485**	14	31	f	557	14	31	f	909	3	31	f	-12.9%
13	**1419**	27	31	f	1482	15	31	f	1726	10	31	f	-4.3%
14	**822**	31	31	f	876	25	31	f	1066	6	31	f	-6.2%
15	**1445**	21	29	f	1463	21	31	f	1788	23	31	f	-1.2%
16	**1442**	31	31	f	1503	30	31	f	1778	7	31	f	-4.1%
17	**2023**	25	31	f	2193	25	31	f	2222	5	31	f	-7.8%
18	**1608**	24	31	f	1723	31	31	f	1753	28	31	f	-6.7%
19	**2477**	15	17	f	2535	31	31	f	2822	10	31	f	-2.3%
20	**3041**	15	17	f	3213	6	31	f	3292	30	31	f	-5.4%
DriverLog													
01	**196**	1	4	**t**	**196**	2	4	**t**	**196**	15	15	**t**	0.0%
02	**574**	6	29	**t**	**574**	3	31	f	619	23	31	f	0.0%
04	**642**	13	31	f	741	31	31	f	766	22	31	f	-13.4%
05	**723**	2	31	f	736	30	31	f	1156	3	31	f	-1.8%
06	**442**	12	31	f	673	14	31	f	857	28	31	f	-34.3%
07	**616**	31	31	f	696	28	31	f	958	7	31	f	-11.5%
08	**811**	13	31	f	1002	22	31	f	1099	14	31	f	-19.1%
09	**680**	6	31	f	**680**	10	31	f	812	28	31	f	0.0%
10	**705**	18	31	f	707	23	31	f	1140	4	31	f	-0.3%
11	**740**	2	31	f	759	7	31	f	934	2	31	f	-2.5%
12	**1215**	2	31	f	**1215**	3	31	f	1260	4	31	f	0.0%
13	**746**	1	19	**t**	**746**	7	31	f	818	3	31	f	0.0%
14	**1234**	6	31	**t**	**1234**	13	31	f	1444	22	31	f	0.0%
15	**1494**	1	5	**t**	**1494**	7	31	**t**	1628	4	31	f	0.0%
16	**2577**	9	31	f	2678	18	31	f	2916	10	31	f	-3.8%
17	**2557**	29	31	f	3098	31	31	f	3188	21	31	f	-17.5%
18	**3217**	14	31	f	3261	22	31	f	3376	11	31	f	-1.3%
19	**3946**	25	31	f	4044	21	31	f	4660	31	31	f	-2.4%
20	**3965**	21	31	f	4432	25	31	f	4460	9	31	f	-10.5%
Truck													
01	**274**	2	31	f	**274**	7	31	f	**274**	23	31	f	0.0%
02	**389**	10	31	**t**	**389**	14	31	f	436	31	31	f	0.0%
03	**548**	14	31	f	**548**	6	31	f	574	29	31	f	0.0%
04	**721**	31	31	f	**721**	12	31	f	793	31	31	f	0.0%
05	**705**	30	31	f	723	31	31	f	845	13	31	f	-2.5%
06	**986**	22	31	f	**986**	31	31	f	987	21	31	f	0.0%
07	**638**	27	31	f	667	30	31	f	834	3	31	f	-4.3%
08	**832**	14	31	f	974	30	31	f	942	31	31	f	-11.7%
09	**1053**	7	11	f	1074	21	31	f	1077	30	31	f	-2.0%
10	1283	15	29	f	**1198**	19	31	f	1296	8	31	f	+6.6%

Table 5.3: IPC 3 (Rovers,$m = 6$) (DriverLog, $m = 6$) IPC 4 (Truck, $m = 9$) $y = 100$, $t_{lim} = 200$.

Problem	WPSP(y) score	first	last	opt	Rand score	first	last	opt	Basic score	first	last	opt	WPSP(y) improvement
Depot													
02	**357**	2	31	**t**	**357**	5	31	f	386	6	31	f	0.0%
03	**771**	4	31	**t**	**771**	5	31	f	800	3	31	f	0.0%
04	**391**	3	11	**t**	**391**	13	31	**t**	524	1	31	f	0.0%
05	**970**	14	31	f	**970**	14	31	f	1106	7	31	f	0.0%
06	**692**	1	7	**t**	**692**	5	29	**t**	**692**	3	31	**t**	0.0%
07	**475**	14	31	**t**	486	25	31	f	719	3	31	f	-2.3%
08	**704**	6	31	f	735	26	31	f	719	7	31	f	-2.1%
09	**1689**	19	31	f	1941	16	31	f	2072	1	31	f	-13.0%
10	**669**	7	31	f	681	18	31	f	737	30	31	f	-1.8%
11	**1299**	31	31	f	1304	31	31	f	1383	31	31	f	-0.4%
12	**1204**	5	31	f	1245	25	31	f	1297	5	31	f	-3.3%
13	**596**	4	31	f	**596**	30	31	f	726	5	31	f	0.0%
14	**635**	15	31	f	679	30	31	f	1027	4	31	f	-6.5%
15	**1536**	30	31	f	1809	26	31	f	1906	3	31	f	-15.1%
16	**778**	19	31	f	881	20	31	f	816	2	31	f	-4.7%
17	**450**	15	31	f	894	30	31	f	936	13	31	f	-49.7%
18	**1781**	22	31	f	2210	28	31	f	2267	1	31	f	-19.4%
19	**1051**	30	31	f	1294	15	31	f	1150	1	31	f	-8.6%
20	**1845**	25	31	f	2069	31	31	f	2281	29	31	f	-10.8%
21	**651**	28	31	f	1138	31	31	f	1300	7	31	f	-42.8%
Pathways													
01	**93**	2	31	f	**93**	2	31	**t**	**93**	3	7	**t**	0.0%
03	**341**	13	31	f	**341**	6	31	**t**	349	10	31	f	0.0%
04	349	31	31	f	**335**	7	31	**t**	371	8	31	f	+4.0%
05	617	22	31	f	**590**	10	31	f	793	11	31	f	+4.4%
06	1219	23	31	f	**1198**	29	31	f	1231	14	31	f	+1.7%
07	1711	15	31	f	**1686**	26	31	f	1767	8	31	f	+1.5%
08	2004	21	31	f	**1997**	22	31	f	2048	16	31	f	+0.3%
09	2083	29	31	f	**2070**	31	31	f	2124	1	31	f	+0.6%
10	**1229**	30	31	f	**1229**	15	31	**t**	1243	30	31	f	0.0%
11	2262	15	31	f	**2171**	31	31	f	2373	6	31	f	+4.0%
12	2203	30	31	f	**2177**	22	31	f	**2177**	15	31	f	+1.2%
13	2257	30	31	f	**2136**	29	31	f	2225	7	31	f	+5.4%
14	3083	31	31	f	**3034**	22	31	f	3073	18	31	f	+1.6%
15	2721	31	31	f	**2473**	31	31	f	2796	6	31	f	+9.1%
16	1927	25	31	f	**1914**	30	31	f	1995	4	31	f	+0.7%
17	2128	31	31	f	**1927**	29	31	f	2154	8	31	f	+9.4%
18	3636	15	31	f	**3059**	30	31	f	3655	16	31	f	+15.9%
19	3634	31	31	f	**3417**	31	31	f	3664	17	31	f	+6.0%
20	3755	30	31	f	**3168**	31	31	f	3832	29	31	f	+15.6%
21	3135	29	31	f	**2968**	29	31	f	3255	13	31	f	+5.3%
22	4292	31	31	f	**4161**	31	31	f	4283	23	31	f	+3.1%
23	5062	13	31	f	**4801**	31	31	f	5120	3	31	f	+5.2%
24	2401	31	31	f	**2106**	30	31	f	2800	7	31	f	+12.3%
25	4307	31	31	f	**4277**	27	31	f	4500	14	31	f	+0.7%
26	3749	15	31	f	**3223**	31	31	f	3741	7	31	f	+14.0%
27	3834	29	31	f	**3829**	27	31	f	3952	15	31	f	+0.1%
28	4856	29	31	f	**4508**	31	31	f	4822	15	31	f	+7.2%
29	6030	22	31	f	**5612**	31	31	f	6171	29	31	f	+6.9%
30	3112	30	31	f	**2600**	31	31	f	3139	2	31	f	+16.5%
ZenoTravel													
01	**22**	1	1	**t**	**22**	1	2	**t**	**22**	2	2	**t**	0.0%
02	**126**	1	5	**t**	**126**	1	5	**t**	**126**	5	5	**t**	0.0%
03	**270**	5	31	**t**	285	31	31	f	442	22	31	f	-5.3%
04	**275**	9	31	f	**275**	22	31	f	301	31	31	f	0.0%
05	**147**	2	12	**t**	**147**	2	20	**t**	370	31	31	f	0.0%
06	**187**	15	31	f	**187**	22	31	**t**	327	6	31	f	0.0%
07	**576**	28	31	f	**576**	11	31	f	730	3	31	f	0.0%
08	**441**	29	31	f	459	14	31	f	521	3	31	f	-3.9%
09	691	14	31	f	**600**	30	31	f	997	22	31	f	+13.2%
10	**671**	31	31	f	857	30	31	f	968	22	31	f	-21.7%
11	603	12	21	f	**585**	6	31	f	966	8	31	f	+3.0%
12	**655**	18	24	f	767	15	31	f	981	6	31	f	-14.6%
13	**1009**	18	29	f	1140	28	31	f	1371	31	31	f	-11.5%

Table 5.4: IPC 3 (Depot, $m = 7$) (ZenoTravel, $m = 5$) IPC 4 (Pathways, $m = 8$) $y = 100$, $t_{lim} = 200$.

Solution trajectories

In order to examine the effects of the three branching strategies, their trajectories over several restarts are plotted. Since each problem p that is solved is finite, and the initial decision variable selection strategy s is a random process, then the cost, $X_{i,s,p}$, of the best solution found at the end of each restart iteration i is a random variable and will have a population mean $\mu_{i,s,p}$. We are interested in how this population mean is altered by the use of different selection strategies. Note that since all problems have an optimal solution, for different complete strategies s and s' we expect $\lim_{i\to\infty} \mu_{i,s,p} = \lim_{i\to\infty} \mu_{i,s',p}$.

Ideally, we want to find a strategy that lowers the mean as quckily as possible towards the optimal solution cost. In order to investigate this, we plot, for each problem and strategy pair, the sample average of the lowest cost solution obtained at the end of each restart iteration calculated over ten runs. Due to early termination, there is a possibility that fewer than ten runs reached a certain restart iteration. In those cases, we ignore all results for that iteration; hence, the sample averages that are displayed are supported by exactly $n = 10$ data points.

The plots are constructed using the following manner for each problem and solution strategy. For each iteration i, strategy s and problem p, we compute the sample mean $\bar{X}_{i,s,p}$ and the sample standard deviation[3] $S_{i,s,p}$ of the lowest cost solution found by that configuration, with a sample size n. If n is sufficienty large then $\bar{X}_{i,s,p}$ is normally distributed by the central limit theorem and

$$(\bar{X}_{i,s,p} - \mu_{i,s,p})\frac{\sqrt{n}}{S_{i,s,p}}, \tag{5.1}$$

follows Student's t-distribution with $n-1$ degrees of freedom (Spiegel and Stephens 2008). We use this to plot the 95% confidence interval for the true population mean $\mu_{i,s,p}$ as

$$\bar{X}_{i,s,p} \pm 2.262\frac{S_{i,s,p}}{\sqrt{n}}, \tag{5.2}$$

where 2.262 is taken from the 95% two-sided Student's t-distribution with 9 degrees of freedom.

Hypothesis 1. *Null hypothesis $H_0(s_1, s_2, i, p)$: There is no difference in the mean lowest cost solution obtained by strategies s_1 and s_2 at the ith restart iteration for problem p.*

Hypothesis 2. *Alternate hypothesis $H_a(s_1, s_2, i, p)$: There is a difference in the mean lowest cost solutions obtained by strategies s_1 and s_2 at the ith restart iteration for problem p.*

[3]This is computed from the unbiased sample variance using Bessel's correction.

By using the above confidence interval we can apply a conservative visual check for significance by testing if the intervals overlap for two different strategies. If the intervals do not overlap then we can infer significance at the 5% level and reject the null hypothesis; however, if the intervals do overlap we can neither reject nor accept the null hypothesis and are required to conduct a less conservative analysis of the data (Schenker and Gentleman 2001). We adopt this scheme for ease of presentation of our results. In many of the graphs that we present there is a clear separation of the 95% confidence intervals so we argue that our method frequently reduces the average lowest cost of solutions found through the intial phase of the search. This scheme is also conservative in the sense that we could apply a one-sided test to show significance in the reduction direction with a greater chance of being able to reject the null hypothesis.

Distribution analysis

In order to test the validity of the above assumptions used in forming the confidence intervals, we investigated the empirical distribution of best solution costs obtained at the ith restart iteration. To check for heavy-tailed behaviour, we looked at the sample mean over 5000 trials using the Rand selection strategy. The problems we experimented with had to be sufficiently easy to solve in order to gather data for such a large number of trials.

Figure 5.4 shows that the sample mean converges to a single value as the number of trials increase. This is what one would expect when each sample is generated from independent and identical distributions with finite expected value, according to the law of large numbers.

Since the problems have finite size, the variance σ_i^2 is also finite. One would expect, through the central limit theorem, that sample means over n trials follow a normal distribution $N(\mu_i, \frac{\sigma_i^2}{n})$ for large n. To check this, we divided the 5000 trials into 500 groups of 10 trials, which were used to generate 500 sample means. Figure 5.5 shows histograms of the sample means obtained for various restart iterations. One can visually check that the histograms are close to the predicted normal distributions, which are plotted using the sample mean and sample variance over the 5000 trials to give an estimate for $\mu_{i,s,p}$ and $\sigma_{i,s,p}^2$, respectively (see Figure 5.4). One can see that as the restart iteration i increases, the peak of the sample mean is reduced towards the optimal solution, which has cost equal to 574 for p02 in DriverLog. We see that for $n = 10$ the approximation is reasonable.

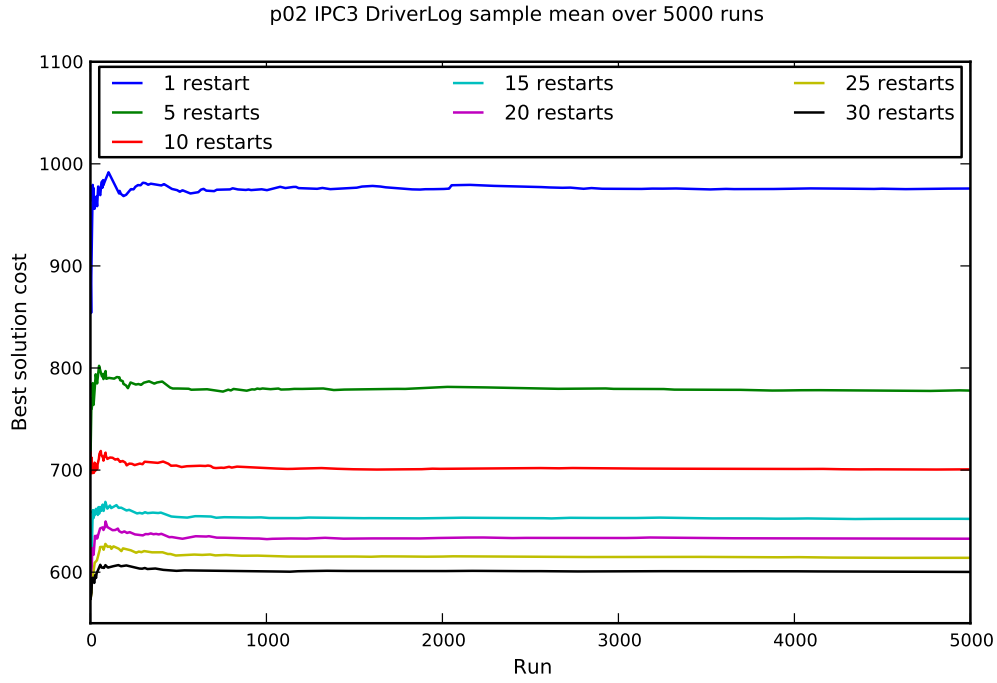


Figure 5.4: Convergence of the sample mean over 5000 runs of the Rand selection strategy on problem two from the DriverLog domain.

