Using Plan Decomposition for Continuing Plan Optimisation and Macro Generation

Fazlul Hasan Siddiqui

A thesis submitted for the degree of
DOCTOR OF PHILOSOPHY
The Australian National University

January 2016

Except where otherwise indicated, this thesis is my own original work.

Fazlul Hasan Siddiqui
25 January 2016

to my mother, Syeda Fazlunnessa Siddiqua

Acknowledgments

I am ever grateful to almighty Allah for his blessings, and for having made this work possible. This thesis, in fact, is a result of many people's contributions; my heartfelt appreciation goes to all of them for their valuable support and inspiration.

First and foremost I would like to extend my sincerest gratitude to my supervisor, **Dr. Patrik Haslum**. He is truly a spring of ideas and inspiration which exceptionally enhanced my motivation for research. He helped me happily every single time I needed him during this academic journey. I consider myself fortunate for having such an outstanding supervisor like Patrik; without his guidance and broad technical expertise this thesis would not have been possible. I also express my sincere appreciation to my advisors, **Dr. Phil Kilby** and **Dr. Charles Gretton**, for providing informative and valuable feedback throughout my doctoral study, which helped me significantly to enrich my knowledge in the field of my research. I owe my deepest gratitude to **Prof. Sylvie Thiébaux** for giving her inspirational advice at different stages of my doctoral study. I would also like to thank **Prof. Martin Müller** and **Dr. Hootan Nakhost** at University of Alberta, and **Dr. Enrico Scala** at NICTA for their valuable suggestions on my research problems that greatly helped me in shaping my research. Further, I am very grateful to **Dr. Lukas Chrpa** at University of Huddersfield for his direct involvement in my doctoral work, and for his exceptional help and inspiration to enrich this thesis. My special thanks to all the anonymous reviewers of our different publications for their insightful comments and suggestions that greatly aided to improving the final version of this thesis.

During this endeavour, I was fortunate to be able to work with my talented fellow students from ANU and NICTA. Special thanks to Mr. Mohammad Abdulaziz, Mr. Suvash Sedhain, Mr. Paul Scott, Mr. Karsten Lehmann, Mr. Terrence Mak, Mr. Alan Lee, and Ms. Kate Quenzer for rendering their friendliness and support. I would also like to acknowledge the Australian National University, the Australian Government, and NICTA who supported me financially that enabled me to focus on my doctoral research. I also owe my gratitude to my friends Dr. Wayes Tushar, Dr. Sejuti Rahman, Dr. Rifat Shahriyar, Dr. Md Masud Hasan, Dr. Abu Barkat Ullah, Mr. Ahmadullah Sadi, Mr. Mohsin Ali, Mr. Zakir Hossain, Ms. Ifa Islam, Ms. Shakila Khan Rumi, and Ms. Imun Ruby who made my stay in Australia very enjoyable and colourful.

Last, but certainly not the least, I would like to express my deepest gratitude to my parents Mr. Mokhtar Ali Gazi and Mrs. Syeda Fazlunnessa Siddiqua, my brother Mr. Sakib Hasan Siddiqui, and my sister Ms. Kaniz Fatema for their unconditional love and support. My special thanks to my beloved wife Ms. Nevis Wadia for her ceaseless sacrifice and mental support, and my adorable sons Afeef Ayman and Abeed Ayman for always cheering me up with their sparkling smiles.

Abstract

This thesis addresses three problems in the field of classical AI planning: decomposing a plan into meaningful subplans, continuing plan quality optimisation, and macro generation for efficient planning. The importance and difficulty of each of these problems is outlined below.

(1) Decomposing a plan into meaningful subplans can facilitate a number of post-plan generation tasks, including plan quality optimisation and macro generation – the two key concerns of this thesis. However, conventional plan decomposition techniques are often unable to decompose plans because they consider dependencies among steps, rather than subplans.

(2) Finding high quality plans for large planning problems is hard. Planners that guarantee optimal, or bounded suboptimal, plan quality often cannot solve them In one experiment with the Genome Edit Distance domain optimal planners solved only 11.5% of problems. Anytime planners promise a way to successively produce better plans over time. However, current anytime planners tend to reach a limit where they stop finding any further improvement, and the plans produced are still very far from the best possible. In the same experiment, the LAMA anytime planner solved all problems but found plans whose average quality is 1.57 times worse than the best known.

(3) Finding solutions quickly or even finding any solution for large problems within some resource constraint is also difficult. The best-performing planner in the 2014 international planning competition still failed to solve 29.3% of problems. Re-engineering a domain model by capturing and exploiting structural knowledge in the form of macros has been found very useful in speeding up planners. However, existing planner independent macro generation techniques often fail to capture some promising macro candidates because the constituent actions are not found in sequence in the totally ordered training plans.

This thesis contributes to plan decomposition by developing a new plan deordering technique, named block deordering, that allows two subplans to be unordered even when their constituent steps cannot. Based on the block-deordered plan, this thesis further contributes to plan optimisation and macro generation, and their implementations in two systems, named BDPO2 and BloMa. Key to BDPO2 is a decomposition into subproblems of improving parts of the current best plan, rather than the plan as a whole. BDPO2 can be seen as an application of the large neighbourhood search strategy to planning. We use several windowing strategies to extract subplans from the block deordering of the current plan, and on-line learning for applying the most promising subplanners to the most promising subplans. We demonstrate empirically that even starting with the best plans found by other means, BDPO2 is still able to continue improving plan quality, and often produces

better plans than other anytime planners when all are given enough runtime. BLOMA uses an automatic planner independent technique to extract and filter "self-containe" subplans as macros from the block deordered training plans. These macros represent important longer activities useful to improve planners coverage and efficiency compared to the traditional macro generation approaches.

List of Publications

(a) Siddiqui, F.H. & Haslum, P., 2015. Continuing Plan Quality Optimisation. Journal of Artificial Intelligence Research (JAIR), 54(nov 2015), 369–435. AAAI Press. Available at: http://jair.org/media/4980/live-4980-8969-jair.pdf.

(b) Chrpa, L. & Siddiqui, F.H., 2015. Exploiting Block Deordering for Improving Planners Efficiency. In Proc. of the 24th International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015, 1537–1543. AAAI Press.
Available at: http://ijcai.org/papers15/Papers/IJCAI15-220.pdf.

(c) Siddiqui, F.H. & Haslum, P., 2013. Plan Quality Optimisation via Block Decomposition. In Proc. of the 23rd International Joint Conference on Artificial Intelligence, IJCAI 2013, Beijing, China, August 3-9, 2013, 2387–2393. AAAI Press.
Available at: http://ijcai.org/papers13/Papers/IJCAI13-351.pdf.

(d) Siddiqui, F.H. & Haslum, P., 2013. Local Search in the Space of Valid Plans. In Proc. of the 2013 ICAPS Workshop on Evolutionary Techniques in Planning and Scheduling, EVOPS 2013, Rome, Italy, June 10-14, 2013, 22–31.
Available at: http://icaps13.icaps-conference.org/wp-content/uploads/2013/05/evops13-proceedings.pdf.

(e) Siddiqui, F.H. & Haslum, P., 2012. Block-Structured Plan Deordering. In Proc. of the 25th Australasian Joint Conference on Advances in Artificial Intelligence, AI 2012, Sydney, Australia, December 4-7, 2012, vol. 7691 of Lecture Notes in Computer Science, 803–814. Springer, Berlin, Heidelberg.
Available at: http://dx.doi.org/10.1007/978-3-642-35101-3_68.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The classical planning problem in the field of Artificial Intelligence (AI) involves representing models (i.e., initial and goal states) of the world and available actions in some formal modelling language, and reasoning about the preconditions and effects of the actions. Given a planning problem, a planning system (or planner, for short) generates a plan consisting of a sequence of actions, whose application transforms the world from the initial state to a desired goal state. Thus, through constructing and executing the plan of actions, planning makes a system intelligent, autonomous, and able to attain its goals.

This thesis focuses on the challenges and opportunities of three important problems in the field of classical AI planning: (1) decomposing a plan into meaningful subplans, (2) continuing plan quality optimisation, and (3) macro generation for efficient planning. To address these planning problems, first, we develop a novel technique, termed *block deordering*, that decomposes a given plan into coherent subplans. Afterwards, we develop two new systems separately, each of which is based on the plan decomposition, to achieve the continuing plan optimisation and macro generation for efficient planning. In the rest of this chapter, we briefly describe the challenges, motivations, and our contributions to solving these problems.

1.1 Plan Decomposition

A plan can be totally or partially ordered. A totally ordered (also known as sequential) plan allows one single order of execution of its constituent steps to achieve the goal, which is the plan itself. In contrast, steps in a partially ordered plan can be unordered (i.e., independent) w.r.t. each other, which allows them to be executed in any order consistent with the partial order and still achieve the goal. This flexibility of execution allows partially ordered plans to be scheduled for improved efficiency or robustness [Policella et al., 2004]. However, the current state space search planners that produce totally ordered sequential plans are far more efficient than the older partial order planners.

Deordering is a process that converts a sequential plan into a partially ordered plan by removing ordering constraints between steps. Thus, deordering plays a useful role in that it enables more efficient generation of partially ordered plans than the

partial order planners. The standard interpretation of a partially ordered plan, however, restricts deordering to only the cases where individual steps are independent and non-interfering. This means that for two subplans (i.e., two subsequences of the plan) to be unordered, every interleaving of steps from the two must form a valid execution. In other words, every sequential plan that is a topological sort of the steps has to be valid (according to the semantics of sequential plan execution).

We discover that decomposing a plan into non-interleaving subplans eliminates the above restriction on plan deordering, and allows the plan to be deordered to a much greater degree than the existing step-wise deordering techniques. As a result, it provides better flexibility in plan execution. Decomposing a (totally or partially ordered) plan, in general, implies breaking the plan into subplans, where each subplan is a subsequence of some linearisation of the plan. We have seen in our experiments that decomposing a plan into "meaningful" subplans can be helpful for improving plan quality and planners' efficiency. A more meaningful subplan is relatively more coherent, more encapsulated, less interfering with the other parts of the plan, and often represents a purposeful activity (e.g., "delivering a package" in the Transport domain).

1.1.1 Motivation and Challenges

Plan decomposition can be useful for breaking unnecessary orderings between subplans, which facilitates further analysis of the plan such as: a) explaining the plan to a user, b) reducing the coordination overhead among subplans if the plan is to be executed by a distributed team of agents, c) improving plan quality by local modifications, or d) macro generation for improving planners' performances. We are particularly concerned with the last two reasons: plan quality optimisation and macro generation, which is achievable through finding and utilizing meaningful subplans.

Finding a useful plan decomposition that can achieve the above targets, however, is challenging. Simply taking a subsequence of a totally ordered plan often forms a subplan that is of little or no importance. For example, we have seen in our experiments (described in Section 3.3) that more than 75% of the subproblems for which we find an improved subplan correspond to a non-consecutive part of the sequential input plan. Furthermore, such a random subsequence of a plan often is not meaningful; therefore, the corresponding decomposition does not allow the plan to deorder.

1.1.2 Contributions to Plan Decomposition

We develop a new form of plan decomposition, named *block decomposition*, that helps to eliminate the restriction on deordering, imposed by the standard interpretation of a partially ordered plan. In block decomposition, we use a conservative notion of partial ordering: we divide a plan into coherent subplans (often consisting of non-consecutive steps of the plan), called *blocks* [Siddiqui and Haslum, 2012], such that the steps in a block may not be interleaved with steps outside the block, but un-

ordered blocks can be executed in any sequence. The restriction to non-interleaved executions allows "transient" dependencies and effects to be encapsulated within a block, and thus not cause interference with other blocks. This relaxes ordering constraints between blocks, and thus enables deordering of blocks in some cases also when their constituent steps cannot be deordered under step-wise interpretation of partially ordered plans. The process of deordering a plan into a block decomposed partially ordered plan is called *block deordering*. We present a block deordering algorithm, and show empirically that it deorders plans to a much greater degree than the conventional step-wise deordering techniques in most domains.

Attaining enhanced deordering by block-structured plan decomposition is not an end in itself. Such decomposition can join together necessary (often non-consecutive) steps of the plan into meaningful subplans (by removing unnecessary orderings from the plan), which helps in plan quality optimisation and macro generation. Furthermore, plan optimisation and macro generation often require a large set of candidate subplans which are easier to generate by the block decomposed partially ordered plan than the traditional partially ordered or sequential plan.

1.2 Continuing Plan Quality Optimisation

One of the key concerns in automated planning is producing high quality plans. Planners using optimal or bounded suboptimal (heuristic) search methods guarantee the plan quality, but they are far from able to solve large problems. Fast planners, using a greedy heuristic search or other techniques, on the other hand, can solve large problems but often find poor quality plans. The gap between the capabilities of these two kinds of planners means that producing high quality plans for large problems is still a challenge. An example of this gap is shown in Figure 1.1. Much progress has been made separately on those two targets – solution quality and solver efficiency, but very little on combining the two, in order to reduce the optimality gap. Our interest in addressing this gap is to build a system that can perform a continuing improvement of the plan quality at varying time scales.