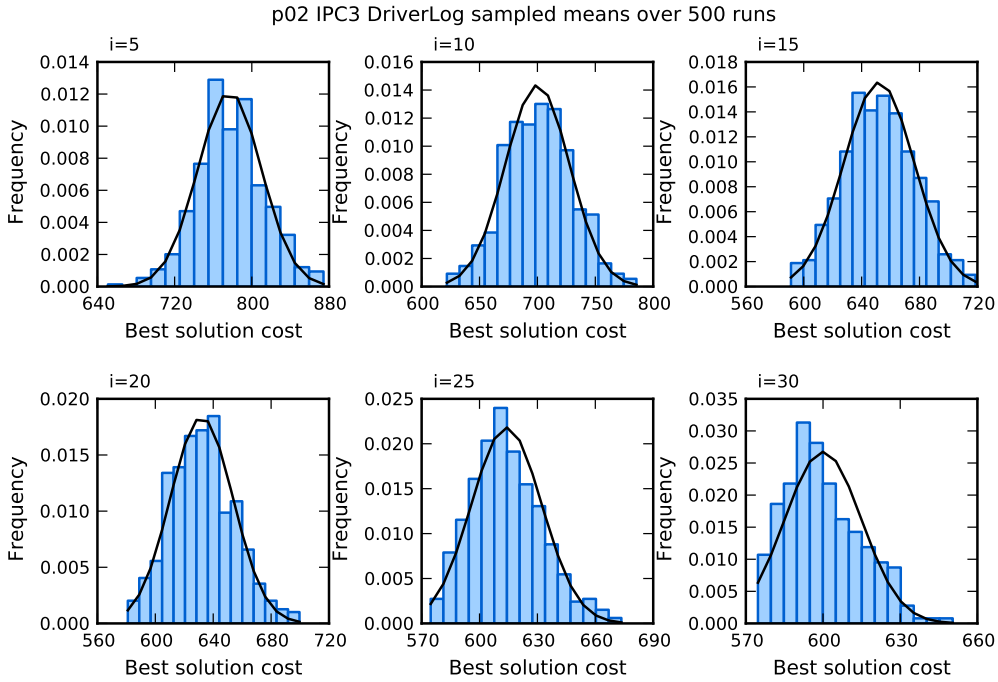


Figure 5.5: Distribution of 500 sample means obtained by running the Rand selection strategy on problem two from the DriverLog domain ten times to obtain each sample. Overlayed is the normal distribution that the central limit theorem predicts the distribution will converge to as the sample sizes increase. The cost of the optimal solution is 574.

5.3.3 Solution trajectories

The following pages present and discuss the solution trajectories we measured for each of the three strategies across the six domains we obtained data for.

DriverLog

Results for the DriverLog domain, shown as trajectory plots in Figures 5.6 and 5.7 show the WPSP(y) strategy to be advantageous over the other strategies we considered. In problems 16, 17, 18 and 20 there is a clear separation between WPSP(y) and other strategies. The Basic strategy either matches the trajectory or is worse than the Rand strategy, with the exception of problem 20 where it shows better performance in the inital restarts.

Table 5.5 shows the percentage of successful variable selections performed by the WPSP(y) strategy. This is high for most problems in the domain, close to 100%; however, for problems 2, 13, 14 and 15, which are optimally solved by at least one run of the WPSP(y) strategy, the percentage of successful selections is low.

With the exception of problem 10 and those problems for which optimality is proven, the WPSP(y) strategy is able to complete all 31 restart iterations. Combining this statistic with the high percentage of successful variable selections that are made by the WPSP(y) strategy indicates that for many problems in this domain it is easy to find fixed points of the message passing equations that yield fact variables with sufficiently high bias to be considered for selection.

	Trial (%)									
Problem	1	2	3	4	5	6	7	8	9	10
DriverLog										
01	100	100	100	100	100	100	100	100	100	100
02	61	55	62	57	56	58	62	53	51	62
04	100	100	100	100	100	100	100	100	100	100
05	99	95	99	96	98	99	97	97	95	95
06	100	100	100	100	100	100	100	100	100	100
07	100	100	100	100	100	100	100	100	100	100
08	100	100	100	100	100	100	100	100	100	100
09	99	100	100	100	100	100	100	100	100	100
10	100	99	99	100	100	100	100	100	99	100
11	86	82	78	86	85	79	84	88	91	83
12	96	98	96	97	93	96	95	97	95	99
13	70	68	66	63	67	65	79	68	64	68
14	67	58	57	64	55	59	62	54	60	59
15	36	55	55	54	33	40	37	44	56	37
16	67	64	63	64	66	66	67	66	66	66
17	100	100	100	100	100	100	100	100	100	100
18	76	70	76	72	69	73	76	72	78	74
19	100	100	100	100	100	100	100	100	100	100
20	100	100	100	100	100	100	100	100	100	100

Table 5.5: Percentage of successful selections performed by the WPSP(y) strategy for the DriverLog domain.

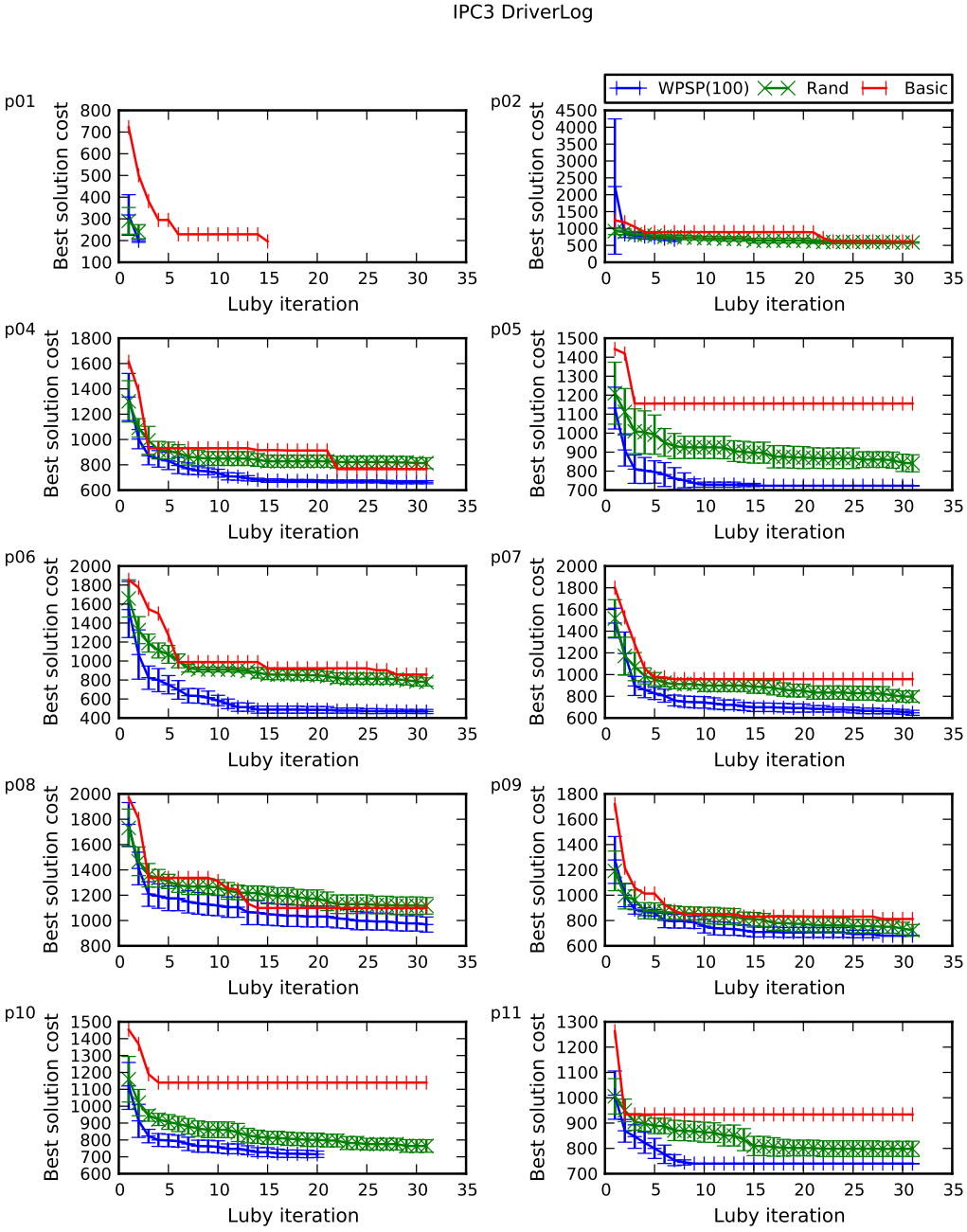


Figure 5.6: Solution trajectories for problems 1–11 from the DriverLog domain (makespan = 6). Problem 1 is optimally solved by all three strategies in at least one of the ten runs. Problem 2 is optimally solved by at least one run of the WPSP(y) strategy.

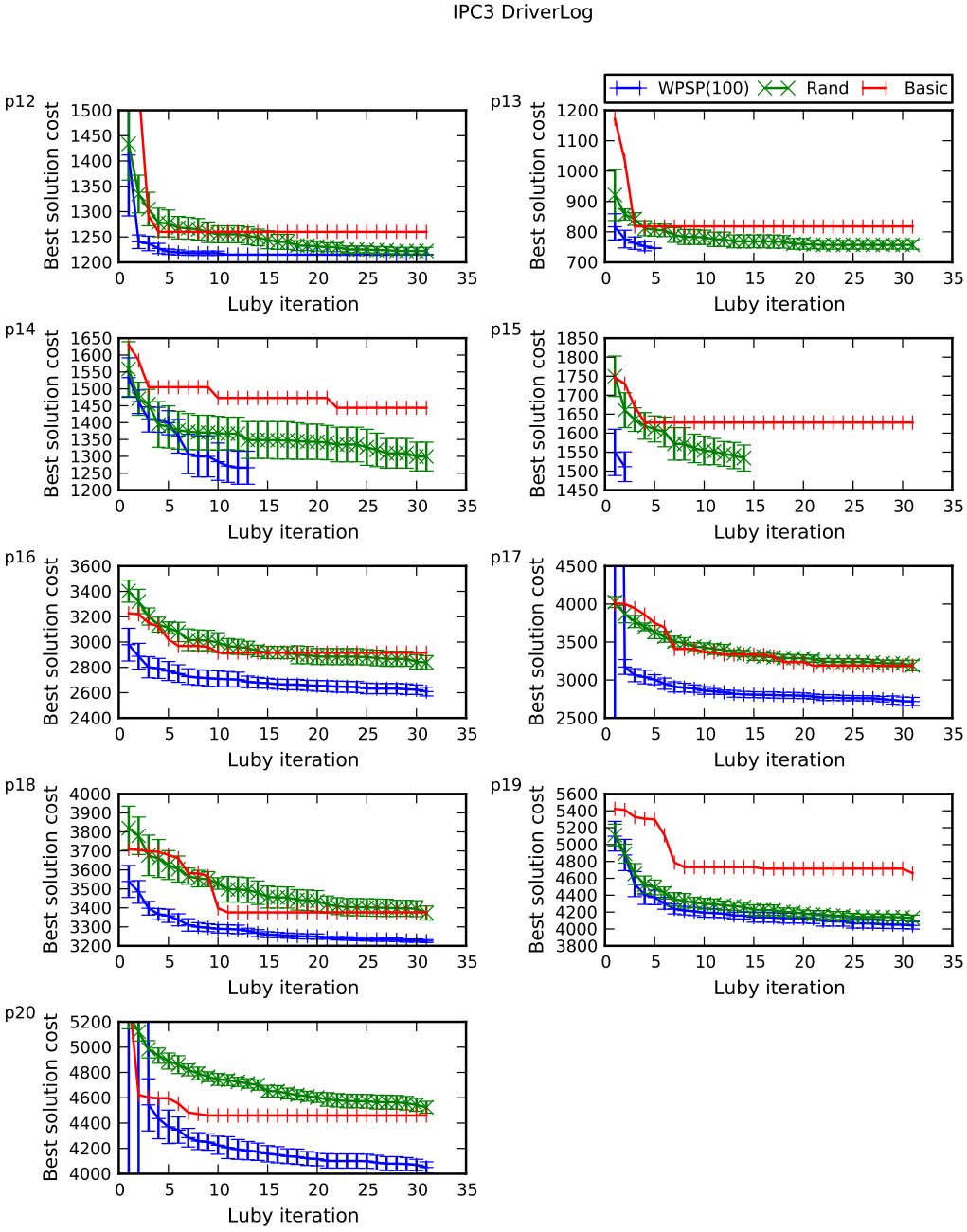


Figure 5.7: Solution trajectories for problems 12–20 from the DriverLog domain (makespan = 6). Problems 13, 14 and 15 are optimally solved by the WPSP(y) strategy. Problem 15 is also optimally solved by at least one run of the Rand strategy.

Depot

The trajectories shown in Figures 5.8 and 5.9 drawn from the experimental results from the Depot domain show that the WPSP(y) strategy exhibits good performance when compared to the Rand and Basic strategies. It is competitive over all problems: the average performance of WPSP(y) is not worse than the alternative strategies for any problem in this domain. In problems 9, 10, 12, 15 and 17, WPSP(y) achieves, by restart fifteen, a statistically significant improvement in the average lowest cost solution found over that achieved at restart thirty-one by either of the alternative strategies. WPSP(y) also solves problems 2, 3, 4, 6 and 7 optimally in at least one of the ten runs, compared to Rand , which solves only problems 4 and 6 in at least one run, and Basic that solves only problem 6.

The WPSP(y) strategy performs particularly well on problem 21: by the end of 31 restart iterations, it has a mean cost around half that of the other two strategies. The performance of Basic is notable on problem 16, where it finds a very low-cost solution within the first restart iteration which is only contested by other strategies in the latter half of restart iterations.

The percentage of successful variable selections performed by the WPSP(y) strategy, as presented in Table 5.6, is high (over 90%) for problems 12 and above. With the exception of problem 18, where the WPSP(y) strategy fails to complete all thirty-one restart iterations, the data suggests that it is not too difficult to find fixed points of the message passing equations and that they often result in fact variables having sufficient bias to be considered for selection.

	Trial (%)									
Problem	1	2	3	4	5	6	7	8	9	10
Depot										
02	95	95	92	92	96	96	94	96	94	93
03	87	88	88	87	90	86	88	87	89	89
04	86	79	80	80	85	80	80	80	78	82
05	94	89	90	95	92	94	89	88	93	92
06	95	90	87	87	86	81	90	82	93	76
07	88	88	88	88	86	87	88	89	90	89
08	88	91	90	89	91	86	90	87	91	87
09	95	88	94	95	91	89	93	91	90	92
10	96	97	92	94	97	92	97	99	97	97
11	78	74	74	69	75	68	73	62	67	71
12	95	92	93	96	94	93	93	94	94	93
13	93	90	94	91	91	88	89	91	87	94
14	99	100	99	100	99	100	100	100	100	100
15	99	99	100	99	99	99	100	99	98	99
16	91	91	94	92	92	93	89	91	92	91
17	98	100	99	99	100	100	99	99	98	99
18	100	100	99	99	99	99	98	100	99	99
19	91	92	89	92	94	90	92	88	91	92
20	97	94	94	96	96	95	96	95	97	96
21	100	100	100	99	100	100	99	100	100	100

Table 5.6: Percentage of successful selections performed by the WPSP(y) strategy for the Depot domain.

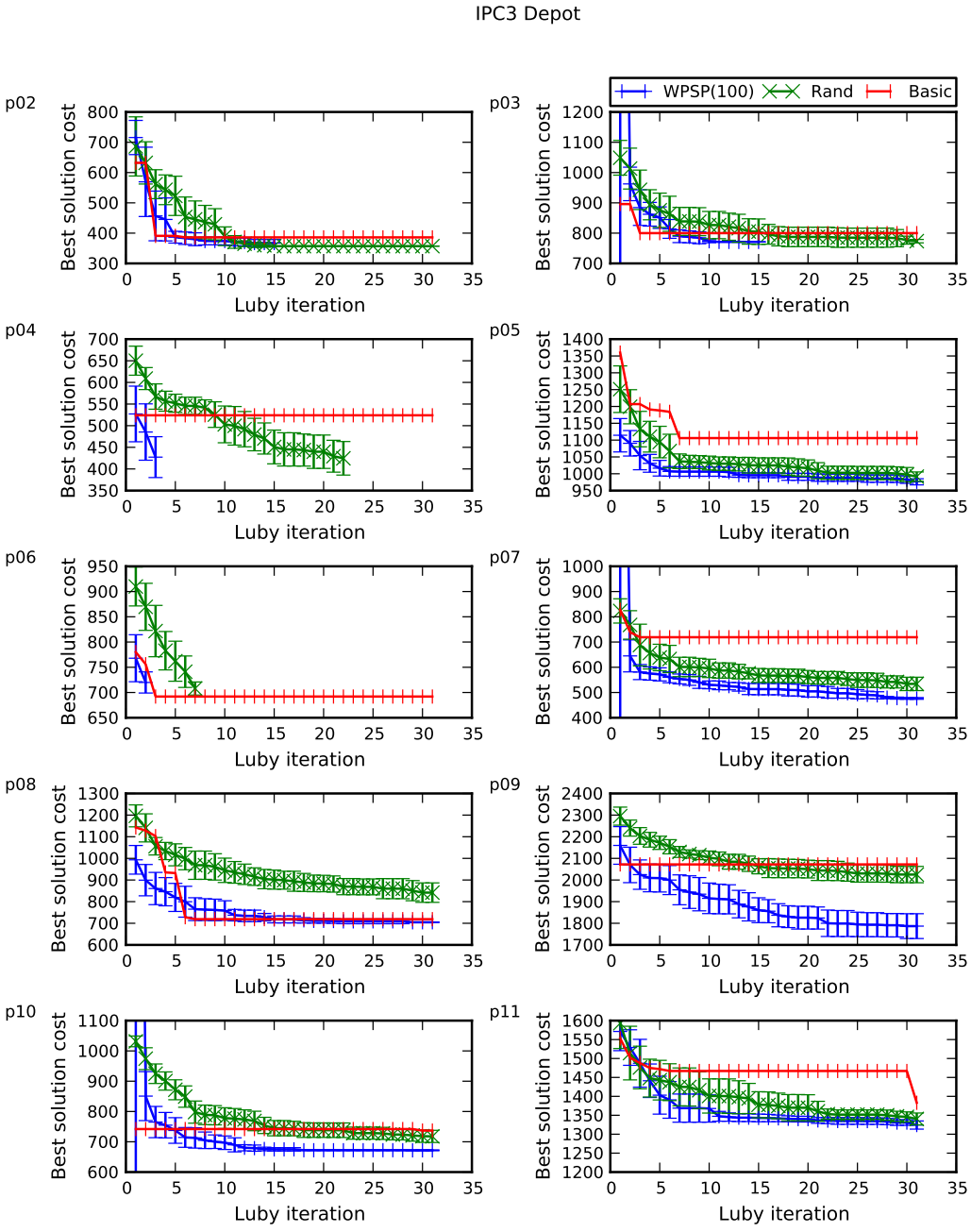


Figure 5.8: Solution trajectories for problems 2–11 from the Depot domain (makespan = 7). Problems 2, 3, 4, 6 and 7 were optimally solved by at least one run out of the ten runs of WPSP(y). Problems 4 and 6 were optimally solved by at least one run out of the ten runs of Rand. Problem 6 was optimally solved by Basic.