1.2.1 Challenges in Reducing the Optimality Gap

Anytime search tries to strike a balance between the optimal (or bounded suboptimal) and greedy heuristic search methods. Anytime search algorithms do so by finding an initial solution, possibly of poor quality, quickly and then continuing to search for better solutions the more time they are given. Anytime search algorithms such as, for example, RWA* [Richter et al., 2010] or AEES [Thayer et al., 2012b] have been successfully used for anytime planning. However, these planners are often not effective at making use of increasing runtime beyond the first few minutes. Xie et al. [2013] define the "unproductive time" of a planner as the amount of time remaining when a planner finds its best plan, out of the total time given. They show that in four IPC-2011 domains (Barman, Elevators, Parcprinter, and Woodworking), the unproductive time of the LAMA planner (which uses RWA*), given 30 minutes per

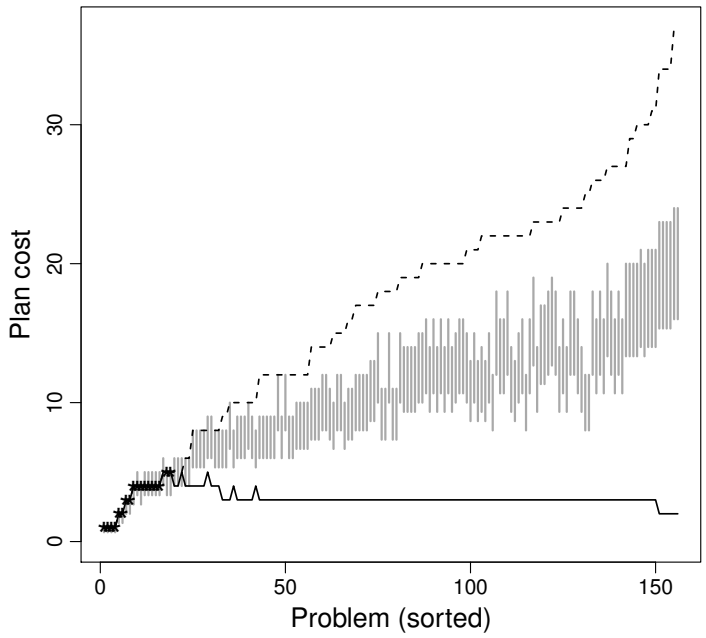


Figure 1.1: Illustration of the plan quality gap. The dashed line represents the best (lowest-cost) plan for 156 problems from Genome Edit Distance (GED) domain [Haslum, 2011] found by different non-optimal planners, including anytime planners. The solid line represents the corresponding highest known lower bound. The difference between these two is the optimality gap. The '$\star$' points represent plans found by optimal planners, while the vertical bars show the optimality gap obtained by a problem-specific algorithm (GRIMM).

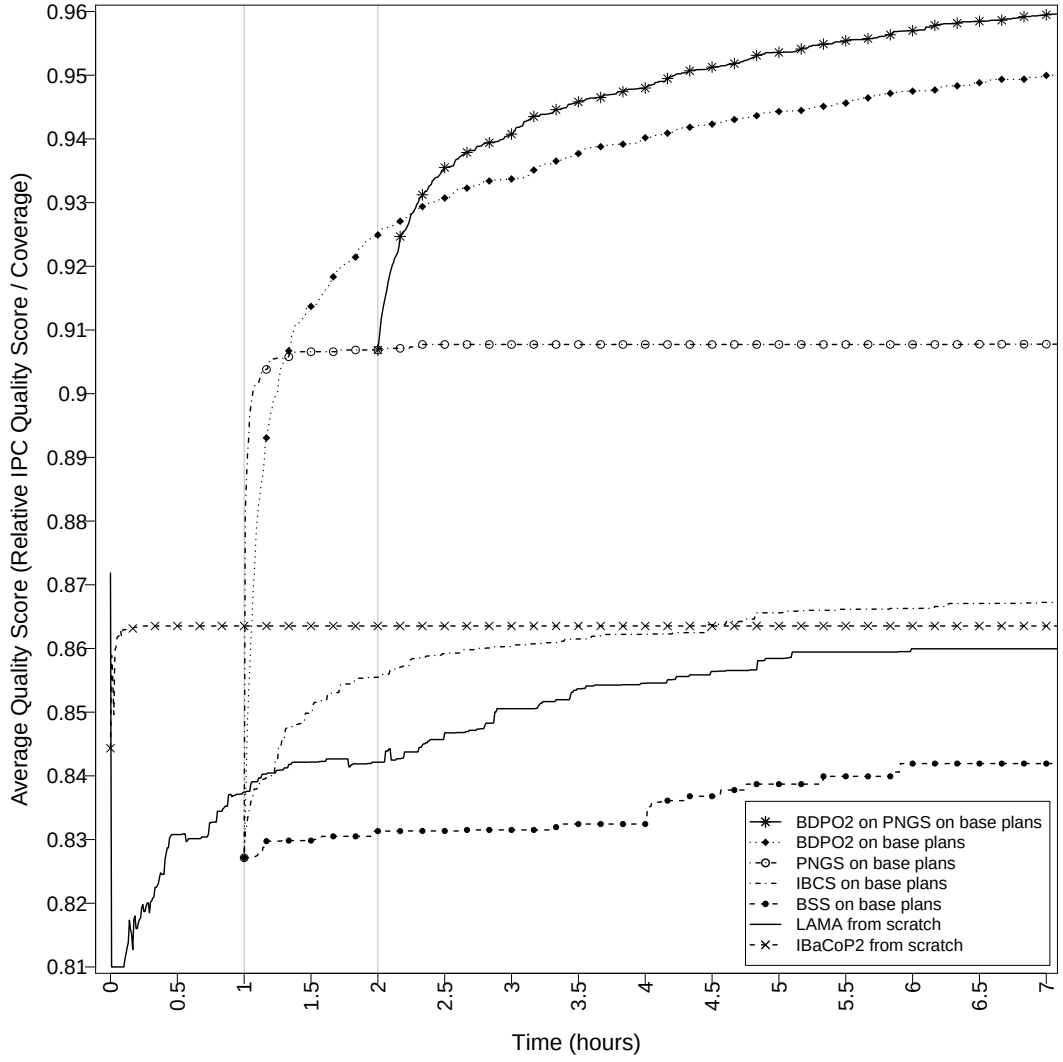


Figure 1.2: Average IPC quality score as a function of time per problem, on a set of 182 large-scale planning problems. The quality score of a plan is c^{ref}/c, where c is the cost of the plan and c^{ref} the "reference cost" (least cost of all plans for the problem); hence a higher score represents better plan quality. Anytime planners (LAMA, IBaCoP2) start from scratch, while post-processing (PNGS, BDPO) and bounded-cost search (IBCS, Beam-Stack Search) methods start from a set of base plans. Their curves are delayed by 1 hour to account for the maximum time given to generating each base plan. The experiment setup and results for additional planners are described in Section 3.3 (page 42).

problem, is more than 90%. We have observed similar results, as shown in Figure 1.2. The figure shows the average IPC quality score as a function of time for several anytime planners and plan optimisation methods, including the LAMA planner. (A full description of the experiment setup, and results for even more anytime planners, is presented in Chapter 3, from page 37.) LAMA is fast at finding a first solution; for 92.3% of the problems it solves (within a 7 hour time limit), the first plan is found in less than 10 minutes. The quality of LAMA's plans improve rapidly early on, but the trend after 30 minutes is one of "flattening out", i.e., decreasing increase. The big drop at the beginning is due to the figure showing average quality: as initial, low-quality, plans for larger problems are found, the average drops, before increasing again as better plans are found. Between 1 and 7 hours CPU time, LAMA improves the plans of 21.3% of solved problems. Yet, for a further 51.6% of the problems, better plans have been found by other methods. In the same time interval, LAMA's average quality score increases by only 2.8%, while an increase of at least 11.9% is still possible. IBaCoP2 [Cenamor et al., 2014] (described in Section 3.10.4), the best performing planner in the 2014 IPC, is a portfolio-based planning system (described in Section 3.10.4) that runs several subplanners in sequence with short timeouts. IBaCoP2 is very quick at finding improvements (as shown in Figure 1.2), because of using the subplanner having full strength to solve the current problem. However, this planner also stagnates and then quickly runs out of memory. Iterated bounded-cost search (IBCS) (as described in Section 3.6) and Beam-Stack Search (BSS) [Zhou and Hansen, 2005] repeatedly solve a bounded-cost search problem [Stern et al., 2011], using the cost of the current best plan as the bound. (In this experiment, the cost of the best plan found by LAMA after 1 hour, or the plan found by IBaCoP2 in the 2014 IPC is considered as the initial bound of the bounded-cost search.) They show the same pattern of diminishing improvement with increasing time as other anytime search methods.

Plan *post-processing* is an alternative to anytime search, taking as input a valid plan and seeking to improve it. Figure 1.2 shows results for Plan Neighbourhood Graph Search [Nakhost and Müller, 2010], the current best plan post-processing approach. In the experiment, the set of input plans (referred to as "base plans") are the best plans found by LAMA after 1 hour, or the plan found by IBaCoP2 in the 2014 IPC. PNGS tries to improve a plan by finding shortcuts in a subgraph of the state space of the problem, constructed around the current plan. (The PNGS implementation used in this experiment also applies Nakhost's and Müller's "action elimination" technique.) Applying PNGS results in substantial plan quality improvements quickly – 94.8% of improved plans are found in less than 10 minutes – but then the improvement ceases because the planners exhaust memory.

All the above anytime search-based and post-processing-based methods continue to search for better plans until they exhaust the search space, or run out of time or memory. However, as Figure 1.2 shows, they become unproductive as runtime increases. The main reason is that they run out of memory. In this experiment, LAMA exhausted available memory (8 Gb) on 67% of problems before the 7 hour CPU time limit; the corresponding figure for PNGS is 93.7%. In other words, these

methods do not provide a continuing improvement of plan quality at every time scale, which opens up a challenge for further research.

1.2.2 Contributions to Plan Optimisation

We address the optimality gap by proposing a new approach to continuing plan improvement, and its implementation in a system called BDPO2, which is able to tackle large problems and works at varying time scales. BDPO2, like PNGS, is a plan post-processing system, meaning that it starts from an initial suboptimal plan provided by other methods. What Figure 1.2 shows is that switching to BDPO2 after some time can overcome the limitation of current anytime planning techniques, and continue to improve plan quality as allotted time increases. The set of input plans for BDPO2 starting at 1 hour (shown in Figure 1.2) are the same as for PNGS. The best result, as shown, is obtained by chaining several techniques together, applying first PNGS on the base plans, and then BDPO2 on the best result produced by PNGS. This result could not have been achieved by previous anytime planning or post-processing-based plan improvement approaches alone.

The BDPO2 approach uses Large Neighborhood Search (LNS), a local search technique. The local search explores a neighbourhood around the current solution plan for a better quality plan. In LNS, the neighbourhood of a solution defined by "destroy" and "repair" methods, which together replace a part of the current solution, while keeping the rest of it unchanged. In BDPO2, the destroy step selects a subsequence of some linearisation of a deordering of the current plan (we call this a "window") and the repair step applies a bounded-cost planner to the subproblem of finding a better replacement for this subplan. This focus on solving smaller subproblems makes local search, and LNS in particular, scale better to large problems. The size and structure of the neighborhood, however, plays a crucial role in the performance of local search [Hoffmann, 2001]. In our setting, the neighbourhood is determined by the strategies used to select windows and subplanners. Traditionally, LNS destroy methods often contain an element of randomness, and the local search is a simulated annealing which may accept moves to lower-quality solutions [Ropke and Pisinger, 2006; Schrimpf et al., 2000]. In contrast, we explore the neighbourhood systematically, examining candidate windows generated and ordered by several heuristics, and accept only moves to strictly better plans. We also introduce into LNS the idea of *delayed restarting*, meaning that we search for and combine multiple local improvements before restarting the next iteration from the best of all new plans. We have found that delayed restarts allow better exploration of subplans from different parts of the current plan, and help avoid local minima that otherwise occur when the system attempts to re-optimise the same part of the plan in successive iterations.

The BDPO2 framework shown in Figure 1.3 consists of two key components: plan decomposition and LNS-based plan optimisation. The first step, also the core component of BDPO2, is a decomposition of input plans into meaningful subplans using block deordering (discussed previously in Section 1.1.2, and later in Chapter 2). The

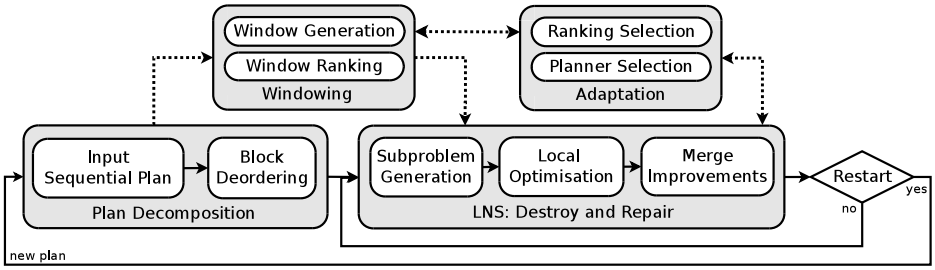


Figure 1.3: General framework of BDPO2.

plan decomposition allows us to generate a large collection of subplans of various types for local optimisations rather than optimising the plan as a whole. In the next step, BDPO2 uses Large Neighborhood Search (LNS) over the decomposed plan to find better quality neighboring plans, by performing local optimisation of subplans. The LNS process is further enhanced by two components: windowing and on-line adaptation. The windowing component uses (1) a portfolio of strategies for extracting candidate subplans from the decomposed plan, and (2) a set of ranking policies for ordering the subplans such that the system attempts to optimise more "promising", meaning more likely to be improved, subplans first. The on-line adaptation of the LNS process helps the process to make the right choice for the current problem (e.g., which subplanner to select for each local optimisation attempt). We take advantage of the fact that the system makes these choices many times over the course of the local search, to learn on-line which is the best policy for the current problem. In particular, we use the UCB1 multi-armed bandit learning policy [Auer et al., 2002] for subplanner selection, and a sequential portfolio of window ranking policies. Using different strategies within windowing and on-line adaptation improves the capability and robustness of the system, since no single strategy dominates all others across all problems.

1.3 Macro Generation

For many large problems in complex (or even in simple) domains, it remains challenging to find any solution within some resource constraint, despite the recent progress in planning technologies. Reducing problem complexity by automatic reengineering of the domain is one way to address this challenge. The structural knowledge of planning problems can be acquired by analysing training plans (e.g., solutions to simpler problems), and can be exploited to achieve such domain enhancement. Macro-operators (or *macros*, for short), formulated by encapsulating a sequence of planning operators, are a well known and a widely studied kind of structural knowledge that can be encoded in the same way as original planning operators in the domain model. Macros represent high level tasks comprising low level details. Combining several steps in the state space, macros provide extended visi-

bility of the search space to the planner. Therefore, macros can be used to reach the goals using relatively short plans, or to escape local heuristic minima. Thus, macros can improve the planners coverage, meaning the number of planning problems that can be solved given some resource constraints.

1.3.1 Challenges in Finding Useful Macros

A large number of existing macro learning methods are planner or domain specific; they are formulated by exploiting specific structural knowledge about planners or domains. MARVIN [Coles et al., 2007], for example, learns macros that help a forward chaining heuristic based planner escape plateaus in its heuristic profile. Such plateau-escaping properties are not likely to be common with most other planning paradigms. On the other hand, a few macro learning methods, Wizard [Newton et al., 2007] and MUM [Chrpa et al., 2014], for example, work with arbitrarily chosen planners and domains. Most macro learning methods, regardless of being planner (or domain) independent or not, formulate macros based on a subsequence of steps from totally ordered training plans. Such methods often fail to capture some promising macro candidates in a wide range of domains. This is because a subsequence of a totally ordered plan often contains unnecessary orderings, comprising seemingly random sequences of steps, which causes the subsequence to be unexplainable. Such activities appear frequently in plans, and thus cannot be useful as a macro. Chrpa [2010] shows a technique that can formulate macros with non-adjacent steps of a totally ordered training plan by investigating pair-wise step dependencies in the plan. However, the technique is unable to find many important activities, which could otherwise be captured by investigating dependencies between two groups of steps, rather than a pair of steps. Besides the difficulties in finding useful macros, often macros have a large number of instances, which can be memory demanding because of considerably increasing the size of grounded problem representation. Therefore, it still remains a challenge to find truly useful macros in a planner and domain independent way.

1.3.2 Contribution to Macro Generation

Further to the plan optimisation system, we present in this thesis a new way of computing macros, which results in improved planning efficiency and coverage. The macro generation process, like BDPO2, is based on block-structured plan decomposition. Plans provide useful knowledge for formulating macros. Plans invariably reflect the successful choices of actions to traverse the problem state space, and thus could bear the characteristics of the planner, the domain, or the problem inherently [Newton et al., 2007]. For example, a repeating subsequence of plan steps could indicate the presence of structural repetitions in the corresponding domain or problem. Therefore, the choices of a planner on solving a problem, or other domain structure can be captured from plans. Block decomposed plans that hierarchically organise coherent subplans can provide better structural knowledge for formulating macros

than totally ordered or standard partially ordered plans. In fact, a block itself is a promising candidate for a macro because of its nature of encapsulating effects and preconditions of constituent steps, and reducing interference with steps outside the block. We introduce *macro candidates* that extend the blocks by considering their structural relations, in order to capture more promising and complex (or bigger) activities frequently used in plans. We then extract macros from macro candidates, useful for the current domain. The macros that appear frequently in macro candidates are added into the domain model. Training plans are then re-generated using the macro enhanced domain model. Macros that do not appear, or that appear infrequently in the re-generated training plans are filtered out. Thus, we let the planner decide which macros are useful for it. As a final step, we use macro-specific entanglements [Chrpa and McCluskey, 2012], relations between planning operators and predicates, for efficient pruning of unnecessary instances of macros. We present a new macro learning process, implemented in a system called BLOMA, that can generate useful longer macros in problems whose structure relies on repetitive application of larger sets of plan steps. In the Barman domain, for example, BLOMA finds an eight-step long macro that captures an important activity: shaking a cocktail, pouring it into a shot, and cleaning the shaker afterwards. Traditional macro learning techniques that are based on "operator chaining" approaches (i.e. assembling operators one by one) are often not able to find such long macros. BLOMA is evaluated by using the IPC benchmarks with state-of-the-art planning engines, and shows considerable improvement in solving more problems (within a given time and memory bound) in a number of domains.

1.4 Thesis Outline

The body of this thesis is structured around the three key contributions outlined above, and according to the same order: (1) plan decomposition, (2) continuing plan quality optimisation, and (3) macro generation. The necessary background in the literature, empirical results with analysis, and related work to the above contributions are discussed separately in the relevant chapters.

We discuss the plan decomposition and macro generation separately in Chapters 2 and 6 respectively. Particularly Chapter 2 describes our novel block-structured plan deordering technique, showing how it is capable of decomposing a plan into encapsulated subplans, and removes the inherent limitations of existing plan deordering techniques. Chapter 6 describes our new method BLOMA that learns domain-specific macros from block decomposed plans, showing how it is able to capture longer macros representing high-level activities useful for improving planners' efficiency and coverage.

The LNS-based continuing plan quality optimisation, being the biggest and the most important contribution of this thesis, is further divided into three parts: (1) LNS applied to plan optimisation, (2) windowing, and (3) on-line adaptation. These are discussed in Chapters 3, 4, and 5 respectively. Chapter 3 provides an overview

of our plan quality optimisation system, BDPO2, followed by a discussion of the LNS part of BDPO2. This chapter also presents the experimental setups used to evaluate BDPO2 and the main empirical results. Chapter 4, as mentioned, presents a detailed discussion of the windowing strategies, showing how to extract candidate subplans from block decomposed plans, and rank those based on how likely they are to be improved. In Chapter 5, we describe how we can learn, over the course of local optimisation process, the relative success rate of different subplanners, window generation strategies, and ranking policies on the current problem, in order to adapt the system to the current problem.

Finally Chapter 7 concludes the thesis, describing how the contributions have identified, quantified, and addressed the challenges of plan decomposition, continuing plan quality optimisation, and macro generation. Also, it outlines ideas for future research.

Chapter 2

Plan Decomposition

This chapter provides necessary background on classical planning and plan deordering, followed by a detailed discussion of our contributions: (1) Block decomposed partially ordered plans (a new form of partially ordered plans), and (2) block deordering, the process of transforming a sequential or standard partially ordered plan into a block decomposed partially ordered plan. In describing block deordering, we show how decomposing a plan into meaningful non-interleaving subplans can relax the ordering constraints between these subplans, which allows deordering of a plan also in some cases where no deordering is possible by the conventional plan deordering techniques.

We structure this chapter as follows. Section 2.2 provides necessary background and existing techniques in the literature on plan decomposition. Sections 2.3 describes the formation and validity of our proposed block decomposed partially ordered plan, and how such plans differ from the standard partially ordered plans. Section 2.4 presents necessary conditions and their correctness under which adding blocks to a block decomposition admits the removal of basic ordering constraints. Section 2.5 describes our complete block deordering algorithm. The results of our experiments, run over a large collection of benchmark problems, are analyzed in Section 2.6 where we justify the importance of this plan decomposition.

This chapter describes work published as "Block-Structured Plan Deordering" [Siddiqui and Haslum, 2012]. However, the theory and practice of block deordering presented here is slightly different from the published account. To be specific, we introduce here the notion of threat protection ordering (described in Section 2.4) that contrasts the semantics of block decomposed partially ordered plans with the traditional partially ordered plans in a clearer way. Explicitly checking this ordering during the process of block deordering allows us to do further deordering in some cases. This thesis also explains in more details the correctness of the deordering rules.

2.1 Introduction

Plan decomposition, in general, divides a plan into subplans. A meaningful decomposition of a plan can facilitate post-plan generation tasks, such as scheduling

the plan for improved efficiency or robustness, or breaking it into subplans for distributed execution, etc. We are interested in decomposing a plan into subplans that are more coherent and exhibit little interference with other parts of the plan. We find such a decomposition useful for relaxing the ordering constraints between the subplans, which in turn helps to formulate subplans suitable for local optimisation and macro generation – the two other topics of this thesis. However, coherent subplans are not easy to extract from a, typically sequential, input plan, because of having unnecessary ordering constraints among the plan steps. Therefore, a key step is removing unnecessary ordering constraints from the input plan. This process is called *plan deordering*. The importance of deordering is clearly seen in our experiment (described in Section 3.9) on plan quality optimisations, where the input plans were already of high quality. In that experiment, the total quality improvements (measured by the average IPC quality score) found by our plan improvement system BDPO2 without performing any deordering is 28.7% less than that found by BDPO2 using our plan deordering technique.

The standard notion of a valid partially ordered plan requires all unordered steps in the plan to be non-interfering (i.e., for two subsequences of the plan to be unordered, every interleaving of steps from the two must form a valid execution). This limits the amount of deordering that can be done, in some cases to the extent that no deordering of a sequential plan is possible. (An example of this situation is shown in Figure 2.3 on page 22.) To remedy this, we have introduced *block deordering* [Siddiqui and Haslum, 2012], which creates a hierarchical decomposition of the plan into non-interleaving blocks and deorders these blocks. This makes it possible to deorder plans further, including in some cases where conventional, "step-wise", deordering is not possible. (Again, an example can be found in Figure 2.3 on page 22.)

2.2 Background on Plan Decomposition

This section provides necessary background on the plan decomposition technique developed and described later in this chapter. To be specific, this section presents a standard representation of classical planning problems, formation and validity of sequential and partially ordered plans, and the existing deordering techniques that transform a sequential plan into a partially ordered plan as a basic approach to plan decomposition.

2.2.1 The Planning Problem, Sequential Plan and its Validity

We consider the standard set-theoretic representation of classical planning problems, which is defined by a tuple $\Pi = \langle \mathcal{M}, \mathcal{A}, \mathcal{C}, I, G \rangle$, where $\mathcal{M}$ is a set of atoms (alternatively called fluents or propositions), $\mathcal{A}$ is a set of actions, $\mathcal{C} : \mathcal{A} \to \mathbb{R}^{0+}$ is a cost function on actions, which assigns to each action a non-negative cost, $I \subseteq \mathcal{M}$ is the initial state, and $G \subseteq \mathcal{M}$ is the goal.

An action a is characterised by a triple $\langle \text{pre}(a), \text{add}(a), \text{del}(a) \rangle$, where $\text{pre}(a)$, $\text{add}(a)$, and $\text{del}(a)$ are the preconditions, add and delete effects of a respectively. We

also say that action a is a *consumer* of an atom m if $m \in \text{pre}(a)$, a *producer* of m if $m \in \text{add}(a)$, and a *deleter* of m if $m \in \text{del}(a)$. An action a is applicable in a state S if $\text{pre}(a) \subseteq S$, and if applied in S, results in the state $\text{apply}(a, S) = (S \setminus \text{del}(a)) \cup \text{add}(a)$. A sequence of actions $\pi = \langle a_i, a_{i+1}, ..., a_j \rangle$ is applicable in a state S_i if (1) $\text{pre}(a_k) \subseteq S_k$ for all $i \leq k \leq j$, and (2) $S_{i+1} = \text{apply}(a_i, S_i)$, $S_{i+2} = \text{apply}(a_{i+1}, S_{i+1})$, and so on; the resulting state is $\text{apply}(\pi, S_i) = S_{i+j+1}$.

A valid *sequential plan* (also *totally ordered plan*) $\pi_{\text{seq}} = \langle a_1, ..., a_n \rangle$ for a planning problem Π is a sequence of actions that is applicable in I and such that $G \subseteq \text{apply}(\pi_{\text{seq}}, I)$. The actions of π_{seq} must be executed in the specified order.

2.2.2 The Partially Ordered Plan and its Validity

Plans can be partially ordered, in which case actions can be unordered with respect to each other. A *partially ordered plan* (*p.o. plan*) is a tuple, $\pi_{\text{pop}} = \langle \mathcal{S}, \prec \rangle$, where $\mathcal{S}$ is a set of steps (each of which is labelled by an action from $\mathcal{A}$) and $\prec$ represents a strict (i.e., irreflexive) partial order over $\mathcal{S}$. The unordered steps in π_{pop} can be executed in any order. $\prec^+$ denotes the transitive closure of $\prec$. An element $\langle s_i, s_j \rangle \in \prec$ (also $s_i \prec s_j$) is a *basic ordering constraint* iff it is not transitively implied by other constraints in $\prec$. The basic orderings are unique, because the ordering relation is acyclic, and the transitive reduction of an acyclic graph (or relation) is unique. For a plan step s, we use $\text{pre}(s)$, $\text{add}(s)$ and $\text{del}(s)$ to denote the preconditions, add and delete effects of the action associated with s. We also use the terms producer, consumer, deleter, and cost for plan steps, referring to their associated actions. We include in $\mathcal{S}$ two more steps, s_I and s_G. s_I is ordered before all other steps, consumes nothing and produces the initially true atoms, while s_G is ordered after all other steps, consumes the goal atoms and produces nothing.