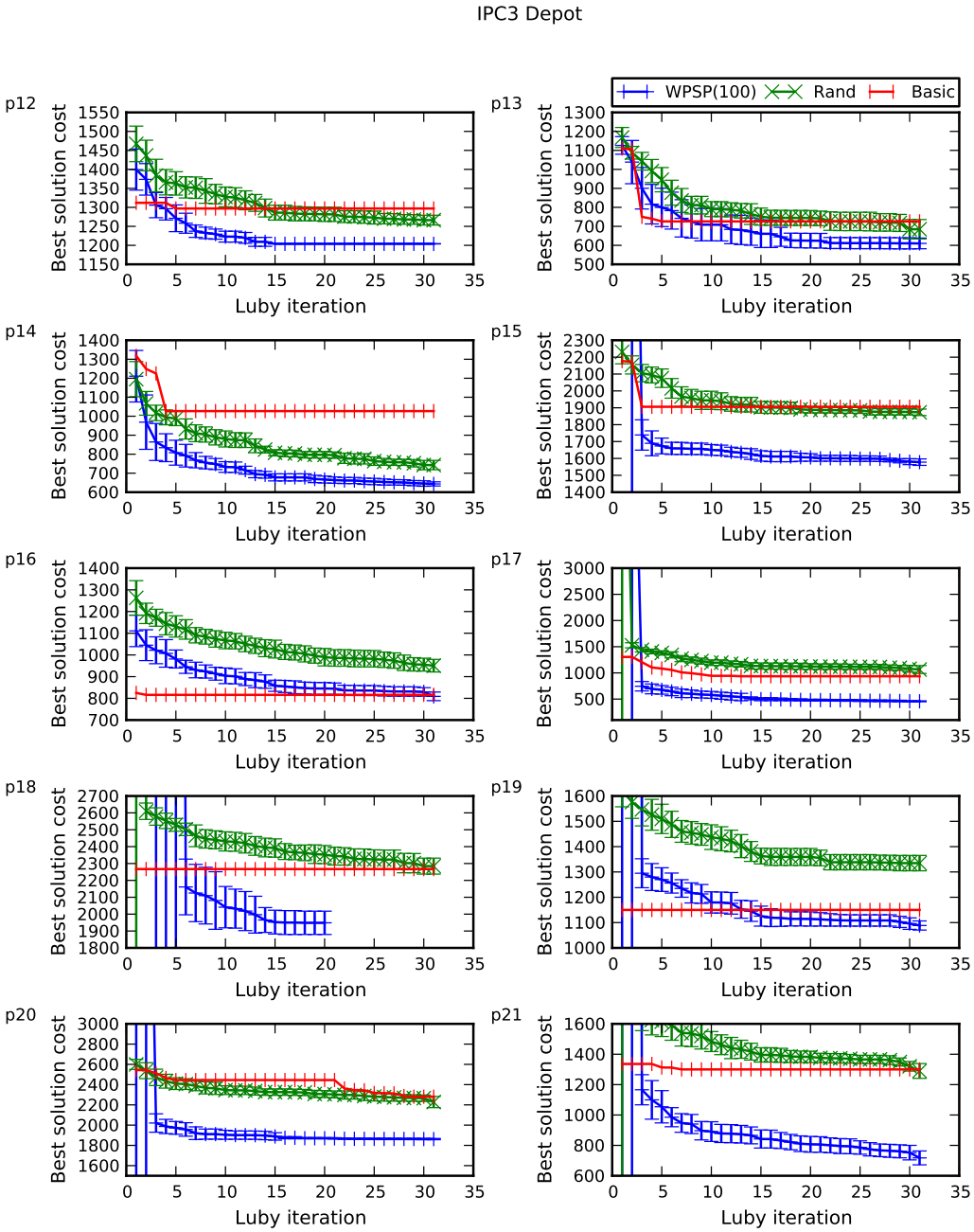


Figure 5.9: Solution trajectories for problems 12–21 from the Depot domain (makespan = 7).

Rovers

The trajectories shown in Figures 5.10 and 5.11 drawn from experimental results from the Rovers domain suggest that the WPSP(y) strategy is very effective at reducing average best solution costs. The Rand strategy is competitive with the Basic strategy, frequently outperforming it but sometimes lagging behind. In contrast, the WPSP(y) strategy is usually better and is never worse than the other strategies.

The percentage of successful variable selections performed by the WPSP(y) strategy is high, over 95% for most problems, as can be seen in Table 5.7. This means that fixed points of message passing are almost always found and that when they are, there are fact variables that have a bias that exceeds the threshold that makes them eligible for selection. The WPSP(y) strategy failed to complete all thirty-one restart iterations in at least one run for problems 15, 18, 19 and 20. Given that the percentage of successful selections by the WPSP(y) strategy are so high, this must indicate that iterations of message passing are slow and restarts are being made in order to search for fixed points of message passing. However, in those cases, with the exception of problem 18, the WPSP(y) strategy achieves a mean cost that is lower than the other strategies achieve by the end of thirty-one restarts.

	Selections (%)									
Problem	1	2	3	4	5	6	7	8	9	10
Rovers										
01	99	91	87	95	99	91	89	95	90	95
02	96	96	96	97	98	95	97	98	98	98
03	76	84	88	77	84	75	75	77	77	81
04	100	93	94	88	100	92	96	100	94	94
05	100	98	98	98	100	99	99	97	99	99
06	98	99	99	99	99	97	98	99	99	99
07	96	98	100	99	99	99	98	99	99	99
08	100	100	100	100	100	100	100	99	100	100
09	99	99	99	100	99	99	98	98	99	97
10	100	100	100	100	100	100	100	100	100	100
11	100	100	99	100	100	100	100	100	100	100
12	99	100	100	100	100	100	100	100	99	100
13	100	100	100	100	100	100	100	99	100	100
14	99	99	99	99	100	100	99	100	100	100
15	100	100	100	100	100	100	100	100	99	100
16	100	100	100	99	100	99	99	100	100	97
17	100	100	100	100	100	100	100	100	100	100
18	100	100	100	100	100	100	100	100	100	100
19	100	100	100	100	100	100	100	100	100	100
20	100	100	100	99	100	100	100	99	100	99

Table 5.7: Percentage of successful selections performed by the WPSP(y) strategy for the Rovers domain.

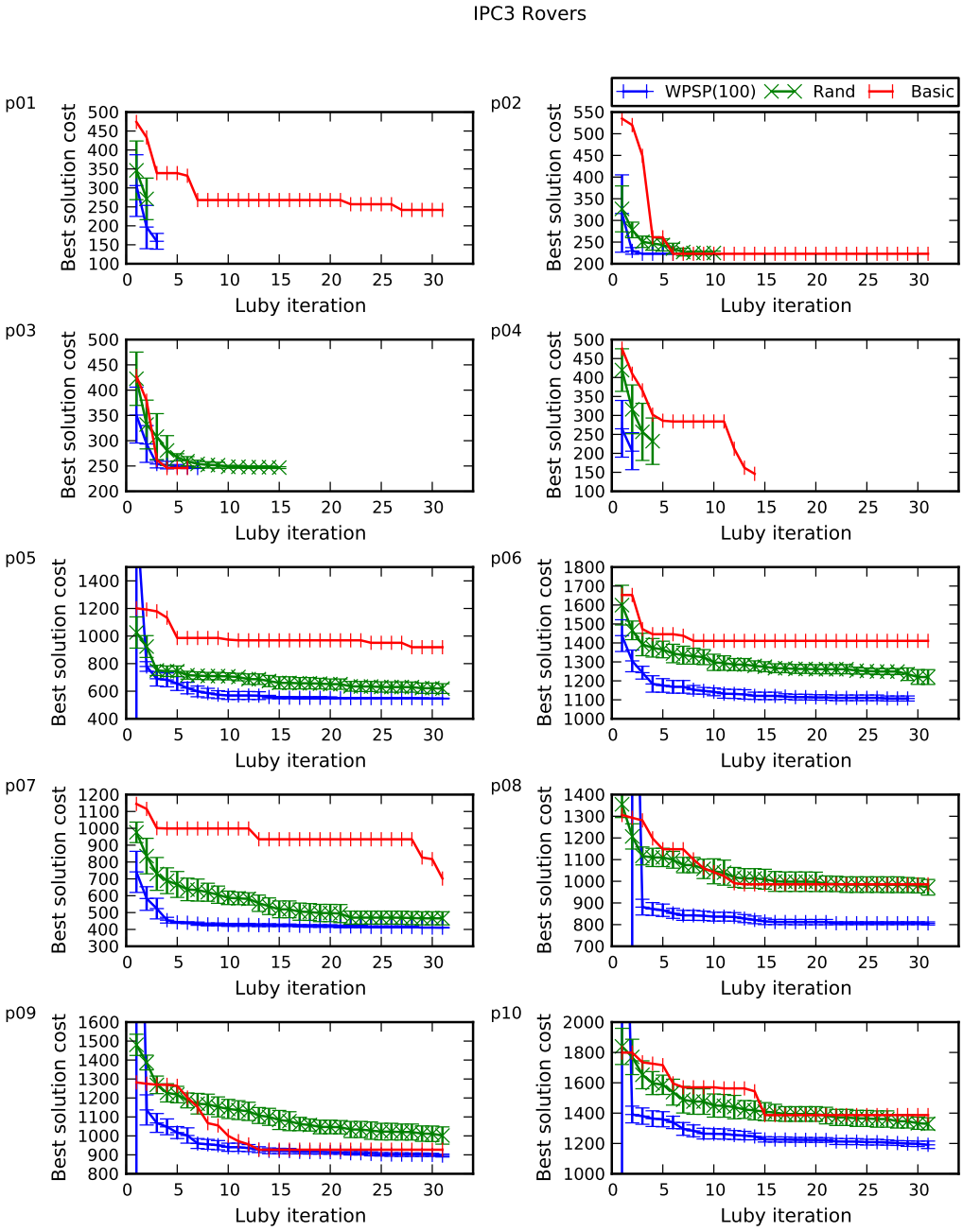


Figure 5.10: Solution trajectories for problems 1–10 from the Rovers domain (makespan = 6). Problems 1–4 are solved optimally by at least one run of the WPSP(y) and Basic strategies. Problems 3–4 are solved optimally by the Basic strategy.

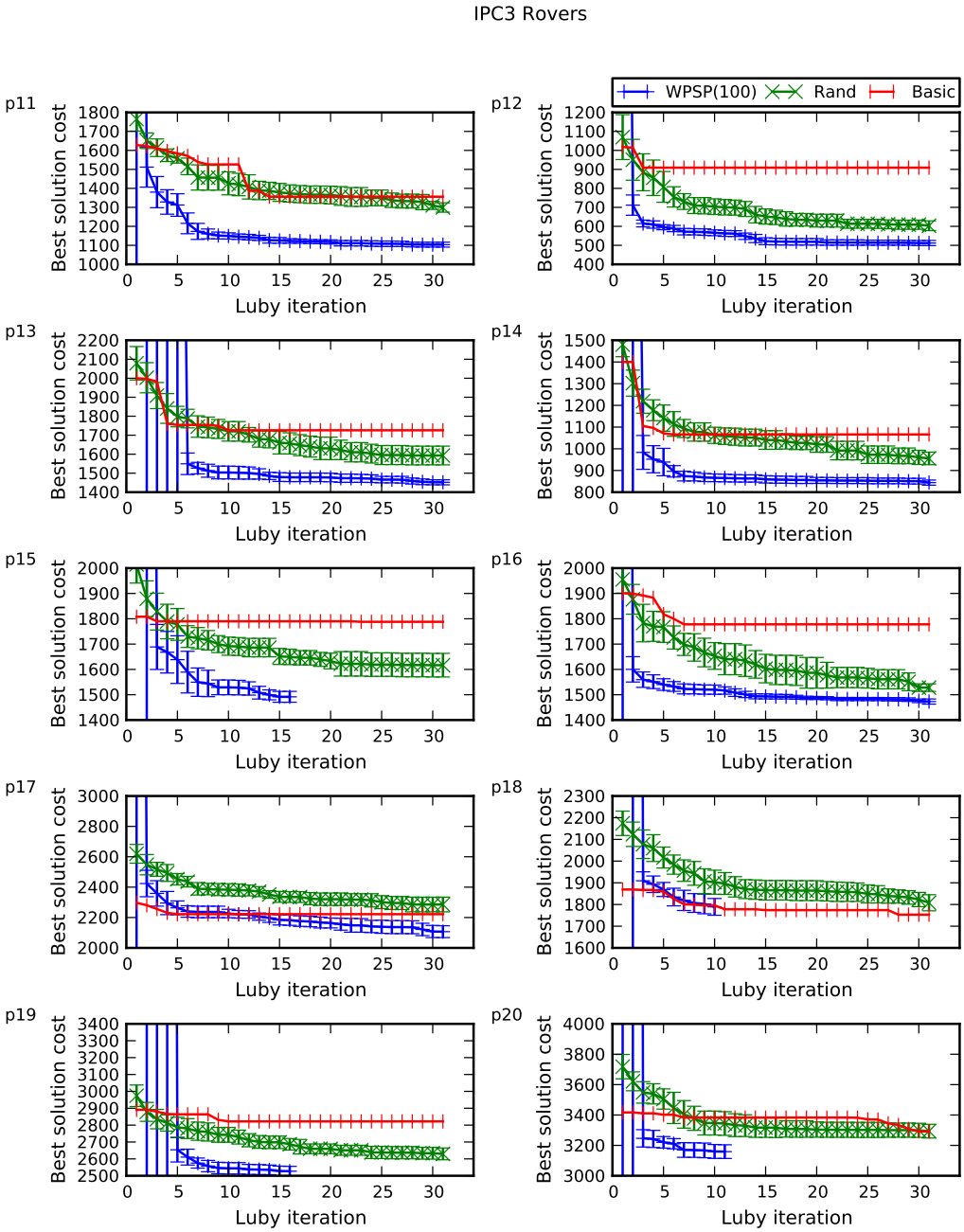


Figure 5.11: Solution trajectories for problems 11–20 from the Rovers domain (makespan = 6).

Truck

Although the results presented in Table 5.3 suggest that the WPSP(y) strategy is able to find lower cost solutions than the other strategies, the trajectories shown in Figure 5.12 reveal that the average performance of the WPSP(y) strategy is poor compared to the other strategies. The average performance is significantly worse on problems 7, 9 and 10, where the solution cost means are so large that including them in the graph would have resulted in a y-axis scale that would not allow enough detail to observe the differences between the Rand and Basic strategies. For other problems the WPSP(y) strategy performed poorly on the first 10–15 restart iterations where it often failed to find assignments that satisfied all hard clauses which resulted in a high mean and variance in the solution costs. After this period of a high mean, the mean suddenly drops to a value that is competitive with that achieved by the other strategies.

The percentage of successful variable selections made by the WPSP(y) strategy is shown in Table 5.8 and reveals that it had difficulty finding useful fixed points of the message passing equations. For any problem/run pair in this domain, the WPSP(y) strategy made approximately between 50% and 80% successful variable selections, having to resort to Basic selection in the other cases.

Only the WPSP(y) strategy was able to prove optimality for a problem in this domain (problem 2).

	Selections (%)									
Problem	1	2	3	4	5	6	7	8	9	10
Truck										
01	70	70	65	58	62	65	62	61	67	62
02	65	66	66	65	67	67	62	59	64	66
03	67	66	66	69	69	64	69	67	68	66
04	72	71	74	72	74	77	71	74	75	74
05	75	75	74	74	71	77	78	80	73	77
06	75	79	78	80	79	74	78	76	69	80
07	79	74	81	78	82	76	80	87	81	79
08	75	73	73	76	69	75	79	75	76	72
09	80	75	81	80	80	76	76	75	78	69
10	76	76	75	79	81	77	74	77	77	80

Table 5.8: Percentage of successful selections performed by the WPSP(y) strategy for the Truck domain.