A *linearisation* of π_{pop} is a total ordering of the steps in $\mathcal{S}$ that respects $\prec$. A p.o. plan π_{pop} is *valid* (for a planning problem Π) iff every linearisation of π_{pop} is a valid sequential plan (for Π). In other words, a p.o. plan can be viewed as a compact representation of a set of totally ordered plans, namely its linearisations.

Every basic ordering constraint, $s_i \prec s_j$, in π_{pop} has a set of associated reasons, denoted by $\text{Re}(s_i \prec s_j)$. These reasons explain why the ordering is necessary for the plan to be valid: If $\text{Re}(s_i \prec s_j)$ is non-empty, then some step precondition may be unsatisfied before its execution in some linearisations of π_{pop} that violate $s_i \prec s_j$. The reasons are of three types:

PC(m) (producer–consumer of atom m): The first step, s_i, produces m which is a precondition of the second step, s_j. Thus, if the order is changed, and s_j executed before s_i, the precondition of s_j may not have been established when it is required.

CD(m) (consumer–deleter of m): The second step, s_j deletes m, which is a precondition of s_i. Thus, if the order is changed, m may be deleted before it is required.

DP(m) (deleter–producer of m): The first step, s_i deletes m, which is produced by the second step, s_j. If the order is changed, the add effect of the producer step

may be undone by the deleter, causing a later step to fail. It is, however, not necessary to order a producer and deleter if no step that may occur after the producer in the plan depends on the added atom.

Note that an ordering constraint can have several associated reasons, including several reasons of the same type but referring to different atoms. The producer–consumer relation $\mathrm{PC}(m) \in \mathrm{Re}(s_i \prec s_j)$ is usually called a *causal link* from s_i to s_j for m [Dean and McKeown, 1991], and denoted by a triple $\langle s_i, m, s_j \rangle$. A causal link $\langle s_i, m, s_j \rangle$ is threatened if there is any deleter of m that may be ordered between the last producer of m before s_j and s_j, since this implies a possibility of m being false when required for the execution of s_j. The formal definition is as follows.

Definition 1. *Let $\pi_{\mathrm{pop}} = \langle \mathcal{S}, \prec \rangle$ be a p.o. plan, and $\langle s_p, m, s_c \rangle$ be a causal link in π_{pop}. $\langle s_p, m, s_c \rangle$ is* threatened *if there is a step s_d that deletes m such that neither (1) $s_c \prec^+ s_d$ nor (2) $\exists s'_p : m \in \mathrm{add}(s'_p) \wedge s_d \prec^+ s'_p \prec^+ s_c$ is true.*

As mentioned above, a p.o. plan, $\pi_{\mathrm{pop}} = \langle \mathcal{S}, \prec \rangle$ for a planning problem Π is valid iff every linearisation of π_{pop} is a valid sequential plan for Π. However, the following theorem gives an alternative, equivalent, condition for p.o. plan validity.

Theorem 1 (e.g., [Nebel and Bäckström, 1994])**.** *A p.o. plan is valid iff every step precondition can be supported by a causal link such that there is no threat to that causal link.*

This condition is the same as Chapman [1987]'s modal truth criterion, that

$$\begin{aligned}
&\forall s_c \in \mathcal{S},\ \forall m \in \mathrm{pre}(s_c) : \\
&\quad \exists s_p \in \mathcal{S} : (\mathrm{PC}(m) \in \mathrm{Re}(s_p \prec s_c) \wedge \\
&\qquad \forall s_t : \Big(m \in \mathrm{del}(s_t) \Rightarrow \Big(s_c \prec^+ s_d \vee \exists s'_p : \Big(m \in \mathrm{add}(s'_p) \wedge s_d \prec^+ s'_p \prec^+ s_c \Big) \Big) \Big) .
\end{aligned}$$

2.2.3 Deordering

The process of *deordering* converts a sequential plan into a p.o. plan by removing ordering constraints between steps, such that the steps of the plan can be successfully executed in any order consistent with the partial order and still achieve the goal [Bäckström, 1998]. We will refer to this as *step-wise* deordering, to distinguish it from the block decomposition and deordering that we introduce later in this chapter. Since current state space search planners can produce sequential plans very efficiently, deordering plays an important role in efficient generation of p.o. plans.

Let $\pi_{\mathrm{pop}} = \langle \mathcal{S}, \prec \rangle$ be a valid p.o. plan. A (step-wise) deordering of π_{pop} is a valid plan $\pi'_{\mathrm{pop}} = \langle \mathcal{S}, \prec' \rangle$ such that $(\prec')^+ \subset \prec^+$. That is, π'_{pop} is the result of removing some basic ordering constraints without invalidating the plan. A sequential plan $\pi_{\mathrm{seq}} = \langle a_1, ..., a_n \rangle$ can be represented as a p.o. plan by first introducing a set S of steps where each step $s_i \in S$ is labelled by a separate action a_i in π_{seq}, and then by introducing a set of ordering relations of the form $s_i \prec s_j$ whenever $i < j$. Hence there is no two steps in S which are unordered to each other. Thus, deordering a sequential plan is no different from (further) deordering a p.o. plan.

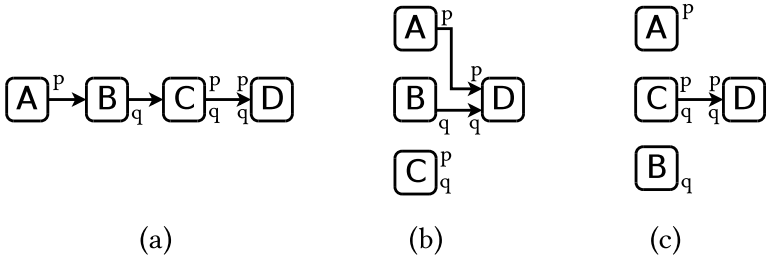


Figure 2.1: A failure case (derived from Figure 14 in Bäckström [1998]'s discussion of plan deordering) for making a plan least-constrained by the KK algorithm . (a) is an input sequential plan, (b) is the plan produced by KK after choosing the earliest possible producer (for the validation structure) of the preconditions p and q of D, and (c) is the minimal deordered version of (a). For simplicity, the goal atoms produced by the steps: A, B, C, or D are not shown in the figure.

Computing a (step-wise) deordering with a minimum number of ordering constraints is NP-hard [Bäckström, 1998], but there are several non-optimal algorithms [Pednault, 1986; Veloso et al., 1990; Régnier and Fade, 1991]. We have used the explanation-based generalisation algorithm by Kambhampati and Kedar [1994] (here referred to as KK) for step-wise plan deordering. The algorithm works in two phases: In the first phase it constructs a validation structure, which has exactly one causal link $\langle s_p, m, s_c \rangle$ for each precondition m of each step s_c. s_p is chosen as the earliest (instead of the latest as used by Veloso et al. [1990]) producer of m preceding s_c in the input plan, with no intervening threatening step (i.e., that deletes m) between s_p and s_c. In the second phase, the algorithm builds a partial ordering, keeping only those orderings in the original plan which either correspond to causal links in the validation structure or that are required to prevent a threatening step from becoming unordered w.r.t. the steps in such a causal link.

KK's deordering algorithm, due to its non-admissible greedy strategy, does not guarantee optimality. One example for KK's algorithm is shown in Figure 2.1, where it fails to transform a totally ordered plan to a least-constrained plan. However, a recent study found that KK's algorithm did produce optimal step-wise plan deorderings of all plans on which it was tested [Muise et al., 2012].

In this thesis, our purpose of plan deordering is to find a deordering that is adequate for generating useful candidate subplans for local optimisation and macro generation, and not to find an optimal step-wise deordering. More important than achieving the optimal deordering is overcoming the inherent limitation of step-wise deordering, which only allows plan steps to be unordered when they are non-interfering. Block deordering, described in the next two sections, can remove further orderings from input plans by forming blocks, which helps generate a decomposed plan that is more suitable for extracting subplans for local optimisation and macro generation.

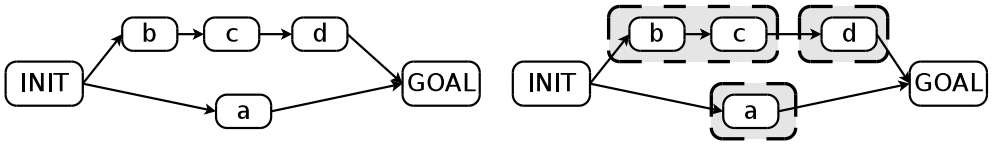


Figure 2.2: A typical p.o. plan (left) represents the set of all sequential plans that are linearisations of the plan steps, in this example $\langle a, b, c, d\rangle$, $\langle b, a, c, d\rangle$, $\langle b, c, a, d\rangle$, and $\langle b, c, d, a\rangle$. A block decomposed p.o. plan (as shown on the right with dashed outlines for blocks) allows unordered blocks to be executed in any order, but not steps from different blocks to be interleaved. Thus, only $\langle a, b, c, d\rangle$, $\langle b, c, a, d\rangle$, and $\langle b, c, d, a\rangle$ are possible linearisations of this plan.

2.3 Block Decomposition

In a conventional p.o. plan, whenever two subplans are unordered every interleaving of steps from the two forms a valid execution. This limits deordering to cases where individual steps are non-interfering. To remove this restriction, we have proposed *block decomposed partial ordering*, which restricts the interleaving of steps by dividing the plan steps into *blocks*, such that the steps in a block must not be interleaved with steps not in the block. However, steps within a block can still be partially ordered. This is illustrated with an example in Figure 2.2. The figure shows the difference in linearisations between a p.o. plan and a block decomposed p.o. plan. b, a, c, d is a valid linearisation in the standard partial ordering but not in the block decomposed p.o. plan. The formal definition of a block is as follows.

Definition 2. *Let $\pi_{\text{pop}} = \langle \mathcal{S}, \prec \rangle$ be a p.o. plan. A* block w.r.t. $\prec$*, is a subset $b \subset \mathcal{S}$ of steps such that for any two steps $s, s' \in b$, there exists no step $s'' \in (\mathcal{S} \setminus b)$ such that $s \prec^+ s'' \prec^+ s'$.*

A *decomposition* of a plan into blocks can be recursive, i.e., a block can be wholly contained in another. However, blocks cannot be partially overlapping. Two blocks are ordered $b_i \prec b_j$ if there exist steps $s_i \in b_i$ and $s_j \in b_j$ such that $s_i \prec s_j$ and neither block is contained in the other (i.e., $b_i \not\subset b_j$ and $b_j \not\subset b_i$).

Definition 3. *Let $\pi_{\text{pop}} = \langle \mathcal{S}, \prec \rangle$ be a p.o. plan. A set $\mathcal{B}$ of subsets of $\mathcal{S}$ is a* block decomposition *of π_{pop} iff (1) each $b \in \mathcal{B}$ is a block w.r.t. $\prec$ and (2) for every $b_i, b_j \in \mathcal{B}$, either $b_i \subset b_j$, $b_j \subset b_i$, or b_i and b_j are disjoint. A block decomposed plan is denoted by $\pi_{\text{bdp}} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$.*

The semantics of a block decomposed plan is defined by restricting its linearisations (for which it must be valid) to those that respect the block decomposition, i.e., that do not interleave steps from disjoint blocks. If $b_i \prec b_j$, all steps in b_i must precede all steps in b_j in any linearisation of the block decomposed plan.

Definition 4. *Let $\pi_{\text{bdp}} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a block decomposed p.o. plan for planning problem Π. A* linearisation *of π_{bdp} is a total order $\prec_{\text{lin}}$ on $\mathcal{S}$ such that (1) $\prec \subseteq \prec_{\text{lin}}$ and (2) every $b \in \mathcal{B}$ is a block w.r.t. $\prec_{\text{lin}}$. π_{bdp} is valid iff every linearisation of π_{bdp} is a plan for Π.*

Blocks behave much like (non-sequential) macro steps, having preconditions, add and delete effects that can be a subset of the union of those of their constituent steps. This enables blocks to encapsulate some plan effects and preconditions, reducing interference and thus allowing more deordering. The following definition captures those preconditions and effects that are visible from outside the block, i.e., those that give rise to dependencies or interference with other parts of the plan. These are what we need to consider when deciding if two blocks can be unordered. (Note that a responsible step is a step in a block that causes it to produce, consume or threaten an atom.)

Definition 5. *Let $\pi_{\text{bdp}} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a block decomposed p.o. plan, and $b \in \mathcal{B}$ be a block. The block semantics are defined as:*

- *b adds m iff b does not have precondition m, and there is a responsible step $\hat{s} \in b$ with $m \in add(\hat{s})$, such that for all $s' \in b$, if s' deletes m then $s' \prec \hat{s}$.*
- *b has precondition m iff there is a responsible step $\hat{s} \in b$ with $m \in pre(\hat{s})$, and there is no step $s' \in b$ such that there is a causal link $\langle s', m, \hat{s} \rangle$ without an active threat.*
- *b deletes m iff there is a responsible step $\hat{s} \in b$ with $m \in del(\hat{s})$, and there is no step $s' \in b$ with $\hat{s} \prec s'$ that adds m.*

Note that if a block consumes a proposition, it cannot also produce the same proposition. The reason for this is that taking the "black box" view of block execution, the proposition simply persists: it is true before execution of the block begins and remains true after it has finished. If the steps within a block are totally ordered, the preconditions and effects of a block according to Definition 5 are nearly the same as the "cumulative preconditions and effects" of an action sequence, defined by Haslum and Jonsson [2000], the only difference being that a consumer block cannot also be a producer of the same proposition.

A conventional p.o. plan, to be valid, must not contain any threat to a causal link. In contrast, a block decomposed p.o. plan allows a threat to a causal link to exist in the plan, as long as the causal link is protected from that threat by the block structure. A causal link is *protected* from a threat iff either (i) the causal link is contained by a block that does not contain the threat, or (ii) the threat is contained by a block that does not contain the causal link and does not delete the threatened atom (i.e., encapsulates the delete effect). A threat to a causal link is *active* if the link is not protected from it, otherwise *inactive*. The formal definition is as follows.

Definition 6. *Let $\pi_{\text{bdp}} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a block decomposed p.o. plan, and $s_t \in \mathcal{S}$ be a threat to a causal link $\langle s_p, m, s_c \rangle$ in π_{bdp}. $\langle s_p, m, s_c \rangle$ is* protected *from $s_t \in \mathcal{S}$ iff there exist a block $b \in \mathcal{B}$ such that either of the following is true: (1) $s_p, s_c \in b$; $s_t \notin b$; or (2) $s_t \in b$, $s_p, s_c \notin b$, and $m \notin del(b)$.*

The following theorem provides an alternative criterion for the validity of a block decomposed p.o. plan, in analogy with the condition for a conventional p.o. plan given in Theorem 1. The only difference is that a block decomposed p.o. plan allows

threats to causal links, as long as those threats are inactive. Let $\pi_{bdp} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a block decomposed p.o. plan. Analogously with Chapman's modal truth criterion, this condition can be stated as follows:

$$\begin{aligned}&\forall s_c \in \mathcal{S},\ \forall m \in \text{pre}(s_c)\\ &\quad \exists s_p \in \mathcal{S} : (m \in \text{add}(s_p) \wedge\\ &\qquad \forall s_t \in \mathcal{S} : (m \in \text{del}(s_t) \wedge s_t \not\prec^+ s_p \wedge s_c \not\prec^+ s_t \Rightarrow \langle s_p, m, s_c \rangle \text{ is protected from } s_t)).\end{aligned}$$

Theorem 2. *A block decomposed p.o. plan is valid iff every step precondition is supported by a causal link that has no active threat.*

Proof. Let $\pi_{\text{bdp}} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a block decomposed p.o. plan of a planning problem Π. Let us first prove the 'if' part, i.e., that if every step precondition is supported by a causal link that has no active threat then every linearisation of π_{bdp} is a valid plan for Π. Let $\pi_{\text{seq}} = \langle \ldots, s_c, \ldots \rangle$ be an arbitrary linearisation of π_{bdp} with a total order $\prec_{\text{seq}}$ on $\mathcal{S}$, where $m \in \text{pre}(s_c)$. Then, according to the validity criteria for a sequential plan, we have to show that m must be satisfied before the execution of s_c in π_{seq}. Since every step precondition is supported by a causal link in π_{bdp} that has no active threat, m must be supported by a causal link $\langle s_p, m, s_c \rangle$ that has no active threat. Moreover, since $\prec \subseteq \prec_{\text{seq}}$ then $s_p \prec_{\text{seq}} s_c$. Let s_t be a threat to $\langle s_p, m, s_c \rangle$ in π_{bdp}. Clearly, $s_p \prec_{\text{seq}} s_t \prec_{\text{seq}} s_c$ is the only possibility that may cause m to be unsatisfied before the execution of s_c. Since $\langle s_p, m, s_c \rangle$ has no active threat, $\langle s_p, m, s_c \rangle$ is protected from s_t, and therefore, according to Definition 6, either (1) $s_p, s_c \in b$ and $s_t \notin b$, or (2) $s_t \in b$, $s_p, s_c \notin b$, and $m \notin \text{del}(b)$, must hold. If (1) is true, then $s_p \prec_{\text{seq}} s_t \prec_{\text{seq}} s_c$ cannot occur in any valid linearisation of π_{bdp}, since it interleaves steps $s_p, s_c \in b$ with $s_t \notin b$, and thus b is not a block w.r.t. $\prec_{\text{seq}}$. In the second case, since $m \notin \text{del}(b)$ then there must be a producer of m, $s'_p \in b$, such that $s_t \prec_{\text{seq}} s'_p$. Moreover, since $s_p, s_c \notin b$, $s_p \prec_{\text{seq}} s_t \prec_{\text{seq}} s_c$ can only be true if $s_p \prec_{\text{seq}} s_t \prec_{\text{seq}} s'_p \prec_{\text{seq}} s_c$. This also makes m true before the execution of s_c in π_{seq}.

Let us now prove the 'only if' part, i.e., that if π_{bdp} is valid then every step precondition is supported by a causal link that has no active threat. Let $s_c \in \mathcal{S}$, $m \in \text{pre}(s_c)$, and $\pi_{\text{seq}} = \langle \ldots, s_c, \ldots \rangle$ be a linearisation of π_{bdp} with a total order $\prec_{\text{seq}}$ on $\mathcal{S}$. We consider two possible situations: (1) there is no producer s' from which a causal link $\langle s', m, s_c \rangle$ in π_{bdp} can be constructed, or (2) there is at least one such producer that can construct the causal link with s_c for the atom m but that causal link has an active threat in π_{bdp}. We will show that none of the above situations can happen as long as π_{bdp} is valid. According to situation (1), there is no s' in π_{seq} as well such that $s' \prec_{\text{seq}} s_c$. This causes m to be unsatisfied before the execution of s_c in π_{seq}, i.e., π_{seq} become invalid. Consequently, π_{bdp} become invalid (since one of its linearisation is invalid), which contradicts with our assumption. Therefore, there must exist at least one producer s' that can construct a causal link $\langle s', m, s_c \rangle$ in π_{bdp}. Now, for situation (2), assume s_p is the last producer of m before the execution of s_c in π_{seq}, i.e., $\forall s'_p \in \mathcal{S} \setminus s_p : m \in \text{add}(s'_p) \Rightarrow (s'_p \prec_{\text{seq}} s_p \vee s_c \prec_{\text{seq}} s'_p)$. Let s_p be the producer in the causal link $\langle s_p, m, s_c \rangle$ in π_{bdp} (which is possible, since s_p is not

ordered after s_c in π_{bdp}). Assume $\langle s_p, m, s_c \rangle$ has an active threat s_t in π_{bdp}. Since $\langle s_p, m, s_c \rangle$ has an active threat s_t (i.e., $\langle s_p, m, s_c \rangle$ is not protected from s_t), then neither (i) $s_p, s_c \in b$; $s_t \notin b$, nor (ii) $s_t \in b$; $s_p, s_c \notin b$, and $m \notin del(b)$, is true. Therefore, $s_p \prec_{\text{seq}} s_t \prec_{\text{seq}} s_c$ is a possible linearisation of π_{bdp}. Moreover, since there is no more producer of m in between s_p and s_c, m must be unsatisfied before the execution of s_c, i.e., π_{seq} becomes invalid. Consequently, π_{bdp} is invalid since one of its linearisations is invalid. Therefore, $\langle s_p, m, s_c \rangle$ must not have any active threat. □

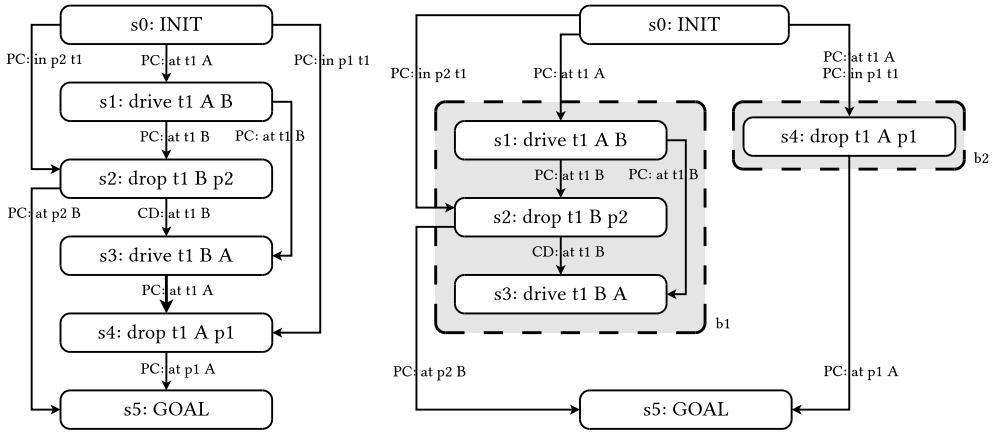
2.4 Block Deordering

Block deordering [Siddiqui and Haslum, 2012] is the process of removing orderings between plan steps by adding blocks to a block decomposed p.o. plan. It may also add to the plan some new basic ordering constraints, but those are transitively implied by the other ordering constraints. Block deordering can often remove ordering constraints where step-wise deordering cannot. This is because the no-interleaving restriction among the blocks affords us a simplified, "black box", view of blocks that localises some interactions, in which only the preconditions and effects of executing the block as a whole are important. Thus, it allows further deordering by being able to ignore some dependencies and effects that matter only internally within the block. In addition to providing more linearisations, by improving deordering, the blocks formed by block deordering often correspond to meaningful subplans that form the basis for (1) the windowing strategies (described in Chapter 4) used to generate candidate subplans for local optimisation, and (2) macro generation (described in Chapter 6) used to improve planners efficiency.

This section presents the conditions under which adding blocks to a block decomposition allows the removal of basic ordering constraints. The full block deordering algorithm is presented in the next section.

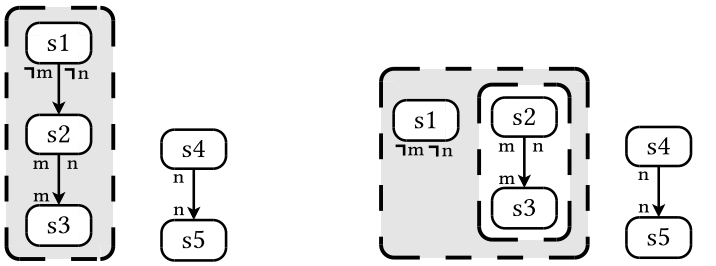
As a simple example of block deordering, Figure 2.3(i) shows a sequential plan for a small Logistics problem. This plan can not be deordered into a conventional p.o. plan, because each plan step has a reason to be ordered after the previous step. Block deordering, however, is able to break the ordering s3 $\prec$ s4 by removing the only reason PC(at P1 A) based on the formation of two blocks b1 and b2 as shown in Figure 2.3(ii). Neither of the two blocks delete or add the atom "at P1 A" (although it is a precondition of both). This removes the interference between them, and allows the two blocks to be executed in any order but without any interleaving. Therefore, the possible linearisations of the block decomposed p.o. plan are only (s1, s2, s3, s4) and (s4, s1, s2, s3). Note that if b2 is ordered before b1, then b1 can be optimised by removing step s3.

Besides the necessary orderings between a pair of steps in a plan due to reasons PC, CD, and DP (stated in Section 2.2.2), a valid block decomposed p.o. plan must maintain one more type of necessary ordering, called *threat protection ordering*. If removing an ordering $s_x \prec^+ s_y$ causes a block containing both steps to have delete effect that it did not have with this ordering, and that delete effect causes a causal



(i) sequential plan (ii) block deordered plan

Figure 2.3: A sequential plan and a block deordering of this plan with two unordered blocks b1 and b2. Ordering constraints are labelled with their reasons: producer–consumer (PC), i.e., causal link, deleter–producer (DP), and consumer–deleter (CD). Note that no ordering constraint in the sequential plan can be removed without invalidating it. Thus, step-wise deordering of this plan is not possible.



(i) valid block decomposition (ii) invalid block decomposition

Figure 2.4: Two block decompositions of a plan containing five steps: s1, s2, s3, s4, and s5. In decomposition (i), there are three (transitively reduced) necessary orderings: s1 $\prec$ s2, s2 $\prec$ s3, and s4 $\prec$ s5, where Re(s1 $\prec$ s2) = {DP(m), DP(n)}, Re(s2 $\prec$ s3) = {PC(m)}, and Re(s4 $\prec$ s5) = {PC(n)}. This decomposition is valid since every step precondition is satisfied by a causal link without active threats. The threat from s1 to causal link $\langle$s4, n, s5$\rangle$ is inactive, since the link is protected by block b_x = {s1, s2, s3} which contains s1 but does not delete m, and is disjoint from the causal link. By forming two blocks, b_y = {s1} and b_z = {s2, s3} it would be possible to remove s1 $\prec$ s2, as shown in (ii), since $\langle$s2, m, s3$\rangle$ is then protected from s1 by b_z. However, in this decomposition the delete effect of block b_x becomes $del(b_x) = \{\text{m}, \text{n}\}$, and the block therefore no longer protects $\langle$s4, n, s5$\rangle$. Therefore, this decomposition and deordering is invalid. The ordering s1 $\prec$ s2 is a threat protection ordering, which must not be broken. Note that s2 has no consumers of its produced atom n, yet acts as a white knight for $\langle$s4, n, s5$\rangle$ to protect n from the deleter s1.

link outside the block to become unprotected (not satisfying either of the two conditions of Definition 6), then $s_x \prec^+ s_y$ is a threat protection ordering, which may not be removed. A threat protection ordering can be introduced during the block deordering process, and once introduced cannot be removed. This is demonstrated in Figure 2.4, where removing this kind of ordering leads to an invalid block decomposed p.o. plan. The threat protection ordering is defined formally as follows.