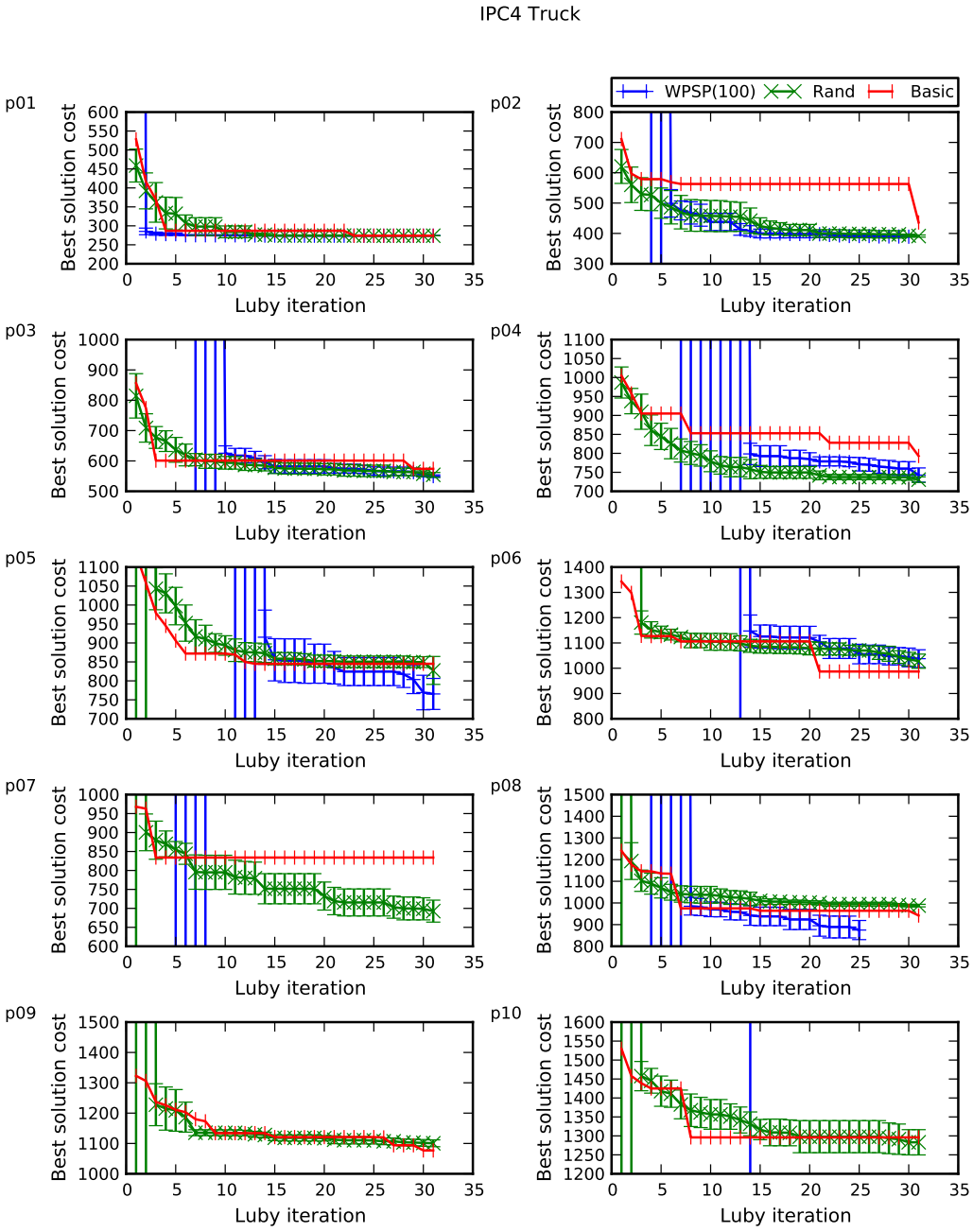


Figure 5.12: Solution trajectories for problems 1–10 from the Truck domain (makespan = 9). The WPSP(y) strategy appears off the chart and is worse than the other strategies for problems 7, 9 and 10.

Pathways

The results in Table 5.9 show that the WPSP(y) strategy has a very high selection rate, which is greater than 97% for most problems in this domain. The trajectories in Figures 5.13, 5.14 and 5.15 also indicate that the WPSP(y) strategy completed all 31 iterations for every one of the 10 runs on each problem. Together, these two statistics suggest that it is easy to find fixed points of the message passing equations for which there are variables of sufficiently high bias to be considered for selection. Disappointingly, the WPSP(y) strategy performs poorly compared to the other strategies.

The Rand strategy matches or outperforms the other strategies across problems in this domain. Although the WPSP(y) strategy often closes in on the Basic strategy by the end of the 31st restart iteration, it also often exhibits very high variance in the initial restart iterations (see problems 23 and 28 in Figure 5.15). This high variance is a result of not finding an assignment that satisfies all hard clauses, which is denoted by a solution cost equal to the large hard clause weight, for at least one run.

The WPSP(y) strategy also failed to prove optimality for any problems in this domain, whereas the Rand strategy was able to prove optimality for runs on problems 1, 3, 4 and 10; and the Basic strategy was able to prove optimality for runs on problem 1.

	Selections (%)									
Problem	1	2	3	4	5	6	7	8	9	10
Pathways										
01	80	80	79	77	80	82	80	79	82	81
03	89	82	84	85	86	81	89	89	86	85
04	96	95	96	93	96	94	96	96	95	96
05	95	93	96	95	96	95	96	96	97	95
06	99	99	99	99	98	98	98	99	99	99
07	98	98	97	97	98	97	98	98	98	98
08	98	98	98	98	99	99	99	99	99	98
09	97	97	98	97	98	97	96	97	97	98
10	99	98	98	99	98	99	98	99	98	99
11	99	98	99	99	99	99	99	97	99	100
12	99	98	98	99	99	99	99	99	98	99
13	100	100	99	99	100	99	99	99	99	100
14	100	99	99	100	100	99	99	100	99	100
15	99	99	99	99	99	100	99	99	100	100
16	100	99	99	100	100	99	99	99	100	99
17	99	100	100	99	99	99	98	100	98	98
18	100	99	100	100	100	99	100	100	100	100
19	99	100	100	100	99	100	100	100	100	100
20	99	100	100	99	100	100	99	100	100	100
21	99	100	99	99	100	99	100	100	98	100
22	100	100	100	100	100	100	100	100	100	100
23	100	100	100	100	100	100	100	100	100	100
24	100	100	100	100	100	99	100	100	100	99
25	100	100	100	100	100	100	100	100	100	100
26	100	100	100	99	100	100	100	100	100	100
27	99	99	99	99	100	100	99	99	100	100
28	100	100	100	100	100	100	100	100	100	100
29	99	100	100	100	100	100	100	99	100	100
30	99	99	99	99	99	99	98	99	98	99

Table 5.9: Percentage of successful selections performed by the WPSP(y) strategy for the Pathways domain.

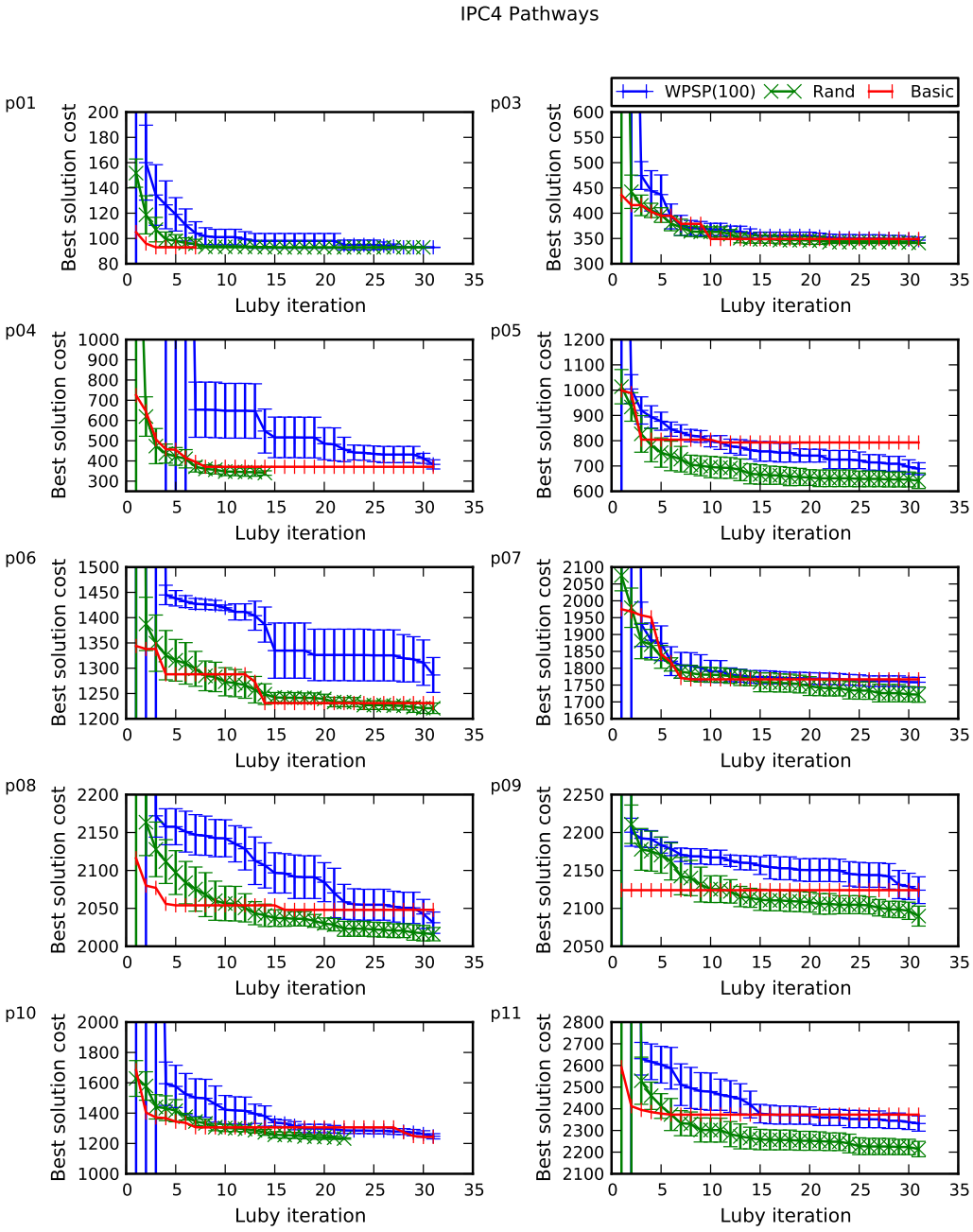


Figure 5.13: Solution trajectories for problems 1 and 3–11 from the Pathways domain (makespan = 8). Problems 1, 3, 4 and 10 were optimally solved by at least one run of the Rand strategy. Problem 1 was optimally solved by the Basic strategy.

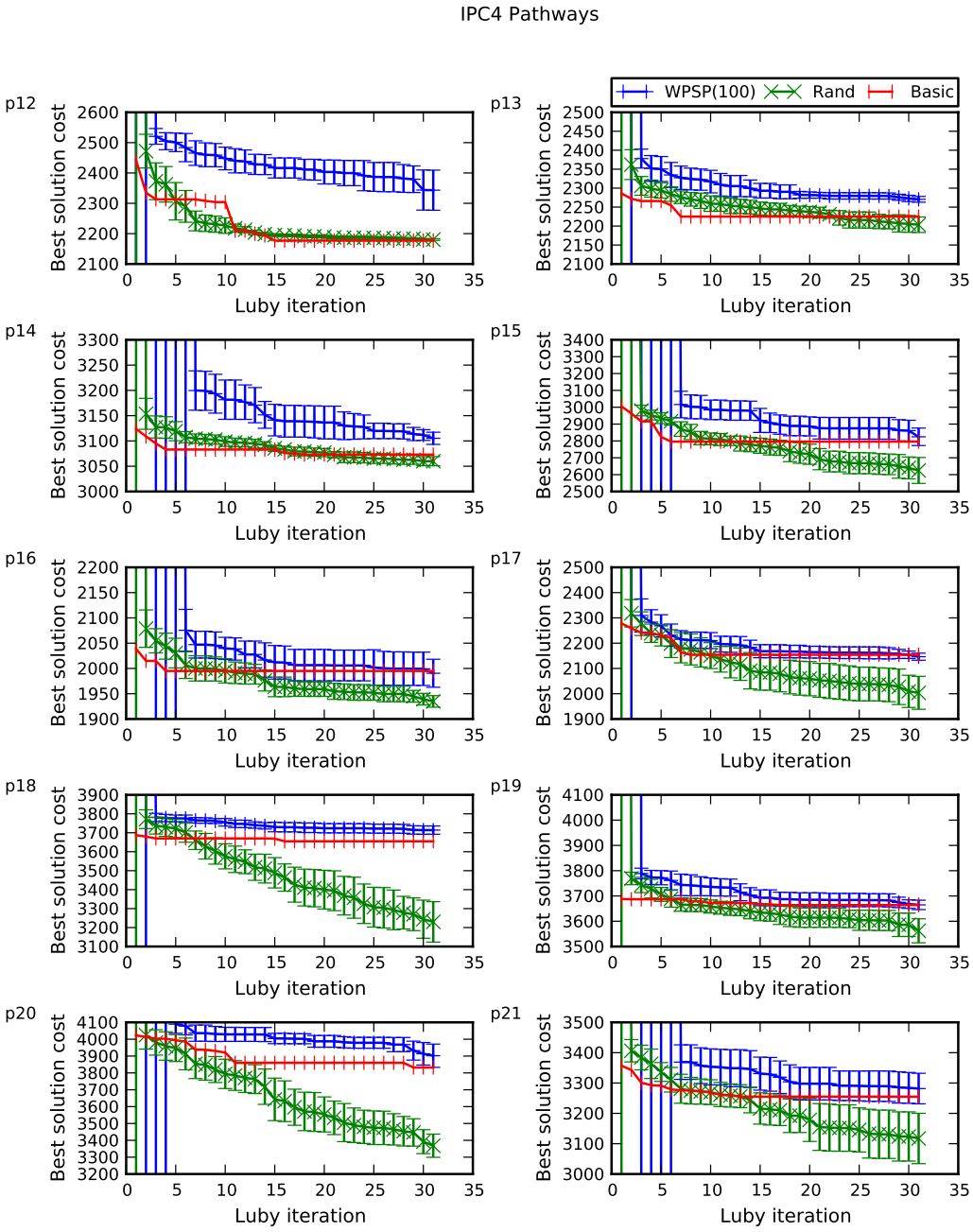


Figure 5.14: Solution trajectories for problems 12–21 from the Pathways domain (makespan = 8).

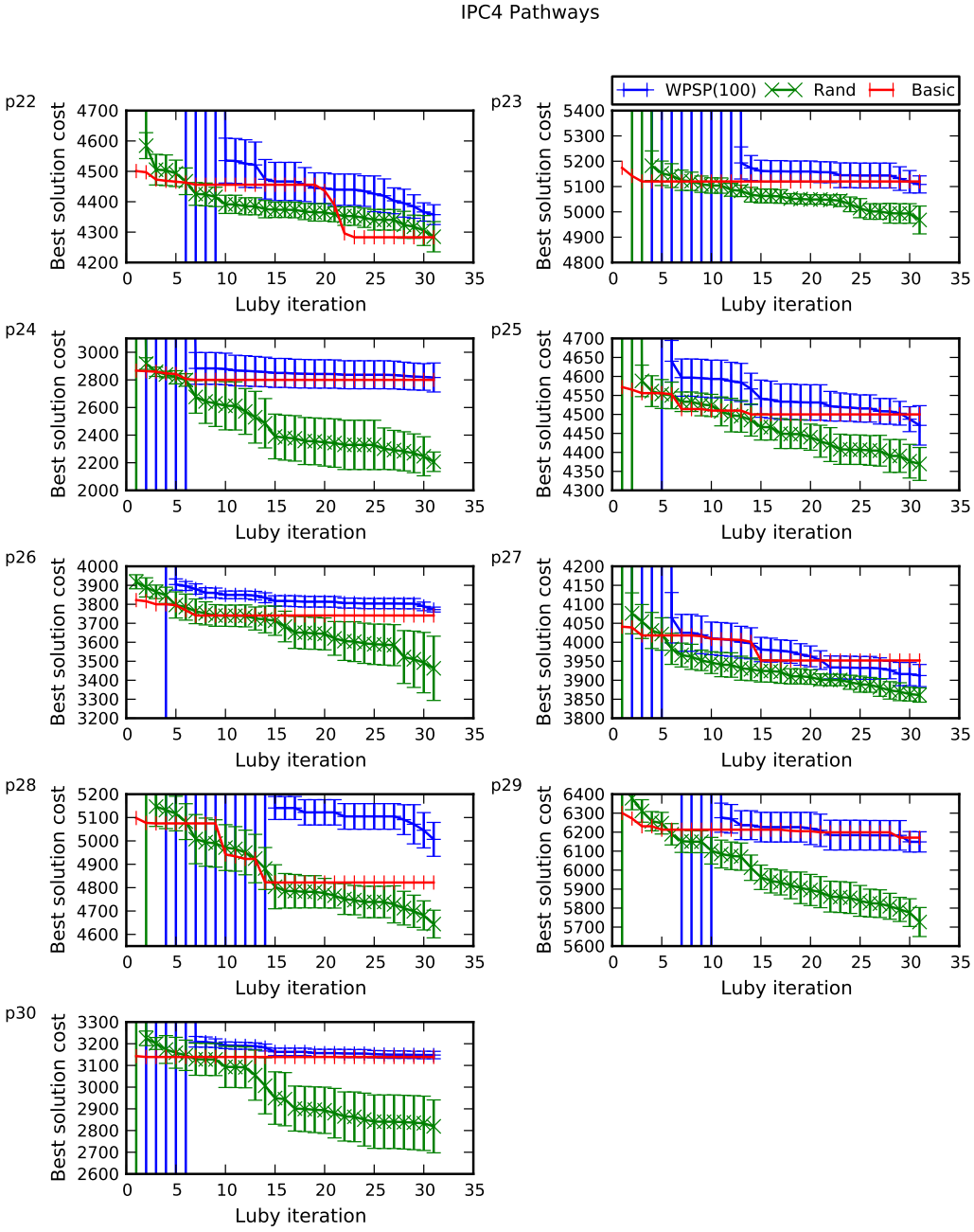


Figure 5.15: Solution trajectories for problems 22–30 from the Pathways domain (makespan = 8).