Definition 7. *Let $\pi_{\text{bdp}} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a block decomposed p.o. plan, and $\langle s_p, m, s_c \rangle$ be a causal link that is protected from $s_t \in \mathcal{S}$ in π_{bdp}. Let $b \in \mathcal{B}$; $s_t, s' \in b$; $s_p, s_c \notin b$; $m \in add(s')$; $m \notin del(b)$; $s_t \prec^+ s'$. $s_t \prec^+ s'$ is a* threat protection ordering *if breaking this ordering causes $m \in del(b)$ and that causes $\langle s_p, m, s_c \rangle$ to become unprotected from s_t.*

The notion of threat protection ordering was missing from our earlier block deordering procedure [Siddiqui and Haslum, 2012], which relied (implicitly) on the stronger restriction that the delete effects of a block do not change due to subsequent deordering inside the block. Explicitly checking only the necessary threat protection orderings allows more deordering inside created blocks to take place.

To remove a basic ordering, $s_i \prec s_j$, from a block decomposed p.o. plan $\pi_{\text{bdp}} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$, we create two blocks, b_i and b_j, where $s_i \in b_i$, $s_j \in b_j$, and $b_i \cap b_j = \emptyset$. Note that one of the two blocks can consist of a single step. Both blocks must be consistent with the existing decomposition, i.e., $\mathcal{B} \cup \{b_i, b_j\}$ must still be a valid block decomposition, in the sense of Definition 2. In the remainder of this section, we define four rules which state conditions on blocks b_i and b_j that allow different reasons for the ordering $s_i \prec s_j$ to be eliminated. Since the ordering $s_i \prec s_j$ can exist for several reasons (including several reasons of the same type, referring to different atoms), it is only if blocks b_i and b_j can be found that allow us to remove every reason in $\text{Re}(s_i \prec s_j)$ that the ordering between the steps can be removed.

Rule 1. *Let $\pi_{\text{bdp}} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a valid block decomposed p.o. plan, $s_i \prec s_j$ be a basic ordering whose removal does not cause any threat protection ordering to be removed, and $\text{PC}(m) \in \text{Re}(s_i \prec s_j)$. Let b_i be a block, where $s_i \in b_i$, $s_j \notin b_i$, and $\forall s' \in b_i : s_i \not\prec s'$. $\text{PC}(m)$ can be removed from $\text{Re}(s_i \prec s_j)$ if $m \in \text{pre}(b_i)$ and $\exists s_p \notin b_i$ such that s_p can establish a causal link to b_i and to s_j.*

As an explanation of Rule 1, if $\text{PC}(m) \in \text{Re}(s_i \prec s_j)$, then b_i must not produce m. Since s_i produces m and is not followed by a deleter of m within b_i (because $s_i \prec s_j$ is a basic ordering and $s_j \notin b_i$) the only way for this to happen is if b_i consumes m. Since the plan is valid, there must be some producer, $s_p \notin b_i$, that necessarily precedes the step (in b_i) that consumes m. Note that $s_p \prec^+ s_j$. Then adding the causal link $\text{PC}(m)$ to $\text{Re}(s_p \prec s_j)$ (i.e., adding $\langle s_p, s_j \rangle$ to $\prec$ if not already present) allows $\text{PC}(m)$ to be removed from $\text{Re}(s_i \prec s_j)$.

Theorem 3. *Deordering according to* Rule 1 *preserves plan validity.*

Proof. Let $\pi_{\text{bdp}} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a valid block decomposed p.o. plan. Therefore, according to Theorem 2, every step precondition of π_{bdp} is supported by a causal link that has no active threat. Let $\text{p} \prec \text{q}$ be a basic ordering constraint (where $\text{p}, \text{q} \in \mathcal{S}$),

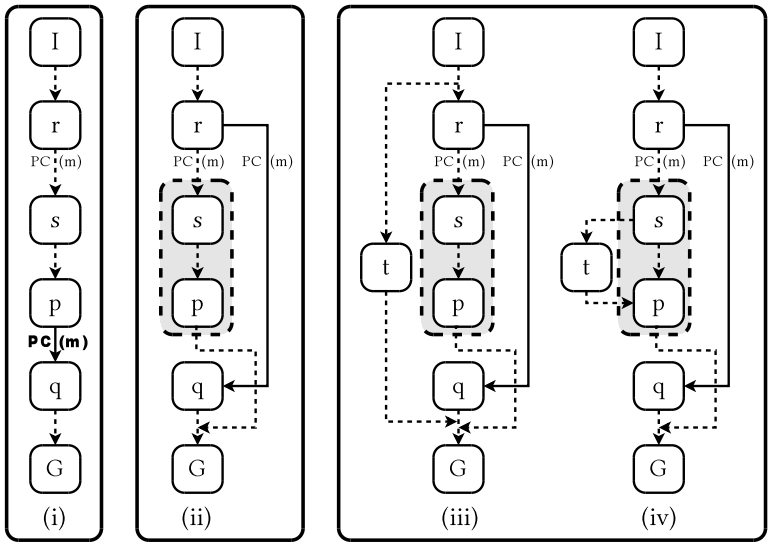


Figure 2.5: Formation of a block {s,p} and addition of a causal link $\langle$r,m,q$\rangle$ in (ii) in order to remove the reason PC(m) behind the basic ordering constraint p $\prec$ q from (i). Different situations, (iii and iv), where a threat, t, may be active to $\langle$r,m,q$\rangle$.

b_p, $b_q \in \mathcal{B}$ be the blocks that meet the conditions for removing PC(m) $\in$ Re(p $\prec$ q), and b_p, b_q are not ordered for any other ordering constraints. We will show that removing PC(m) from Re(p $\prec$ q) results in a new plan, $\pi'_{\text{bdp}} = \langle \mathcal{S}, \mathcal{B}', \prec' \rangle$, that meets the condition of Theorem 2, and therefore remains valid.

Assume PC(m) $\in$ Re(p $\prec$ q) is removed, and the precondition of q is now supplied by the step r based on the newly established causal link $\langle$r,m,q$\rangle$ after deordering and formulating b_p = {s,...,p}, b_q = {q} in π'_{bdp}, as shown in Figure 2.5 (ii). We have to show that $\langle$r,m,q$\rangle$ has no active threat in π'_{bdp}, and therefore, π'_{bdp} is valid. Assume, there is an active threat, t, to $\langle$r,m,q$\rangle$ in π'_{bdp}. Then, of course, t $\not\prec^+$ r and q $\not\prec^+$ t. We will examine every other situation, where t can be an active threat to $\langle$r,m,q$\rangle$.

Situation (1): assume s $\not\prec^+$ t, as shown in Figure 2.5 (iii). Since t is not an active threat to $\langle$r,m,s$\rangle$ in π_{bdp}, then according to Theorem 2, either t is contained by a block that does not delete the threatened atom and does not contain $\langle$r,m,s$\rangle$, or $\langle$r,m,s$\rangle$ is contained by a block $b' = \{r, s, ...\}$ that does not contain t. For the first case, it also holds true in π'_{bdp}, and therefore, t cannot be an active threat to $\langle$r,m,q$\rangle$. For the second case, b' cannot partially overlap with b_p = {s,...,p}, therefore, either $b_p \supseteq b'$ or $b' \supseteq b_p$. If $b_p \supseteq b'$, b_p must contain r, which cannot happen according to the PC removing criteria (i.e., r $\notin b_p$ must hold) stated in Rule 1. If $b' \supseteq b_p$, then b' must contain at least r, s, and p, because b' cannot partially overlap with b_p = {s,...,p}. Since t is also not an active threat to $\langle$p,m,q$\rangle$ in π_{bdp}, $\langle$p,m,q$\rangle$ must be contained by some block $b'' = \{p, q, ...\}$ that does not contain t. Now, since b' and b'' cannot partially overlap, b' or b'' (whichever is bigger) must contain at least r, s, p, and q, for

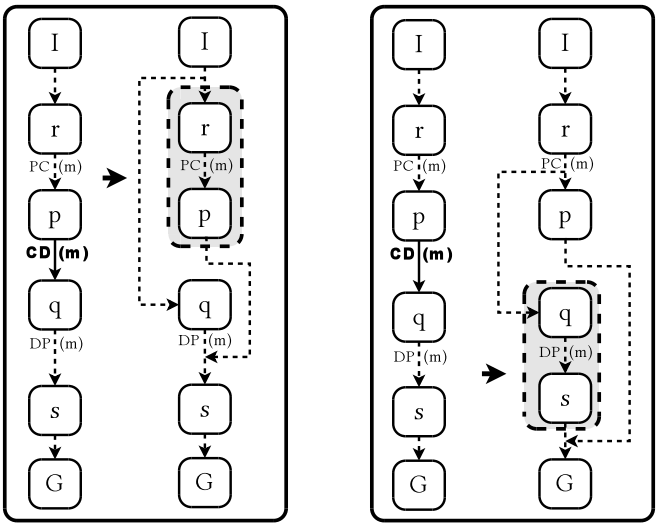


Figure 2.6: Formation of blocks for removing the reason CD(m) behind the basic ordering constraint p $\prec$ q.

which b' or b'' (whichever is bigger) protects ⟨r,m,q⟩ from t.

Situation (2): assume t $\not\prec^+$ p, also shown in Figure 2.5 (iii). Since t is also not an active threat to ⟨p,m,q⟩ in π_{bdp}, like before, we can show that either t is contained by a block that encapsulates the threatened atom (i.e., does not delete m) and does not contain ⟨p,m,q⟩, or ⟨p,m,q⟩ is contained by a block $b' = \{r, s, p, q, ...\}$ that does not contain t. In both cases, ⟨r,m,q⟩ is protected from t.

Situation (3): assume s $\prec^+$ t $\prec^+$ p shown in Figure 2.5 (iv). This is only possible if $t \in b_p$, since t cannot interleave with the steps in b_p if $t \notin b_p$. Therefore, $t \in b_p$, which causes ⟨r,m,q⟩ to be protected from t. This is because b_p does not contain ⟨r,m,q⟩ and does not delete m (since $m \in$ add(p) and t $\prec^+$ p).

Therefore, we can conclude that t can never be an active threat to ⟨r,m,q⟩ under any situation. □

Rule 2. *Let $\pi_{\text{bdp}} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a valid block decomposed p.o. plan, $s_i \prec s_j$ a basic ordering whose removal does not cause any threat protection ordering to be removed, and* CD(m) $\in$ Re($s_i \prec s_j$). *Let b_i and b_j be two blocks, where $s_i \in b_i$, $s_j \in b_j$, and $b_i \cap b_j = \emptyset$. Then* CD(m) *can be removed from* Re($s_i \prec s_j$) *if b_i does not consume m.*

Theorem 4. *Deordering according to* Rule 2 *preserves plan validity.*

Proof. Let $\pi_{\text{bdp}} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a valid block decomposed p.o. plan, and p $\prec$ q be a basic ordering constraint, where p, q $\in \mathcal{S}$ with CD(m) $\in$ Re(p $\prec$ q). In order to meet the condition of Rule 2, let us assume b_p is a block that includes r and p such that ⟨r,m,p⟩ is a causal link and every other consumer of m in b_p (if they exist) is ordered after r in π_{bdp} (as shown in Figure 2.6 (i)). Therefore it meets the condition that b_p

must not consume m. Also, assume b_q is a block that contains $\{\mathrm{q}\}$ and b_p, b_q are not ordered for any other ordering constraints. Therefore, $\mathrm{CD(m)} \in \mathrm{Re(p} \prec \mathrm{q)}$ as well as $\mathrm{p} \prec \mathrm{q}$ are removed, which results a new plan $\pi'_{\mathrm{bdp}} = \langle \mathcal{S}, \mathcal{B}', \prec' \rangle$. We will show that π'_{bdp} is valid according to Theorem 2.

Since π_{bdp} is valid, there is no active threat to any causal link in π_{bdp} according to Theorem 2, but due to the deordering of $\mathrm{p} \prec \mathrm{q}$, the deleter q becomes a new threat only to the causal link $\langle \mathrm{r,m,p} \rangle$ in π'_{bdp}. However, $\langle \mathrm{r,m,p} \rangle$ is contained by b_p that does not contain q, and therefore, according to Definition 6, $\langle \mathrm{r,m,p} \rangle$ is protected from q, i.e., q becomes an inactive threat. As a result, π'_{bdp} remains valid. □

Rule 3. *Let $\pi_{\mathrm{bdp}} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a valid block decomposed p.o. plan, $s_i \prec s_j$ a basic ordering whose removal does not cause any threat protection ordering to be removed, and* $\mathrm{CD}(m) \in \mathrm{Re}(s_i \prec s_j)$. *Let b_i and b_j be two blocks, where $s_i \in b_i$, $s_j \in b_j$, and $b_i \cap b_j = \emptyset$. Then* $\mathrm{CD}(m)$ *can be removed from* $\mathrm{Re}(s_i \prec s_j)$ *if b_j does not delete m.*

Theorem 5. *Deordering according to* Rule 3 *preserves plan validity.*

Proof. Let $\pi_{\mathrm{bdp}} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a valid block decomposed p.o. plan, and $\mathrm{p} \prec \mathrm{q}$ be a basic ordering constraint, where $\mathrm{p, q} \in \mathcal{S}$ with $\mathrm{CD}(m) \in \mathrm{Re(p} \prec \mathrm{q)}$. In order to meet the condition of Rule 3, let us assume b_q is a block that includes q and s such that $\mathrm{DP}(m) \in \mathrm{Re(q} \prec \mathrm{s)}$ and every other deleter of m in b_q (if they exist) is ordered before s in π_{bdp} (as shown in Figure 2.6 (ii)). Therefore it meets the condition that b_q must not delete m. Also, assume b_p is a block that contains $\{\mathrm{p}\}$, and b_p, b_q are not ordered for any other ordering constraints. Therefore, $\mathrm{CD(m)} \in \mathrm{Re(p} \prec \mathrm{q)}$ as well as $\mathrm{p} \prec \mathrm{q}$ is removed, which results a new plan $\pi'_{\mathrm{bdp}} = \langle \mathcal{S}, \mathcal{B}', \prec' \rangle$. We will show that π'_{bdp} is valid according to Theorem 2.