ZenoTravel

All three strategies are closer together in their average performance across problems in this domain. In the trajectory plots shown in Figure 5.17, we see that the WPSP(y) strategy shows statistically significant improvements over the other two strategies in problems 10 and 12, although it fails to complete all 31 iterations for at least one of the ten runs in each problem. WPSP(y) also shows difficulty in completing all 31 iterations in problems 6, 9, 11 and 13. This is partly due to the large encoding sizes: from Table 5.1 we see that problems 10–13 have more than 2,000 variables and 100,000 clauses each, which are among the largest encodings that we test. Such large encoding sizes result in longer computation times for iterations of message passing. The other cause for such long computation times is the difficulty in finding a fixed point of message passing; message passing is not converging within the iteration limits and repeated attempts are made to find a fixed point. Table 5.10 shows that the WPSP(y) strategy can fail to select a variable according to the WPSP(y) equations quite frequently for some problems, for example problems 5, 6, 9 and 13. This indicates that either convergence is not reached or for some fixed points, no variable has a sufficiently high bias to be chosen.

The WPSP(y) and Rand strategies show an advantage over the Basic strategy: they both find provably optimal solutions for four of the thirteen problems in at least one run, whereas the Basic strategy only produces an optimal solution for two problems.

	Trial (%)									
Problem	1	2	3	4	5	6	7	8	9	10
ZenoTravel										
01	100	100	100	88	100	100	100	100	100	100
02	84	90	69	53	56	66	61	67	81	57
03	99	100	100	100	99	99	100	99	100	100
04	99	100	100	99	100	100	100	100	100	100
05	45	47	49	55	47	55	53	47	36	47
06	88	83	85	83	86	86	84	85	89	84
07	96	96	95	96	94	95	96	95	93	97
08	100	100	100	100	100	100	100	100	100	100
09	72	76	71	74	75	74	73	75	73	71
10	81	79	74	78	76	74	84	83	74	76
11	100	99	99	99	99	99	99	99	99	99
12	90	95	96	93	94	95	93	96	91	95
13	53	53	47	48	55	55	43	51	51	49

Table 5.10: Percentage of successful selections performed by the WPSP(y) strategy for the ZenoTravel domain.

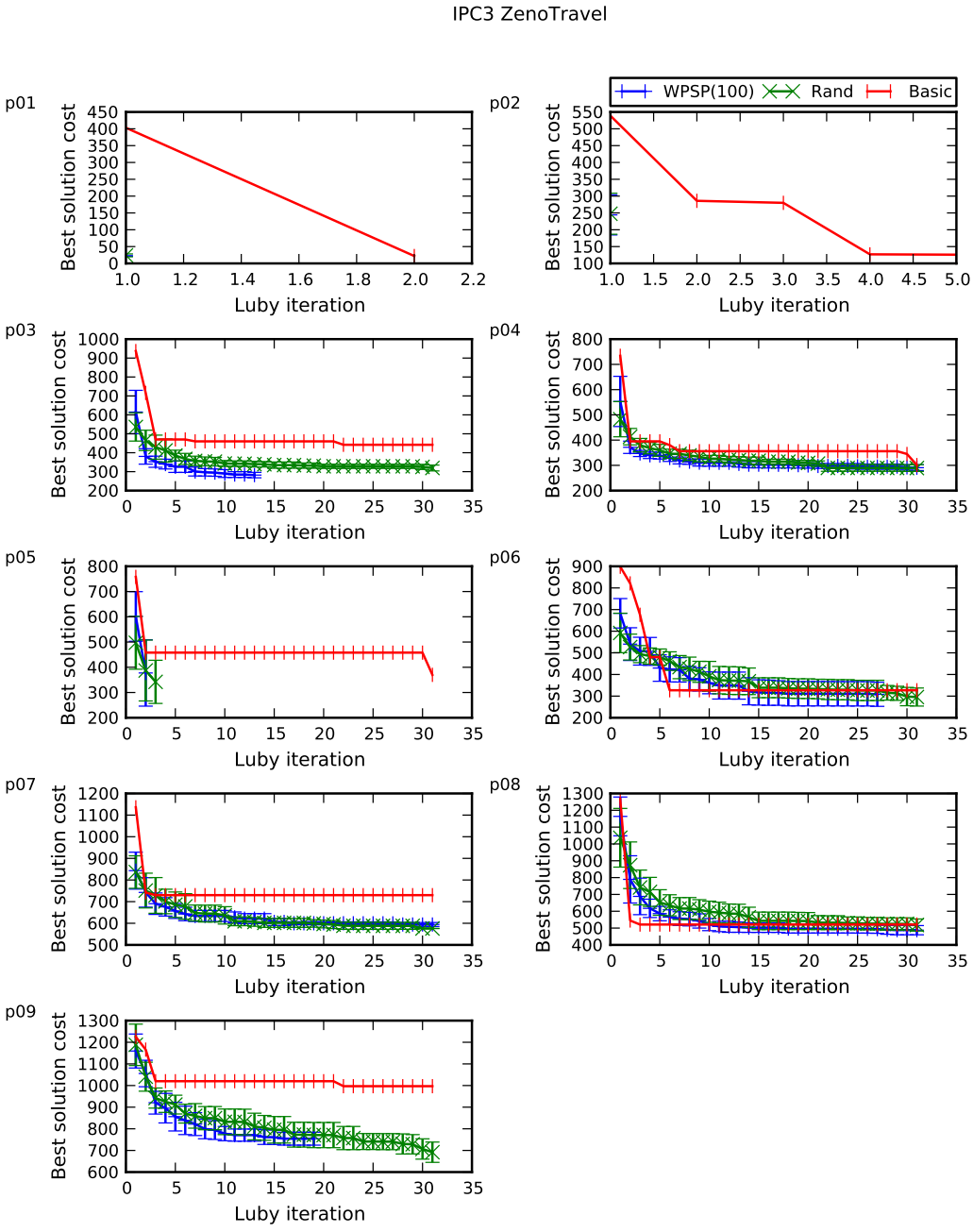


Figure 5.16: Solution trajectories for problems 1–9 from the ZenoTravel domain (makespan = 5). Problems 1–3 and 5 are optimally solved by at least one run of the WPSP(y) strategy. Problems 1, 2, 5 and 6 are optimally solved by at least one run of the Rand strategy. Problems 1 and 2 are optimally solved by the Basic strategy.

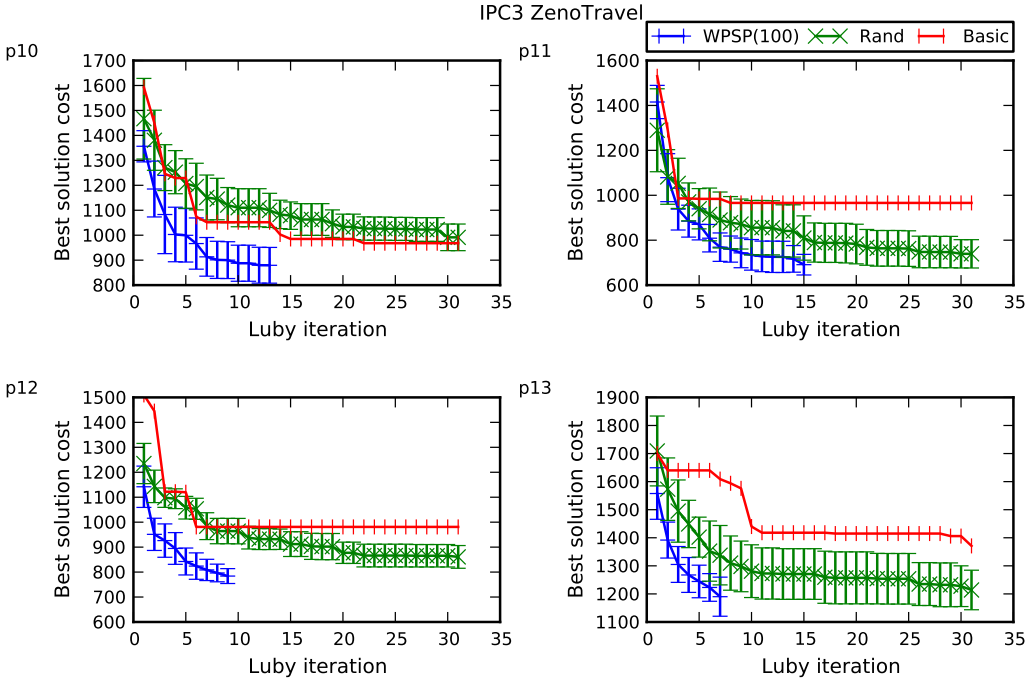


Figure 5.17: Solution trajectories for problems 10–13 from the ZenoTravel domain (makespan = 5).

5.3.4 Discussion

Our results illustrate the benefits of using randomisation with a restart strategy to mitigate the effects of making poor decisions during early decision levels. Problems such as p14 and p7 from Depot; p10 and p5 from DriverLog; p12 and p16 from Rovers; p18 and p30 from Pathways; p11 and p12 from ZenoTravel and p7 from Truck all show the Basic strategy struggling to improve upon its best solution as the number of restart iterations increase. This is despite it being allowed to search more of the tree as the number of allowed backtracks increases. This highlights the difference between what is achievable with the same limits on backtracking if different search paths are chosen during early decision levels.

For any non-trivial problem, all three strategies first attempt to set variables in the earliest decision levels to *True* before trying the assignment *False*. From decision levels 8 and onwards, the same strategy is employed to select the variable/value assignments. Part of the improvement of the Rand strategy over the Basic strategy might be due to restricting variable assignments to fact variables in the earliest decision levels, whereas the remaining difference is the introduction of randomness into the procedure. This randomness is introduced in the variable selection level rather than the value selection level: recall that an alternative version of VSIDS assigns to the selected variable the value *True* with probability 0.5 rather than always selecting *True*. Hence, it may be possible to improve the use of the VSIDS heuristic by selecting randomly from a group of the highest ranked variables.

In the DriverLog, Depot and Rovers domain the WPSP(y) strategy delivers the best trajectories of the three strategies. For almost all problems in those domains, it either matches the other strategies or improves upon them. The WPSP(y) strategy remains competitive in the ZenoTravel domain but only occassionally leads to lower cost solutions; however, it is encouraging that it does this for the harder problems in the domain.

For the Truck and Pathways domain, the WPSP(y) strategy displays the worst trajectories of the three strategies. Although the average performance of WPSP(y) eventually catches up with the other strategies by 31 restart iterations, the intention was to improve the performance during early restart iterations. In this respect, WPSP(y) has not been a success for these two domains. The results appear to suggest that WPSP(y) has difficulty in finding a satisfiable assignment of any cost in the earliest decision levels. This may be because the search is being directed into branches of the search tree that contain few solutions; however, it is more likely that the approximations used in the derivation of the WPSP(y) equations introduce a large amount of error. Developing a region-based survey propagation built on top of generalised belief propagation may help to reduce the error, but would be more expensive to compute.

Computation times

The WPSP(y) strategy is considerably more computationally expensive than the other strategies employed. Part of the difficulty is that the message passing equations are not guaranteed to converge when applied to factor graphs containing loops. If the search for a fixed point is proving to be difficult, eventually we must abandon message passing in order to make progress towards finding a solution.

We observed that when message passing is seen to converge, it often does so within a small number of iterations. To avoid choosing a set of initial messages that converge slowly or not at all, we incorporated randomised restarts. However, we have to specify the number of iterations allowed before a restart is made and the maximum number of restarts allowed. The choice of these parameters has a strong influence on the time taken to find a fixed point. Moreover, their optimum values may vary between problems.

An approximation that can be applied when it is difficult to find fixed points of message passing algorithms is the following. Run message passing for a prespecified number, n, of iterations, halting if convergence is reached. If convergence is not reached, then the messages that are obtained after n iterations are taken as an approximation to a fixed point and used to calculate marginals in the case of belief propagation, or biases in the case of survey propagation and its related methods[4].

We attempted this approach by implementing a new strategy AWPSP(y) that is an approximation of the WPSP(y) strategy. Like our other experiments, the AWPSP(y) strategy is applied only to decision levels < 8. Each time the AWPSP(y) is called to select a literal, it runs message passing for at most 20 iterations. If convergence is reached within those 20 iterations, we use messages from that fixed point; otherwise, we use the messages obtained at the end of 20 iterations. From these messages we calculate biases for all variables. We then select randomly from those fact variables v, such that $Bias(v) > 0.3$. If no such variable exists, then we use the Basic strategy for that variable selection. This variable is then set to *True* for that decision level.

We tested this strategy using the same methodology as described above on the domains for which WPSP(y) was most successful: Rovers, Depot and DriverLog. We used makespans of 5, 6 and 5, respectively. These are one less than the makespans used for the results presented above. We compared the trajectories for the AWPSP(y) strategy against the Rand and Basic strategies.

Figures 5.18 and 5.19 show the trajectories for problems from the Rovers domain. Figures 5.20 and 5.21 show the trajectories for problems from the Depot domain. Figures 5.22 and 5.23 show the trajectories for problems from the DriverLog domain.

[4]Note that loopy belief propagation is already an attempt to approximate marginals, so there are two levels of approximation in this approach.

Our results show that the AWPSP(y) strategy is still more effective than the Rand and Basic strategies at finding lower cost solutions earlier; however, the gap in performance appears to have narrowed. In part this is due to a lower percentage of successful variable selections made by the AWPSP(y) strategy. This occurs when no fact variables have a positive bias greater than 0.3. In decision levels where this happens, the AWPSP(y) strategy temporarily reverts to the Basic strategy for that variable selection. These results may not be as good as WPSP(y) strategy, but they may be of more practical relevance as they place bounds on the computational effort devoted to calculating biases.

Although message passing is expensive, it is only performed for a small number of variable selections, and so it may be worth bearing this extra expense if it leads to significant improvements. Moreover, in the cases where WPSP(y) provides significant improvements, our results suggest that it often does so within a small number of restart iterations of the WPMax-SAT solver. Consequently, it may not be necessary to continue to use WPSP(y) during later restart iterations, which would improve speed considerably. In particular, when we have found an optimal solution, all that remains to be done by the WPMax-SAT solver is to demonstrate that no better solution exists. It would be an interesting topic of future work to test whether the variable/value orderings suggested by the WPSP(y) strategy help to accomplish this 'optimality proof' within the WPMax-SAT solver more efficiently than the other strategies.