Since π_{bdp} is valid, there is no active threat to any causal link in π_{bdp} according to Theorem 2, but due to the deordering of $\mathrm{p} \prec \mathrm{q}$, the deleter q becomes a new threat only to the causal link $\langle \mathrm{r,m,p} \rangle$ in π'_{bdp}. However, q is contained by b_q that does not contain $\langle \mathrm{r,m,p} \rangle$, and does not delete m; therefore, according to Definition 6, $\langle \mathrm{r,m,p} \rangle$ is protected from q, i.e., q becomes an inactive threat. As a result, π'_{bdp} satisfies the condition of Theorem 2 and therefore remains valid. □

Rule 4. *Let $\pi_{\mathrm{bdp}} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a valid block decomposed p.o. plan, $s_i \prec s_j$ a basic ordering whose removal does not cause any threat protection ordering to be removed, and* $\mathrm{DP}(m) \in \mathrm{Re}(s_i \prec s_j)$. *Let b_j be a block, where $s_j \in b_j$ but $s_i \notin b_j$. Then* $\mathrm{DP}(m)$ *can be removed from* $\mathrm{Re}(s_i \prec s_j)$ *if b_j includes every step s' such that* $\mathrm{PC}(m) \in \mathrm{Re}(s_j \prec s')$.

Theorem 6. *Deordering according to* Rule 4 *preserves plan validity.*

Proof. Let $\pi_{\mathrm{bdp}} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a valid block decomposed p.o. plan, and $\mathrm{p} \prec \mathrm{q}$ be a basic ordering constraint, where $\mathrm{p, q} \in \mathcal{S}$ with $\mathrm{DP}(m) \in \mathrm{Re(p} \prec \mathrm{q)}$. Let b_q be a block that includes all the steps – r and s such that $\langle \mathrm{q,m,r} \rangle$, $\langle \mathrm{q,m,s} \rangle$ are the causal links in π_{bdp} (as shown in Figure 2.6 (ii)). Hence, it meets the condition of Rule 4. Also, assume b_p is a block that contains $\{\mathrm{p}\}$ and b_p, b_q are not ordered for any other

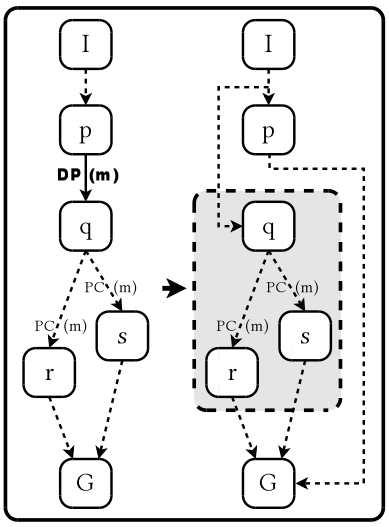


Figure 2.7: Formation of blocks for removing the reason DP(m) behind the basic ordering p $\prec$ q.

ordering constraints. As a result, DP(m) $\in$ Re(p $\prec$ q) as well as p $\prec$ q is removed, which results a new plan $\pi'_{\text{bdp}} = \langle \mathcal{S}, \mathcal{B}', \prec' \rangle$. We will show that π'_{bdp} satisfies the condition of Theorem 2 and therefore remains valid.

Since π_{bdp} is valid, there is no active threat to any causal link in π_{bdp} according to Theorem 2, but due to the deordering of p $\prec$ q, the deleter p becomes a new threat to the only causal links ⟨q,m,r⟩ and ⟨q,m,s⟩ in π'_{bdp}. However, those causal links are contained by b_q that does not contain p, and therefore, according to Definition 6, are protected from p, i.e., p becomes an inactive threat. As a result, π'_{bdp} remains valid. □

Even when, by applying the four rules above, we can find blocks b_i and b_j that remove all reasons for an ordering $s_i \prec s_j$, thus permitting the ordering to be removed, it is not guaranteed that the two blocks b_i and b_j will be unordered. They may be ordered because b_i contains some step other than s_i that is ordered before some step in b_j (whether s_j or another). Even if they are not, if there is a block $b \in \mathcal{B}$ that contains b_i (or b_j but not both), and b is still ordered before b_j (resp. after b_i) due to some constraint in $\prec^+$ other than $\langle s_i, s_j \rangle$, then blocks b_i and b_j will still be ordered, in the sense that b_i will appear before b_j in any linearisation consistent with the block decomposition.

2.5 Block Deordering Algorithm

The previous section described four conditions (Rules 1–4) under which adding blocks to a decomposition allows reasons for ordering constraints, and thus ulti-

mately the ordering constraints themselves, to be removed while preserving plan validity. Next, we describe the algorithm that uses these rules to perform block deordering, i.e., to convert a sequential plan π_{seq} into a block decomposed p.o. plan π_{bdp}.

The algorithm is divided into two phases. First, we apply a step-wise deordering procedure to convert π_{seq} into a p.o. plan $\pi_{\text{pop}} = (\mathcal{S}, \prec')$. We have used Kambhampati and Kedar [1994]'s algorithm for this, because it is simple and has been shown to produce very good results [Muise et al., 2012], even though it has no optimality guarantee.

After the step-wise plan deordering, we extend ordering to blocks. Each individual step in the step-wise plan deordered plan is initially treated as a block by itself. Two blocks b_i and b_j are ordered $b_i \prec b_j$ if there exist steps $s_i \in b_i$ and $s_j \in b_j$ such that $s_i \prec s_j$ and neither block is contained in the other (i.e., $b_i \not\subset b_j$ and $b_j \not\subset b_i$). In this case, all steps in b_i must precede all steps in b_j in any linearisation of the block decomposed plan. We also extend the reasons for ordering (PC, CD and DP) to ordering constraints between blocks, with the set of propositions produced, consumed and deleted by a block given by Definition 5. Recall that a *responsible step* is a step in a block that causes it to produce, consume or delete a proposition. For example, if b produces p, there must be a step $s \in b$ that produces p, such that no step in the block not ordered before s deletes p; we say step s is "responsible" for b producing p.

The next phase is block deordering, which converts the p.o. plan $\pi_{\text{pop}} = (\mathcal{S}, \prec)$ into a block decomposed p.o. plan $\pi_{\text{bdp}} = (\mathcal{S}, \mathcal{B}, \prec')$. This is done by a greedy procedure, which examines each basic ordering constraint $b_i \prec b_j$ in turn and attempts to create blocks that are consistent with the decomposition built so far and that will allow this ordering to be removed. The core of this algorithm is the RESOLVE procedure (Algorithm 1). It takes as input two blocks, b_i and b_j, that are ordered (one or both blocks may consist of a single step), and tries to break the ordering by extending them to larger blocks, b_i' and b_j'. The procedure examines each reason for the ordering constraint and extends one of the blocks to remove that reason, following the rules given in the previous section. After this, the sets of propositions produced, consumed and deleted by the new blocks (b_i' and b_j') are recomputed (following Definition 5) and any new reasons for the ordering constraint that have arisen because of steps that have been included are added to $\text{Re}(b_i' \prec b_j')$. This is repeated until either no reason for the ordering remains, in which case the new blocks returned by the procedure can safely be unordered, or some reason cannot be removed, in which case deordering is not possible (signalled by returning NULL). The function INTERMEDIATE(b_i, b_j) returns the set of steps ordered between b_i and b_j, i.e., $\{s \mid b_i \prec^+ s \prec^+ b_j\}$. Where Algorithm 1 refers to a "nearest" step s' preceding or following another step s, it means a step with a smallest number of basic ordering constraints between s' and s.

If we applied the RESOLVE procedure to each basic ordering constraint we would obtain a collection of blocks with which we can break some orderings. But this collection is not necessarily a valid decomposition, since some of the blocks may have partial overlap. To find a valid decomposition, we use a greedy procedure. We repeatedly examine each basic ordering constraint $b_i \prec b_j$ and call RESOLVE to find

Algorithm 1 Resolve ordering constraints between a pair of blocks.

procedure RESOLVE(b_i, b_j)
  Initialise $b'_i = b_i$, $b'_j = b_j$.
  while $\mathrm{Re}(b'_i \prec b'_j) \neq \emptyset$ **do**
    for each $r \in \mathrm{Re}(b'_i \prec b'_j)$ **do**
      if $r = \mathrm{PC}(p)$ **then**
        // try Rule 1
        Find a responsible step $s \in b'_i$ and a nearest $s' \notin b'_i$ that consumes p such that $s' \prec^+ s$.
        if such s' exists **then**
          Set $b'_i = b'_i \cup \{s'\} \cup \textsc{Intermediate}(s', b'_i)$.
        else return NULL
      else if $r = \mathrm{DP}(p)$ **then**
        // try Rule 4
        Find a responsible step $s \in b'_j$ and all $s' \notin b'_j$ such that each $\langle s, p, s' \rangle$ is a causal link.
        if such s' exists **then**
          Set $b'_j = b'_j \cup \{s'\} \cup \textsc{Intermediate}(b'_j, s')$.
        else return NULL
      else if $r = \mathrm{CD}(p)$ **then**
        // try Rule 3
        Find a responsible step $s \in b'_j$ and a nearest $s' \notin b'_j$ that produces p, such that $s \prec^+ s'$.
        if such s' exists **then**
          Set $b'_j = b'_j \cup \{s'\} \cup \textsc{Intermediate}(b'_j, s')$.
        else
          // try Rule 2
          Find a responsible step $s \in b'_i$ and a nearest $s' \notin b'_i$ that produces p, such that $s' \prec^+ s$.
          if such s' exists **then**
            Set $b'_i = b'_i \cup \{s'\} \cup \textsc{Intermediate}(s', b'_i)$.
          else return NULL.
    Recompute $\mathrm{Re}(b'_i \prec b'_j)$.
  return (b'_i, b'_j).

two extended blocks $b'_i \supseteq b_i$ and $b'_j \supseteq b_j$ that allow the ordering to be removed. In each iteration, constraints are checked in order from the beginning of the plan. A block, once added into π_{bdp}, will not be removed to accommodate another block that partially overlaps with the existing block throughout the procedure, even if the later (rejected) block could produce more deordering than the one created earlier. Since the choice of deordering to apply is greedy, the result is not guaranteed to be optimal. If b'_i or b'_j cannot be added to the decomposition (because one or both of them partially overlaps with an existing block), we consider all blocks ordered immediately after b_i, and check if all these orderings can be broken simultaneously, using the union of the blocks returned by RESOLVE for each ordering constraint. (Symmetrically, we also check the set of blocks immediately before b_j, though this is only very rarely useful.) As an additional heuristic, we discard the two blocks if there is a basic ordering constraint between a step that is internal to one of the blocks (i.e., that has both preceding and following steps within the block) and a step outside the block. If the ordering can be removed, the inner loop exits and the ordering relation is updated with any new constraints between b'_i and blocks ordered after b_j and between b'_j and blocks ordered before b_i. This is done by checking for the three reasons (PC, CD and DP) based on the sets of propositions produced, consumed and deleted by b'_i and b'_j. The inner loop is then restarted, with ordering constraints that previously could not be broken checked again. This is done because removing ordering constraints can make possible the resolution of other constraints, since removal of orderings can change the set of steps intermediate between two steps.

The main loop repeats until no further deordering consistent with the current decomposition is found. If no ordering constraint is removed during an iteration of the main loop, the deordering procedure stops. To remove any ordering constraint, the inner loop has to add some block consistent with the current decomposition. The total number of blocks that can be added by deordering procedure is finite (it cannot be more than the number of subsets of the set of steps, i.e., $2^{|\mathcal{S}|}$). Each iteration of the main loop runs in polynomial time, but we do not know of a polynomial bound on the number of iterations. Our procedure is "any time", in the sense that if interrupted before running to completion, the result at the end of the last completed iteration is a block deordering of the plan.

2.6 Experimental Results and Discussion

We find block deordering useful for maximizing the deordering, found by the conventional deordering algorithm, that helps to achieve greater execution flexibility of a plan. However, in this thesis, the ultimate significance of block deordering is determined by how much we can exploit the additional structural information it provides to improve two post-plan generation tasks – (1) continuing plan quality optimisation (described in Chapter 4) and (2) macro generation for improving planners efficiency (described in Chapter 6). As a simple example, we can observe from Figure 2.3 that

Figure 2.8: An example of a block deordering a p.o. plan that decreases the flex value of the plan. A normal p.o. plan (left) having an ordering a $\prec$ e for the reason $\mathrm{Re}(a \prec e) = \{CD(x)\}$, and a block deordering (right) of this plan after capturing the block {e, f}. This deordering removes two orderings: a $\prec$ e and a $\prec$ f from the p.o. plan, and introduces three new orderings: f $\prec$ b, f $\prec$ c, and f $\prec$ d (even though there is no necessary reason for those orderings) due to the requirement that the block {e, f} not be interleaved. Thus, this block deordering of the p.o. plan decreases its flex value.

if b2 is ordered before b1, then b1 can be optimised by removing step s3.