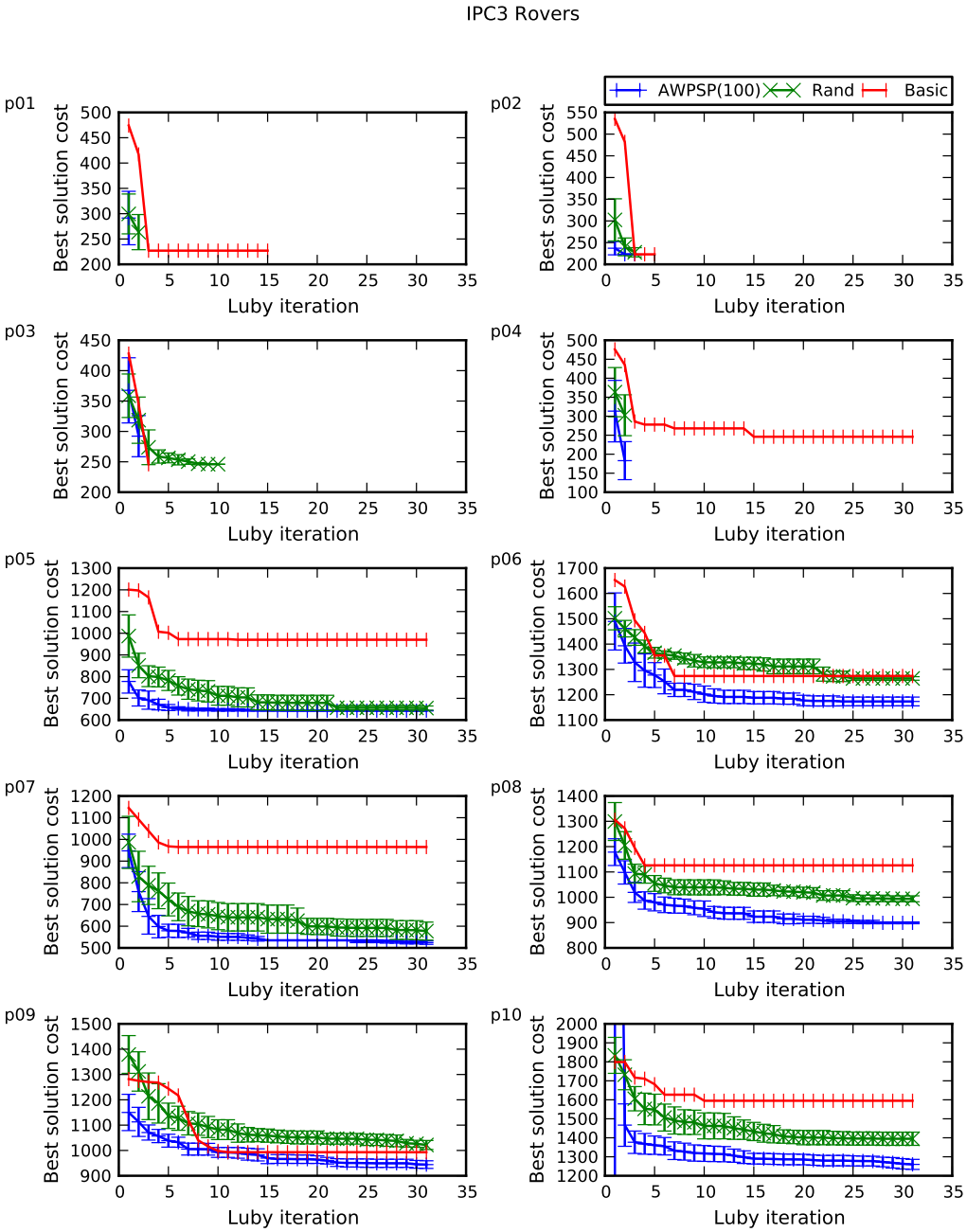


Figure 5.18: Solution trajectories for problems 1–10 from the Rovers domain (makespan = 5).

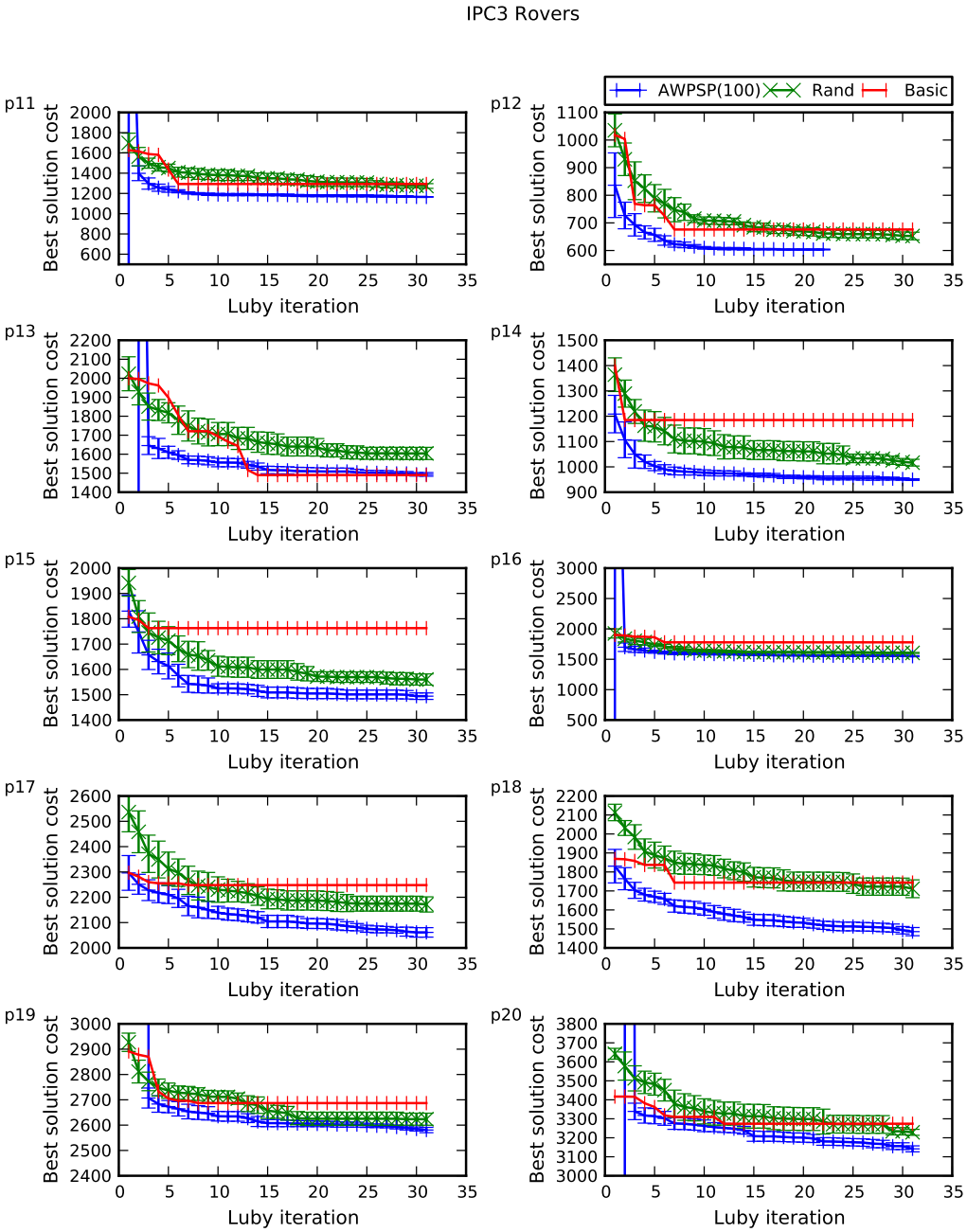


Figure 5.19: Solution trajectories for problems 11–20 from the Rovers domain (makespan = 5).

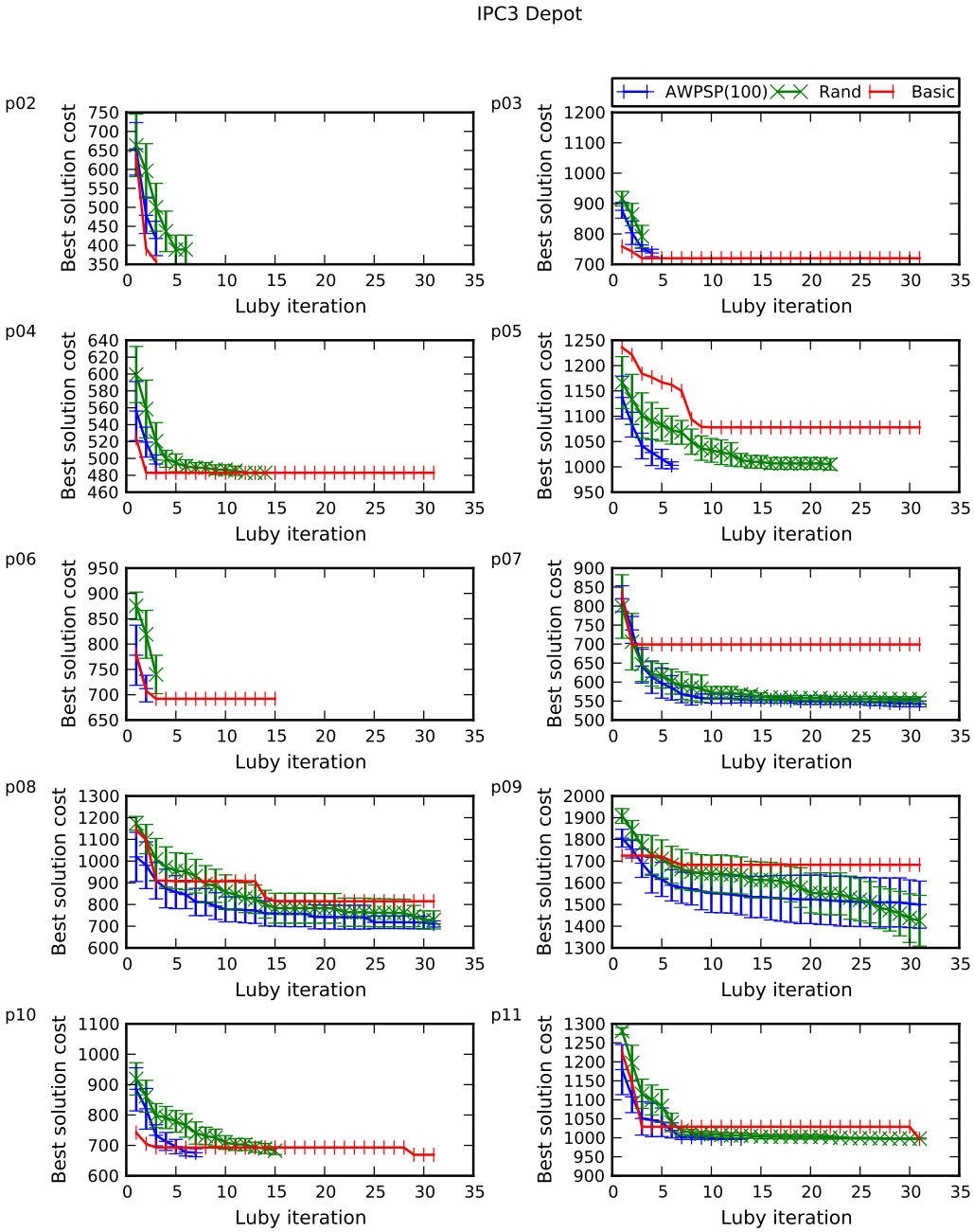


Figure 5.20: Solution trajectories for problems 2–11 from the Depot domain (makespan = 6).

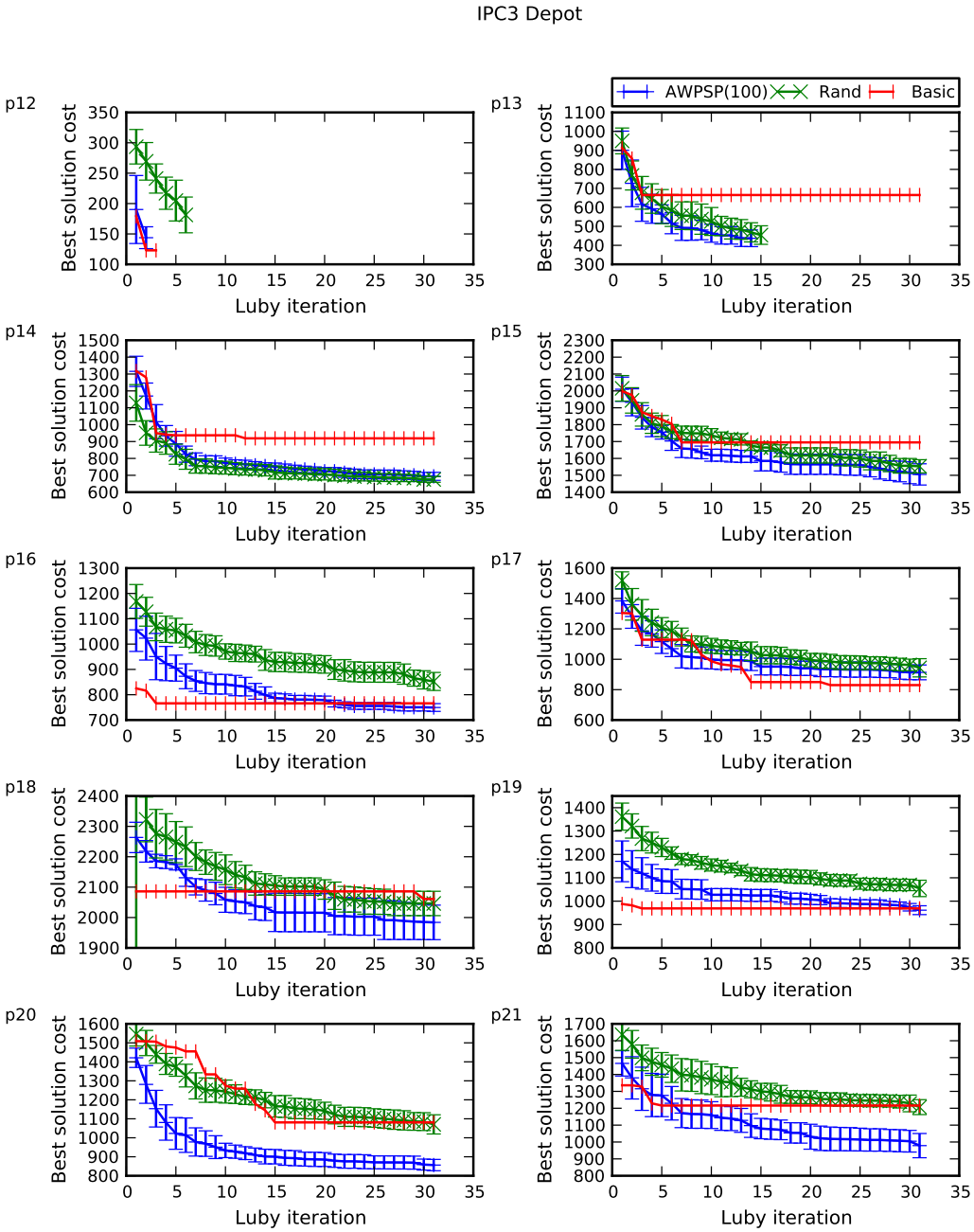


Figure 5.21: Solution trajectories for problems 12–21 from the Depot domain (makespan = 6).

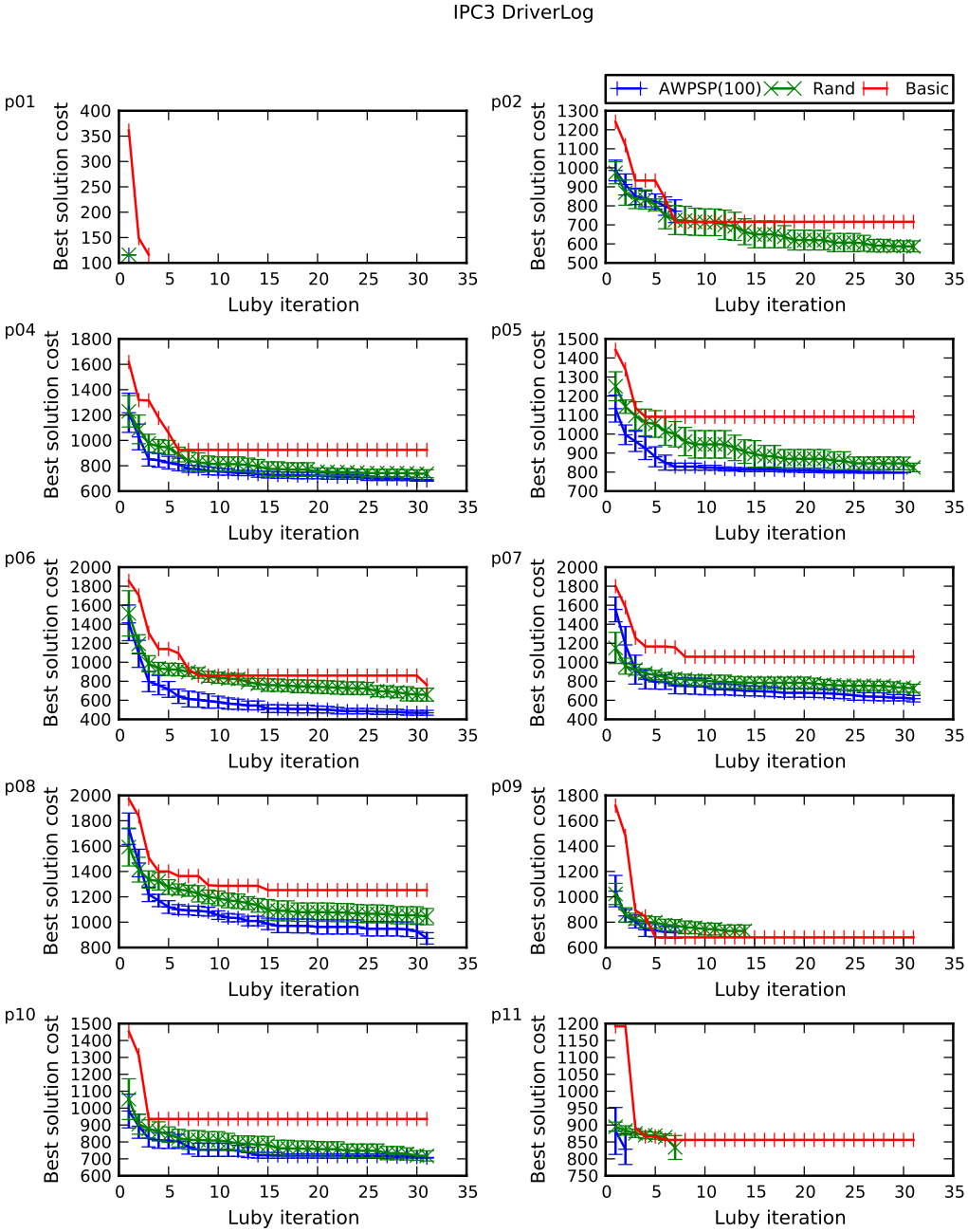


Figure 5.22: Solution trajectories for problems 1, 2 and 4–11 from the DriverLog domain (makespan = 5).

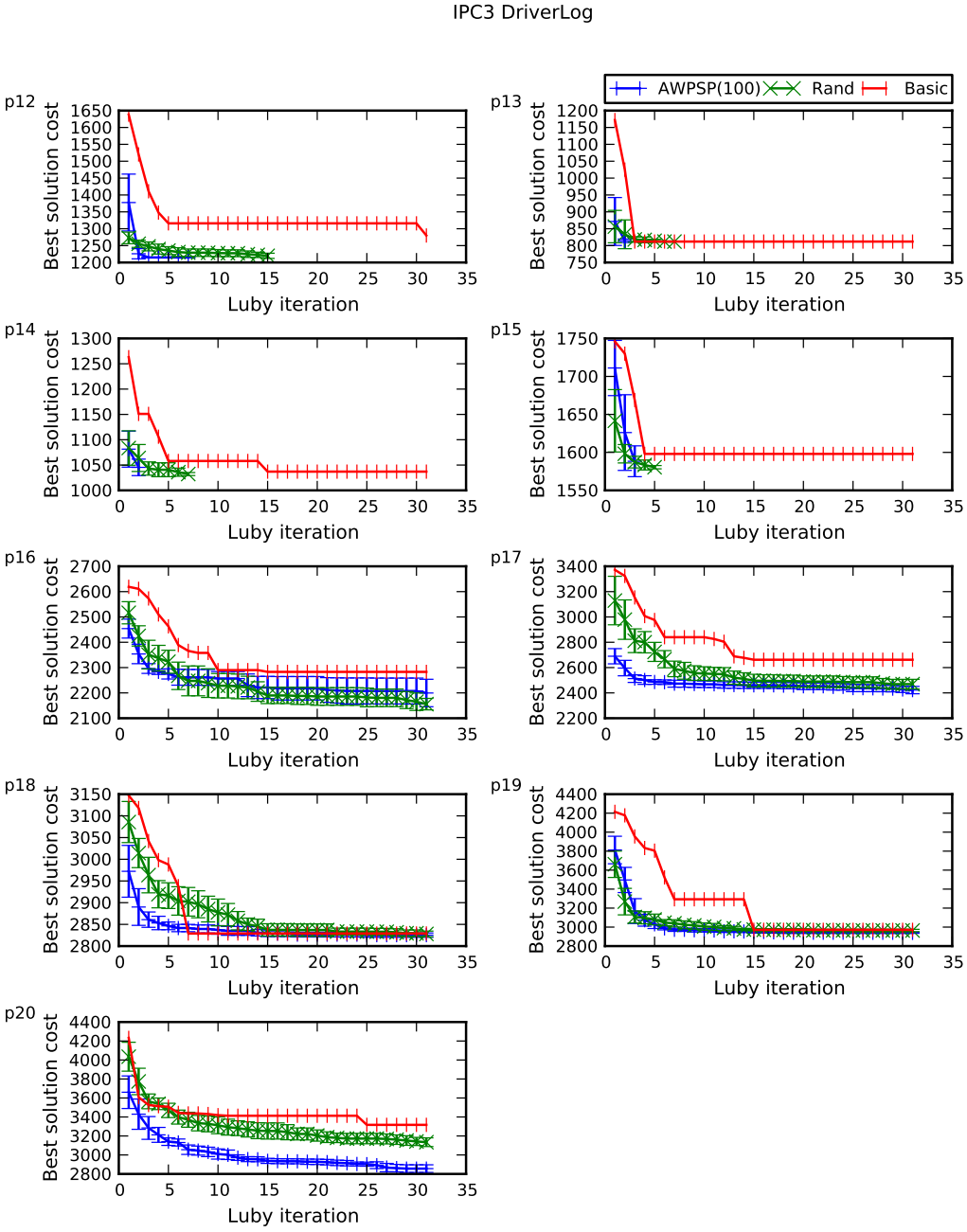


Figure 5.23: Solution trajectories for problems 12–20 from the DriverLog domain (makespan = 5).

5.4 Summary

We have reported two main sets of experimental results. First we compared the performance of a general purpose weighted partial Max-SAT solver (MiniMaxSat) against a mixed integer programming solver when applied to encodings of STRIPS planning problems with goal utility dependencies. We generated the test problems by adding randomly selected action costs and utility functions to problems taken from past International Planning Competitions. We found that the WPMax-SAT approach was competitive and often outperformed the integer programming approach despite the latter using the SAS$^+$ translation step that eliminates much redundancy in the state representation.

We then evaluated the effect of incorporpating bias estimates made by the WPSP(y) equations into a WPMax-SAT solver based upon the architecture of MiniMaxSat. The bias estimates were used as heuristic guidance for selecting decision literals in the first 7 decision levels of search. We compared this strategy to the default heuristic and a uniform random selection strategy. We found that for many problems in the DriverLog, Depot and Rovers domain our WPSP(y) strategy directed the search towards lower cost solutions earlier in the randomised restart search. This is an encouraging result that suggests the theory of survey propagation may give us new insights into practical problems and may have a broader application than the solution of random k-SAT formulae.

Chapter 6

Conclusion

We have demonstrated a system, MSatplan, for solving planning problems with goal utility dependencies using an optimisation variant of propositional satisfiability, known as weighted partial Max-SAT. This system is guaranteed to produce plans that are optimal up to a given makespan. We compared our implementation against a successful integer programming based encoding, implemented as iPUD/SYM, using past benchmark problems from the International Planning Competition. Our results showed that MSatplan is competitive with iPUD/SYM, solving as many as $109 \pm 8\%$ and $235 \pm 24\%$ more subproblems for the Truck and Depots domains; it also demonstrated consistently good performance on the Pathways domain. When iPUD/SYM outperformed MSatplan in our experiments, it did so by a smaller margin: $19 \pm 15\%$, $38 \pm 10\%$ and $3 \pm 3\%$ more problems solved for the DriverLog, ZenoTravel and Rovers domains.

We also applied the technique of survey propagation to the weighted partial Max-SAT encodings produced by the above method. To do this we derived a new set of equations, which we refer to as the WPSP(y) equations. These are a generalisation of the SP(y) equations from Max-SAT to weighted partial Max-SAT. We incorporated the WPSP(y) equations into a weighted partial Max-SAT solver based upon the MiniMaxSat system. Bias estimates from fixed points of the WPSP(y) equations were used to provide heuristic guidance during the first seven decision levels of search. We compared this strategy against uniform random selection from the same set of variables and the default VSIDS and weighted Jeroslow-Wang heuristic.

For the DriverLog, Depot and Rovers domains, we found that using these bias estimates to select variable/value assignments for the first seven decision levels reduced the average cost of the best solution found over many restart iterations of the search. This means that with the same limits on backtracking, using bias estimates directed the solver towards lower cost solutions than the other strategies on average. The method remained competitive for the ZenoTravel domain. For the Pathways and Truck domains, the bias estimates

directed the search towards higher cost solutions. One possible reason for this is that the approximations were inappropriate and introduced a large amount of error into the search.

It is worth remembering that message passing using the WPSP(y) equations is not guaranteed to converge when applied to WPMax-SAT encodings of STRIPS planning problems. We have demonstrated empirically that approximate fixed points of the WPSP(y) equations can be frequently found, through message passing, for encodings produced from several IPC benchmark domains.

Informally, the survey propagation method assumes that the min-sum equations admit many approximate fixed-point solutions. Plans potentially contain a lot of symmetry: pairs of sequences of actions that are independent - that is to say that they do not interact with one another by achieving or invalidating preconditions - may be executed in many different ways through different choices of how to interleave the sequences. Since plans correspond to solutions in our WPMax-SAT encodings, a solution to such an encoding may have many different variations, all of equal cost. It is also worth noting that our encodings contain variables that, for a certain collection of solutions, may be flipped without altering the cost or the satisfiability of the truth assignment; for example, a propositional variable corresponding to a NOOP action at level i may be safely flipped to create a new solution if the fact that the NOOP propagates is achieved at level i but is irrelevant to the rest of the plan from levels $i+1$ onwards.

In summary, it is likely that the solution space for our encodings is structured as follows. The solutions can be divided into several 'cost' levels, where each level contains many solutions of equal cost. Within a cost level, the solutions are grouped into clusters according to their Hamming distance[1]. Each cluster within a cost level corresponds to a particular choice of the levels in the planning graph that each action in the plan is executed in. Contained within each cluster are many solution variations that arise from single flips of variables that correspond to irrelevant details, such as NOOPs that are not required by the plan. This clustered solution space is likely to give rise to many different approximate fixed points of the WPSP(y) equations and may partly explain the success of the WPSP(y) strategy in many of our experiments. Future work might test this hypothesis by testing the WPSP(y) strategy on encodings of planning problems that are designed to not exhibit symmetry in their solutions. Extra axioms might be included in the encoding to eliminate the many variations on a solution that are possible in our current encoding scheme; for example, by asserting that every true fact is either deleted at the next level or is propagated by a NOOP. This should reduce the number of solutions that an encoding has. This might make it hard to find a non-trivial fixed point of the WPSP(y) equations

[1]The Hamming distance between two truth assignments is the number of variables which differ between the two assignments in their assigned value.

or adversely affect the performance of the WPSP(y) strategy.

Practically, we are not really interested in finding all the different symmetrical versions of a plan; instead, we only want to find a lowest cost solution. A clustered solution space, such as the one described above, can lead to an abundance of sub-optimal local minima. It is a well known problem that local search solvers often get trapped in such local minima. Further investigation into the behaviour of the WPSP(y) decimation strategy on encodings of planning problems may help us to gain insight into how to simplify the search space to eliminate such problematic local minima. Similarly, by taking symmetry into account, we may be able to design systematic solvers that operate in a reduced search space, which would reduce the time taken to complete optimality proofs.

Most research on survey propagation has studied its application to random k-SAT. This work demonstrates that the underlying method of survey propagation can be used for beneficial effect in more practical optimisation problems. Our results motivate further study of this method with the desire to make it more computationally practical and to increase its accuracy. Ensuring convergence and reducing the error that results from the factor graph containing loops would develop the work towards meeting those goals.

6.1 Future work

We obtained our results for MSatplan using a general-purpose weighted partial Max-SAT solver to find a plan. An area for future work is to investigate how we can specialize the algorithms used in these solvers to exploit the regular structure that is found in plan encodings.

6.1.1 Plan specific optimisation heuristics

Although the objective of automated planning is to develop domain-independent planning systems, which precludes the use of problem-specific heuristics, there appears to be a discrepancy between compilation and heuristic search approaches to planning. Compilation approaches often focus on creating *encodings* in the target language that model particular aspects of the planning problem. A state-of-the-art general purpose solver for that target language is then used to solve the problem. However, these solvers are designed to be general with respect to the target language, which means that they must be efficient across a broad range of problems, many of which will share no resemblance to encodings of planning problems.

In contrast, heuristic search approaches to planning focus on developing more efficient and accurate ways of extracting structured information from an arbitrary planning problem.

This information is then included in a heuristic to estimate the distance to the goal. These methods are general in the sense that they apply to any problem that can be expressed in the planning language, however they take advantage of typical behaviour and structure that is observed to be common across planning problems. For example, the *fast downward planner* exploits hierarchical structure that is often present in planning problems (Helmert 2006).

There is a significant gap in the literature surrounding the integration of ideas from heuristic search into satisfiability solvers, that are specialised to handle planning problems. In Chapter 2, we mentioned that the performance of planning as CSP benefitted from the inclusion of planning specific heuristics into the CSP solver. Rintanen's (2010) recent work on using planning specific heuristics in a SAT-based planner provides encouraging evidence that SAT-based planning can compete with and outperform a state-of-the-art heuristic search planner such as LAMA.

A general purpose WPMax-SAT solver, such as MINIMAXSAT1.0, that uses a branch-and-bound search must use a lower bounding function that is robust across a wide variety of problems. We believe that there is considerable potential for improving the performance of the paradigm we have set out in Chapter 3, by improving the lower bounding function to exploit planning-specific information. One suggestion for how this might be achieved is through the incorporation of landmarks.

Landmarks in planning

In planning, a *landmark* is a fact that is true at some time in every solution plan (Porteous et al. 2001). Although determining the set of landmarks for a planning problem is PSPACE-hard, the following sufficient condition can be used to identify some landmarks. For a fact F, construct the relaxed problem Π' which contains every action, ignoring their delete effects, from the original problem except those that have F as an add effect. If Π' does not have a solution that achieves the goals of the original problem then F is a landmark. A backward chaining procedure is performed on a relaxed planning graph in order to identify landmark candidates, which are then either confirmed or ignored by applying the above test to each candidate in turn.

The procedure that identifies landmark candidates first takes the set of goals as landmarks. It then collects all the achieving actions for those goals and takes the intersection of their preconditions to obtain new landmark candidates. This is recursively applied until all the preconditions are contained in the initial state. As landmarks are identified they are added as nodes to a directed tree using the ordering that was implied at their discovery: landmarks that were discovered as preconditions of supporting actions for other landmarks are ordered before them.

Porteous et al. then runs a forward state-space planner to plan for the leaf nodes of the tree, that is to say the landmarks that occur least in the ordering. Once these have been achieved they are removed from the tree and the process is reapplied to achieve the new leaf nodes of the tree starting from the end of the last plan. This continues until all landmarks (and hence goals) are achieved or failure is encountered. On success, the plans are concatenated to form one plan that transforms the initial state into a goal achieving state. The experimental evidence suggests that it reduces the time taken to find a solution at the expense of slightly longer plans since landmarks are planned for one layer at a time. Richter et al. (2008) adapted landmark generation to the SAS$^+$ model of planning and uses the generated landmarks as part of a heuristic in a sequential heuristic search planner, LAMA(Richter and Westphal 2008).

6.1.2 Improving lower bounds through landmarks

Landmarks can be used to derive admissible heuristics for cost-optimal planning in an A* search (Karpas and Domshlak 2009). The heuristic is computed for a state as follows. The set L of landmarks that remain to be achieved are identified together with the set A of actions that can achieve these landmarks. For each action $a \in A$ and landmark $\phi \in L$ that it could achieve, a cost $c(a, \phi)$ is associated with the pair. The cost $c(\phi)$ of a landmark $\phi \in L$ is then equal to the minimum $c(a, \phi)$ over actions a that can possibly achieve it. The costs $c(a, \phi)$ are constructed such that the sum $\sum_{\phi'} c(a, \phi') \leq c(a)$ where the sum is over the landmarks $\phi' \in L \cap Add(a)$. A simple implementation might divide the action cost equally amongst landmarks; the quality of the estimate can be improved, at an extra computational cost, by casting the equation as a linear program and solving for the optimisation function that maximises $\sum_{\phi \in L} c(\phi)$.

One can also identify landmark actions, which must occur in any solution plan. Obviously if an identified landmark action is not yet present in a plan up to the current state, then we are certainly required to execute it before we reach the goal; hence, the cost of that action should be added to the heuristic estimate for that state. To compute an admissible heuristic that combines both types of landmarks we can identify all such unperformed landmark actions A' and compute the set of yet to be achieved landmark facts $L' = L \setminus \bigcup_{a \in A'} Add(a)$ that are not achieved by any of the unperformed landmark actions. A heuristic estimate is then given by the sum of costs of actions in A' added with the heuristic estimate, as derived above, of achieving the landmarks in L'.

It may be possible to improve the lower bound on the cost of extending a partial truth assignment to a complete one through the use of landmarks. By considering the set of landmark facts and actions that have yet to be achieved and executed given the current

partial truth assignment, lower bounds on the costs of achieving and executing these facts and actions may yield tighter lower bounds on the true cost.

6.1.3 Inferential strength of landmarks in SAT

Up until recently, we were not aware of any satisfiability approach that included landmarks in its encodings. SatPlanLM (Cai et al. 2011), an entry in the recent 2011 International Planning Competition, uses landmarks generated by the Lama planner and their ordering constraints to add extra axioms to the planning encoding. We have not seen empirical results that explore the effects of including such axioms in the encoding.

One particular class of these ordering constraints are *necessary orderings*. A fact ψ occurs necessarily before a fact ϕ if in any plan where ϕ is made true at time step i, ψ is true at time step $i-1$. For this type of ordering, SatPlanLM adds a clause $\{\neg\phi_i, \phi_{i-1}, \psi_{i-1}\}$.

This clause is equivalent to the following three implications:

1. $\phi_i \wedge \neg\phi_{i-1} \Rightarrow \psi_{i-1}$.
2. $\neg\psi_{i-1} \wedge \neg\phi_{i-1} \Rightarrow \neg\phi_i$.
3. $\phi_i \wedge \neg\psi_{i-1} \Rightarrow \phi_{i-1}$.

If the encoding of the planning problem is the thin-gp encoding, taken from SatPlan, that we use, one can observe that the last two implications can be deduced from unit propagation. First note that necessary orderings are marked when ψ is a common precondition of all the non-NOOP supporting actions for ϕ. Now, for example, if $\neg\psi_{i-1} \wedge \neg\phi_{i-1}$ is true, then the NOOP a_{i-1}^{noop}, which preserves ϕ across time steps $i-1$ and i, must be false because a hard clause $\{\neg a_{i-1}^{noop}, \phi_{i-1}\}$ is present in the encoding. Similarly, all other non-NOOP supporting actions a_{i-1}^{j} of ϕ must be false because a hard clause $\{\neg a_{i-1}^{j}, \psi_{i-1}\}$ is present in the encoding. From the hard clause $\{\neg\phi_i, a_{i-1}^{noop}, a_{i-1}^{1}, \ldots, a_{i-1}^{k}\}$ that is present in the encoding, where ϕ has k supporting actions, unit propagation would deduce that $\neg\phi_i$ must be true, which is the conclusion of the second implication listed above.

A little thought will convince the reader that the first implication is the only one that provides a conclusion that is not possible by applying unit propagation to the thin-gp encoding alone. This is because inference gets as far as concluding that the clause $\{a_{i-1}^{1}, \ldots, a_{i-1}^{k}\}$ must be true, and unit propagation is not powerful enough to observe that no matter which a_{i-1}^{j} is true, there is a clause $\{\neg a_{i-1}^{j}, \psi_{i-1}\}$ (since ψ is a common precondition of all the non-NOOP supporting actions of ϕ) which will lead to the conclusion ψ_{i-1} regardless of which supporting action is chosen.

Observe that if we used Rintanen's (2008) unit propagation with look-ahead procedure, we would not need the axiom for necessary orderings at all. Observe that in the case of the first implication described above, when the unit propagation with look-ahead procedure probed the literal $\neg\psi_{i-1}$, the presence of clauses $\{\neg a^j_{i-1}, \psi_{i-1}\}$ would force a^j_{i-1} to be false for $j = 1, \ldots, k$. This would then transform the clause $\{a^1_{i-1}, \ldots, a^k_{i-1}\}$ into the empty clause. Hence, unit propagation with look-ahead would correctly determine that ψ_{i-1} is true, which is the conclusion of the first implication described above.

The above motivates the idea of strengthening the inferential power of a SAT solver to allow it to make such deductions without requiring those extra axioms to be included in the encoding. Furthermore, it would be interesting to find more theoretical results regarding which axioms add to the information that can be derived from unit propagation, in the same way that Sideris and Dimopoulos and Rintanen have done.

6.1.4 Similarity to probing

As we have suggested above, it seems that the technique of *probing* from the field of satisfiability shares some similarities with landmark discovery. Landmarks are often discovered by removing an action from the set of operators and checking if a relaxed plan still exists. Similarly, probing tries a value for a literal and sees if it can derive the empty clause by unit propagation from that assumption. If it can it knows the literal must take the opposite value if a satisfiable assignment is possible.

Perhaps the idea of probing could be generalised for encodings of planning problems. Given the layered nature of encodings and the way that fresh copies of facts and actions are included in each subsequent layer, instead of setting a single literal to a value, we could set to false, for a fact f, all the propositional variables $f_\ell, \ldots, f_n$ corresponding to that fact. If the empty clause can be derived from that assumption, then we know that at least one of those propositional variables must be true. From this, we know that the clause $\{f_\ell, \ldots, f_n\}$ must be satisfied. In essence, such a procedure has discovered a landmark fact.

Bibliography

D. Achlioptas and F. Ricci-Tersenghi. On the solution-space geometry of random constraint satisfaction problems. In *Proc. of the thirty-eighth annual ACM symposium on Theory of computing*, pages 130–139. ACM, 2006.

A. Armando, L. Compagna, and P. Ganty. SAT-based model-checking of security protocols using planning graph analysis. In *Proc. International Formal Methods Europe Symposium*, 2003.

F. Bacchus and A. Grove. Graphical models for preference and utility. In *Proc. UAI*, 1995.

C. Bäckström and B. Nebel. Complexity results for SAS^+ planning. *Computational Intelligence*, 11:625–655, 1995.

D. Battaglia, M. Kolář, and R. Zecchina. Minimizing energy below the glass thresholds. *Physical Review E*, 70(3):036107, 2004.

R.J. Bayardo and R.C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. AAAI*, 1997.

J. Benton, M. Do, and S. Kambhampati. Anytime heuristic search for partial satisfaction planning. *Artificial Intelligence*, 173:562–592, 2009.

A.L. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90:281–300, 1997.

B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2): 5–33, 2001.

C. Boutilier, R.I. Brafman, H.H. Hoos, and D. Poole. Reasoning with conditional ceteris paribus preference statements. In *Proc. UAI*, 1999.

C. Boutilier, F. Bacchus, and R.I. Brafman. UCP-Networks: A directed graphical representation of conditional utilities. In *Proc. UAI*, 2001.

R.I. Brafman and Y. Chernyavsky. Planning with goal preferences and constraints. In *Proc. ICAPS*, 2005.

R.I. Brafman, C. Domshlak, and S. Shimony. Introducing variable importance tradeoffs into cp-nets. In *Proc. UAI*, 2002.

A. Braunstein and R. Zecchina. Survey propagation as local equilibrium equations. *Journal of Statistical Mechanics: Theory and Experiment*, 2004:P06007, 2004.

A. Braunstein, M. Mézard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures & Algorithms*, 27(2):201–226, 2005.

T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

D. Cai, Y. Hu, and M. Yin. SatPlanLM and SatPlanLM-c: Using Landmarks and Their Orderings as Constraints. `http://ipc.icaps-conference.org/`, 2011.

P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In *Proc. IJCAI*, 1991.

H. Chen, C. Gomes, and B. Selman. Formal models of heavy-tailed behavior in combinatorial search. In *Proc. International Conference on Principles and Practice of Constraint Programming*, 2001.

Y. Chen, X. Zhao, and W. Zhang. Long-distance mutual exclusion for propositional planning. In *Proc. IJCAI*, 2007.

Y. Chen, Q. Lv, and R. Huang. Plan-a: A cost-optimal planner based on sat-constrained optimization. *Proc. of the sixth International Planning Competition*, 2008.

H.L. Chieu and W.S. Lee. Relaxed survey propagation for the weighted maximum satisfiability problem. *JAIR*, 36:229–266, 2009.

C. Coarfa, D.D. Demopoulos, A. San Miguel Aguirre, D. Subramanian, and M.Y. Vardi. Random 3-SAT: The plot thickens. In *Proc. International Conference on Principles and Practice of Constraint Programming*, 2000.

S.A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

J. Díaz, L. Kirousis, D. Mitsche, and X. Pérez-Giménez. On the satisfiability threshold of formulas with three literals per clause. *Theoretical Computer Science*, 410(30-32): 2920–2934, 2009.

M.B. Do and S. Kambhampati. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence*, 132(2):151–182, 2001a.

M.B. Do and S. Kambhampati. Sapa: A domain-independent heuristic metric temporal planner. In *Proc. ECP*, 2001b.

M.B. Do, J. Benton, M. van den Briel, and S. Kambhampati. Planning with goal utility dependencies. In *Proc. IJCAI*, 2007.

S. Edelkamp and M. Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. In *Proc. ECP*, 1999.

S. Edelkamp and J. Hoffmann. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, Universität Freiburg, Institut für Informatik, January 2004.

S. Edelkamp and P. Kissmann. Optimal symbolic planning with action costs and preferences. In *Proc. IJCAI*, 2009.

N. Eén and N. Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

M.D. Ernst, T.D. Millstein, and D.S. Weld. Automatic SAT-compilation of planning problems. In *Proc. IJCAI*, 1997.

T. Estlin, R. Castano, R.C. Anderson, D. Gains, B. Bornstein, C. De Granville, D. Thompson, M. Burl, and M. Judd. Automated targeting for the MER rovers. In *Proc. IEEE International Conference on Space Mission Challenges for Information Technology*, 2009.

R.E. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

M. Fox and D. Long. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *JAIR*, 20(1):61–124, 2003.

A. Gerevini and D. Long. Preferences and soft constraints in PDDL3. In *Proc. Workshop on Preferences and Soft Constraints in Planning (ICAPS)*, 2006.

E. Giunchiglia and M. Maratea. Planning as satisfiability with preferences. In *Proc. AAAI*, 2007.

C. Gomes, B. Selman, and N. Crato. Heavy-tailed distributions in combinatorial search. In *Proc. International Conference on Principles and Practice of Constraint Programming*, 1997.

C.P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proc. AAAI*, 1998.

M. Helmert. The fast downward planning system. *JAIR*, 26:191–246, 2006.

F. Heras and J. Larrosa. New inference rules for efficient Max-SAT solving. In *Proc. AAAI*, 2006.

F. Heras, J. Larrosa, and A. Oliveras. MiniMaxSAT: An efficient weighted Max-SAT solver. *JAIR*, 31:1–32, 2008.

J. Hoffmann and B. Nebel. The ff planning system: Fast plan generation through heuristic search. *JAIR*, 14(1):253–302, 2001.

E. Hsu and S. McIlraith. Characterizing propagation methods for boolean satisfiability. In *Proc. SAT*, 2006.

E. Hsu and S. McIlraith. VARSAT: Integrating Novel Probabilistic Inference Techniques with DPLL Search. In *Proc. SAT*, 2009.

J.S. Ide, F.G. Cozman, F.T. Ramos, et al. Generating random bayesian networks with constraints on induced width. In *Proc. ECAI*, 2004.

R.G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1(1–4):167–187, 1990.

Y. Jiang, H. Kautz, and B. Selman. Solving problems with hard and soft constraints using a stochastic algorithm for MAX-SAT. In *Proc. of the Intl. Joint Workshop on Artificial Intelligence and Operations Research*, 1995.

A.C. Kaporis, L.M. Kirousis, and E.G. Lalas. The probabilistic analysis of a greedy satisfiability algorithm. *Random Structures & Algorithms*, 28(4):444–480, 2006.

E. Karpas and C. Domshlak. Cost-optimal planning with landmarks. In *Proc. IJCAI*, 2009.

H. Kautz and B. Selman. Planning as satisfiability. In *Proc. ECAI*, 1992.

H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. AAAI*, 1996.

H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In *Proc. IJCAI*, 1999.

H. Kautz, D. McAllester, and B. Selman. Encoding plans in propositional logic. In *Proc. KR*, 1996.

H. Kautz, B. Selman, and J. Hoffmann. Satplan: Planning as satisfiability. In *Booklet of 5th IPC*, 2006.

R.L. Keeney and H. Raiffa. *Decisions with multiple objectives: Preferences and value tradeoffs*. Cambridge University Press, 1993.

E. Keyder and H. Geffner. Soft goals can be compiled away. *JAIR*, 36(1):547–556, 2009.

M. Klusch, A. Gerber, and M. Schmidt. Semantic web service composition planning with OWLS-Xplan. In *Proc. AAAI Fall Symposium on Agents and the Semantic Web*, 2005.

L. Kroc, A. Sabharwal, and B. Selman. Survey propagation revisited. In *Proc. UAI*, 2007.

F.R. Kschischang, B.J. Frey, and H.A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, 2001.

M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

I. Lynce and J. Marques-Silva. Efficient haplotype inference with boolean satisfiability. In *Proc. AAAI*, 2006.

D.J.C. MacKay. *Information theory, inference, and learning algorithms*, chapter 26, page 338. Cambridge University Press, 2003.

D.J.C. MacKay and R.M. Neal. Near Shannon limit performance of low density parity check codes. *Electronics letters*, 32(18):1645, 1996.

E. Maneva, E. Mossel, and M.J. Wainwright. A new look at survey propagation and its generalizations. *JACM*, 54(4), 2007.

J. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proc. Portugese Conference on Artificial Intelligence*, 1999.

D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proc. AAAI*, 1991.

D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. Pddl - the planning domain definition language. Technical report, Yale Center for Computational Vision and Control, 1998.

S.A. McIlraith, T.C. Son, and H. Zeng. Semantic web services. *Intelligent Systems*, 16 (2):46–53, 2001.

M. Mézard and A. Montanari. *Information, physics, and computation*. Oxford University Press, USA, 2009. ISBN 019857083X.

M. Mézard and R. Zecchina. Random k-satisfiability problem: From an analytic solution to an efficient algorithm. *Physical Review E*, 66(5):056126, 2002.

M. Mézard, G. Parisi, and R. Zecchina. Analytic and algorithmic solution of random satisfiability problems. *Science*, 297(5582):812–815, 2002.

D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proc. AAAI*, 1992.

M.W. Moskewicz, C.F. Madigan, Y Zhao, L. Zhang, and S Malik. Chaff: Engineering an efficient SAT solver. In *Proc. 38th Annual Design Automation Conference*, 2001.

K. Murphy, Y. Weiss, and M.I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proc. UAI*, 1999.

X.L. Nguyen, S. Kambhampati, and R.S. Nigenda. Planning graph as the basis for deriving heuristics for plan synthesis by state space and CSP search. *Artificial Intelligence*, 135 (1-2):73–123, 2002.

J. Pearl. Reverend Bayes on inference engines: A distributed hierarchical approach. In *Proc. AAAI*, 1982.

A. Pnueli. The temporal logic of programs. In *Proc. 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

J. Porteous, L. Sebastia, and J. Hoffmann. On the extraction, ordering, and usage of landmarks in planning. In *Proc. ECP*, 2001.

M.R. Prasad, A. Biere, and A. Gupta. A survey of recent advances in sat-based formal verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(2):156–173, 2005.

T. Ralphs and M. Guzelsoy. The SYMPHONY callable library for mixed integer programming. In *Proc. of the Conf. of the INFORMS Computing Society*, 2005.

S. Richter and M. Westphal. The lama planner using landmark counting in heuristic search. In *Proc. IPC*, 2008.

S. Richter, M. Helmert, and M. Westphal. Landmarks revisited. In *Proc. AAAI*, 2008.

J. Rintanen. Planning graphs and propositional clause learning. In *Proc. KR*, 2008.

J. Rintanen. Heuristics for planning with SAT. In *Proc. International Conference on Principles and Practice of Constraint Programming*, 2010.

J. Rintanen, K. Heljanko, and I. Niemela. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12-13):1031–1080, 2006.

J. Rintanen et al. Phase transitions in classical planning: an experimental study. In *Proc. KR*, 2004.

N. Robinson, C. Gretton, D.N. Pham, and A. Sattar. A compact and efficient SAT encoding for planning. In *Proc. ICAPS*, 2008.

N. Robinson, C. Gretton, D.N. Pham, and A. Sattar. Cost-optimal planning using weighted MaxSAT. In *Proc. ICAPS Workshop on Constraint Satisfaction Techniques for PLanning and Scheduling Problems*, 2010.

R. Russell and S. Holden. Handling goal utility dependencies in a satisfiability framework. In *Proc. ICAPS*, 2010.

S.J. Russell, P. Norvig, J.F. Canny, J.M. Malik, and D.D. Edwards. Planning. In *Artificial intelligence: a modern approach*, volume 74, page 377. Prentice hall Englewood Cliffs, NJ, 1995.

N. Schenker and J.F. Gentleman. On judging the significance of differences by examining the overlap between confidence intervals. *The American Statistician*, 55(3):182–186, 2001.

B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proc. AAAI*, 1994.

A. Sideris and Y. Dimopoulos. Constraint propagation in propositional planning. In *Proc. ICAPS*, 2010.

D. Smith. Choosing objectives in over-subscription planning. In *Proc. ICAPS*, 2004.

S.J.J. Smith, D.S. Nau, T.A. Throop, et al. Success in spades: Using AI planning techniques to win the world championship of computer bridge. In *Proc. IAAI*, 1998.

N. Sörensson and N. Eén. MiniSat 2.1 and MiniSat++ 1.0 – SAT Race 2008 Editions. `http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf`, 2009.

M.R. Spiegel and L.J. Stephens. *Schaum's outline of theory and problems of statistics*, chapter 11, page 275. McGraw-Hill, 2008.

P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. In *Proc. International Semantic Web Conference*, 2004.

G.S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 2(115-125):10–13, 1968.

M. Van Den Briel, R. Sanchez, M.B. Do, and S. Kambhampati. Effective approaches for partial satisfaction (over-subscription) planning. In *Proc. AAAI*, 2004.

M.H.L. van den Briel and S. Kambhampati. Optiplan: Unifying ip-based and graph-based planning. *JAIR*, 24:919–931, 2005.

M.H.L. van den Briel, T. Vossen, and S. Kambhampati. Loosely coupled formulations for automated planning: An integer programming perspective. *JAIR*, 31:217–257, 2008.

T. Vossen, M. Ball, A. Lotem, and D. Nau. On the use of integer programming models in ai planning. In *Proc. IJCAI*, 1999.

M. Welling and Y.W. Teh. Belief optimization for binary networks: A stable alternative to loopy belief propagation. In *Proc. UAI*, 2001.

J.S. Yedidia, W.T. Freeman, and Y. Weiss. Generalized belief propagation. In *Proc. NIPS*, 2000.

J.S. Yedidia, W.T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. In G. Lakemeyer and B. Nebel, editors, *Exploring artificial intelligence in the new millennium*, chapter 8, pages 239–270. Morgan Kaufmann Publishers, 2003.