This section describes the results of an experiment that was conducted to compare the "amount of deordering" done by the step-wise and block-structured plan deordering. In other words, we ran this experiment to investigate the usefulness of block deordering in achieving greater execution flexibility of a plan. To observe the other benefits of block deordering (i.e., usefulness in plan optimisation and macro generation), we ran other experiments that are described in the relevant chapters. We measure the *flexibility* [1], or "flex", of plans, after the standard deordering (described in Section 2.2.3) and also after block deordering to compare the amount of deordering. The flex of a p.o. plan is defined as the fraction of pairs of steps that are not (transitively) ordered. Thus, a higher flex value indicates a less strictly ordered plan, with a fully sequential plan having a flex of zero. We apply the same definition to block deordered plans. Recall that in a block deordered plan, all steps belonging to two ordered blocks are ordered by the requirement that blocks not be interleaved, even when there is no ordering between an individual pair of steps. This is taken into account when calculating the flex of a block deordered plan. Because of this, in fact, it is possible for block deordering of a partially ordered plan to decrease its flex value in some cases. Figure 2.8 shows an example of how a block deordering can decrease the flexibility.

The experimental setup is as follows. We took all domains from the sequential satisficing tracks of the past editions of the International Planning Competition (IPC), IPC 2000-2014, except for the "CaveDiving" and "CityCar" domains. (These two domains have conditional effects, which our implementation does not handle.) We also took the Alarm Processing for Power Networks (APPN) [Haslum and Grastien, 2011] and data set 2-nd of the Genome Edit Distance (GED) domain [Haslum, 2011]. The plans (7324 in total) used in this experiment as inputs to plan deordering are the plans produced by different planners participating in the IPC 2000-2014. We impose

[1] The term "flexibility" is used with different meanings by different authors. Nguyen and Kambhampati [2001]'s definition is equivalent to ours. Muise et al. [2012] use it for the number of linearisations of a p.o. plan.

a 1800 second time limit on the block deordering procedure. All experiments were run on 6-Core, 3.1GHz AMD CPUs with 6M L2 cache, with an 8 GB memory limit for every system.

Results are summarised by domain in Table 2.1. For each domain, we report the number of plans analysed, the number of plans for which block deordering led to an increase in flex, and the average (over all analysed plans) flex values after step-based deordering and after block deordering. In cases where the limit of 1800 seconds was reached ($\sim$ 0.83% of all plans), we take the flex of the plan after the last completed iteration. The plans that reach the deordering time limit are mostly from the Visitall, Openstacks, and Hiking domains; the length of plans in those domains is often a couple of thousand actions. Longer plans from the BlocksWorld, Barman, Openstacks, and GED domains often experience a high number of iterations of Algorithm 1, mostly because of performing a high degree of deordering.

Note that several domains are purely sequential, and do not permit any deordering of individual steps. (These have an average flex of zero after step deordering.) Yet, even in these domains, it is often possible to block deorder plans. The BlocksWorld, Pegsol, Sokoban, and Visitall domains belong to this category. As an example, Figure 2.9 visualises the execution of part of a plan from the Sokoban domain (the reference plan for problem #11 from the IPC6 satisficing track). Sokoban is a puzzle game, involving a man who must push boxes around on a grid, one at a time, to reach a goal configuration. This domain is purely sequential because actions in the planning encoding of the game move the man from one square to another; thus, every step has a causal link from the step immediately before, and no deordering of individual steps is possible. Block deordering, however, is: In the example plan, the blocks consisting of steps 3–4 and steps 5–28 are independent and can be unordered. As can be seen clearly from the visualisation, moving steps 3–4, as a block, to after step 28 does not invalidate the plan. There are also two blocks, consisting of steps 10–21 and 22–23, within the larger block 5–28, that can be deordered. Our algorithm found all these possibilities.

2.7 Summary

Deordering makes the structure of a plan explicit, showing us which parts are necessarily sequential (because of dependency or interference) and which are independent and non-interfering. Block deordering improves on this by creating an on-the-fly hierarchical decomposition of the plan, encapsulating some dependencies and interferences within each block. Considering blocks, instead of primitive actions, as the units of partial ordering thus enables further deordering of plans, including in cases where no deordering is possible using the standard, step-wise, partial order plan notion. In this chapter, we present a simple greedy algorithm to find block decompositions that can substantially increase the flex of deordered plans across many planning domains.

We use the plan structure information of a block decomposed plan for improving the quality of plans by identifying subplans that can be locally improved, and for

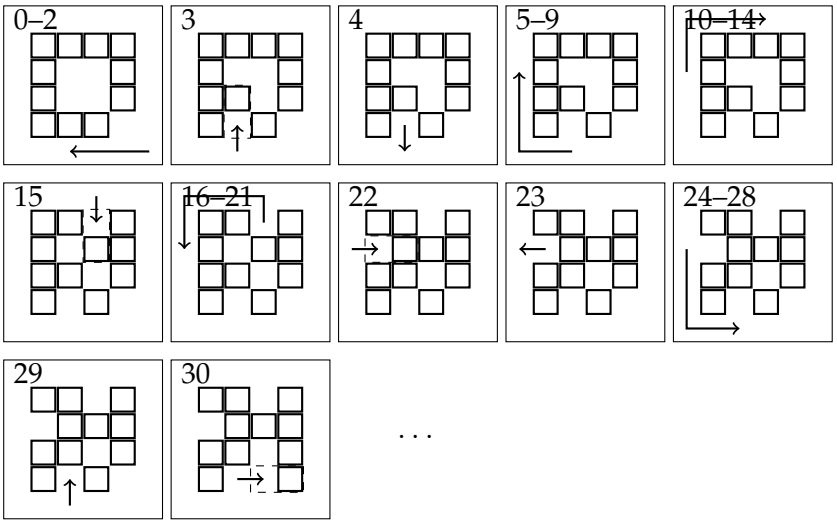


Figure 2.9: Visualisation of the execution of (part of) an example plan in the purely sequential Sokoban domain. Arrows show the movements of the man; dashed outlines show the movement of boxes that he pushes. The blocks consisting of steps 3–4 and 5–28 can be safely unordered, as can the blocks 10–21 and 22–23.

Domain	#plans	#inc.	Average flex after deordering	
			step	block
Airport (IPC4)	563	74	0.2985	0.3172
APPN	59	0	0.3963	0.3963
Barman (IPC7,8)	57	57	0.004	0.1885
BlocksWorld (IPC2)	381	284	0	0.078
ChildSnack (IPC8)	32	31	0.7225	0.7906
Cybersec (IPC6)	109	109	0.0252	0.4745
Depots (IPC3)	234	191	0.296	0.3599
Elevators (IPC6,7)	286	237	0.3093	0.3702
Floortile (IPC7,8)	163	161	0.0761	0.1063
Freecell (IPC2)	362	224	0.102	0.1126
GED [Haslum, 2011]	680	665	0.0056	0.3008
Hiking (IPC8)	208	191	0.0411	0.0543
Logistics (IPC2)	542	459	0.4857	0.4674
Maintenance (IPC8)	92	0	1	1
Nomystery (IPC7)	28	27	0.032	0.0974
Openstacks (IPC5,6,7,8)	994	992	0.08	0.1311
ParcPrinter (IPC6,7)	267	7	0.5634	0.5648
Parking (IPC7,8)	88	33	0.0031	0.0157
Pathways (IPC5)	113	73	0.4898	0.4952
Pegsol (IPC6,7)	325	284	0	0.2136
Rovers (IPC3)	187	143	0.6338	0.6962
Scanalyzer (IPC6,7)	343	200	0.2401	0.5853
Sokoban (IPC6,7)	205	174	0	0.0263
Storage (IPC5)	185	95	0.0845	0.2471
Tetris (IPC8)	22	16	0.4568	0.4042
Thoughtful (IPC8)	79	30	0.1313	0.1269
Tidybot (IPC7)	53	15	0.0699	0.0914
Transport (IPC6,7,8)	307	220	0.4469	0.4811
Visitall (IPC7,8)	72	72	0	0.1605
Woodworking (IPC6,7)	288	5	0.9114	0.9114
Overall	7324	5069	0.2347	0.3232

Table 2.1: Comparison of step-based and block-based plan deordering. For each domain, the first column shows the number of plans analysed, and the second the number of plans in which block deordering increased the flex value above that achieved by step deordering. The average flex values after step and block deordering are over all plans in each domain.

macro generation to improve planners efficiency. The next chapter describes the first use case of this block decomposition, i.e., our overall plan optimisation method, BDPO2, while our macro generation method is described in Chapter 6.

Chapter 3

LNS-based Plan Optimisation

One of the key challenges addressed in this thesis is to design a planning system that can continue to improve plan quality, even at larger time scales. Figure 3.1 (on page 45) shows that the current best planners or post-processing based plan improvement systems stop finding better plans at larger time scales; most of those planners often exhaust memory. We are inspired by the power of local search, in particular large neighborhood search (LNS), which is designed to explore large promising regions of the search space, rather than the whole search space. Thus, LNS requires much less memory compared to systematic search. Hence, we apply LNS in our plan improvement system, BDPO2, to improve a plan, one part at a time, rather than the plan as a whole, which helps us to achieve a continuing improvement of plan quality at any time scale.

There are several important questions that need to be addressed to realise our LNS-based continuing plan improvement system, such as: (1) what type, size, and number of subplans need to be extracted from a plan (and how), (2) how to pick the right subplanner for optimising a subplan when we have some alternatives at hand, (3) how to merge the replacement subplans, (4) how to decide a good point to restart the system in order to avoid local minima, (5) what can we learn from the improved subplans to prioritize the other potentially improvable subplans for optimisation from a large collection of candidate subplans, etc. This chapter gives an overview of our plan improvement system, where we address these questions. The performance of the overall system is shown by some experimental results.

This chapter is organised as follows. We present an overview of our complete plan improvement system in Section 3.2, which is followed by a discussion of the different settings that we have used for our empirical analysis (Section 3.3), and an overview of main results (Section 3.4). We then describe the subplanners used in BDPO2 (Section 3.6), the restart conditions (Section 3.7), the MERGE procedure for merging multiple improved subplans (Section 3.8), and the impact of plan decomposition in plan optimisation (Section 3.9). Finally, the existing works related to our plan optimisation process are discussed in Section 3.10.

The windowing strategies and ranking policies of BDPO2 are described in Chapter 4, while more details of the on-line adaptation methods used are presented in Chapter 5.

The contributions of this and the following two chapters, implemented in BDPO2, have been published as "Continuing Plan Quality Optimisation" in the the Journal of Artificial Intelligence Research [Siddiqui and Haslum, 2015]. BDPO2 extends our earlier system, BDPO [Siddiqui and Haslum, 2013b,a], mainly by using a variety of alternatives for each task: where BDPO used a single windowing strategy (with no ranking) and a single subplanner, BDPO2 has portfolios of window generation and ranking strategies and several subplanners. This improves the capability and robustness of the system. We have found that no single alternative (windowing strategy, subplanner, etc.) dominates all others across all problems (described in the following two chapters). Furthermore, BDPO2 introduces on-line adaptation for making the right choice for the current problem (e.g., which subplanner to select for each local optimisation attempt).

3.1 Introduction

The LNS strategy formulates the problem of finding a good neighbor as an optimisation problem, rather than simply enumerating and evaluating neighbours. This allows a much larger neighbourhood to be considered. LNS has been used very successfully to solve hard combinatorial optimisation problems like vehicle routing with time windows [Shaw, 1998] and scheduling [Godard et al., 2005]. Theoretical and experimental studies have shown that the increased neighborhood size may improve the effectiveness (quality of solutions) of local search algorithms [Ahuja et al., 2007], in spite of the computational complexity of exploring it. If the neighbourhood of the current solution is too small then it is difficult to escape from local minima. In this case, additional meta-heuristic techniques, such as Simulated Annealing or Tabu Search, may be needed to escape the local minimum. In LNS, the size of the neighborhood itself may be sufficient to allow the search process to avoid or escape local minima. In the LNS literature, the neighborhood of a solution is usually defined as the set of solutions that can be reached by applying first a "destroy" heuristic and then a "repair" method. The destroy heuristic selects a part of the current solution to be removed (unassigned), and the repair method rebuilds the destroyed part, keeping the rest of the current solution fixed. The destroy heuristic typically includes an element of randomness, enabling the search to explore modifications to different parts of the current solution.

We apply LNS to improve a plan, one segment at a time, rather than improving the plan as a whole. The main challenge in doing this is identifying good plan segments to re-optimise, and we need to do that by automatic and domain independent methods, rather than problem specific heuristics. For this, we make use of block deordering (described in detail in Chapter 2) which decomposes a plan into blocks that help to formulate subplans more likely to be improved. The role of the destroy heuristic in our system is played by the windowing strategies, which select candidate windows (subplans) for re-optimisation. We explore these windows systematically. Some LNS-based algorithms (e.g., [Ropke and Pisinger, 2006]; [Schrimpf et al., 2000])

use strategies, such as simulated annealing, that allow the local search to move to a neighbouring solution with a lower quality. We consider only strictly improving moves. However, in contrast to most LNS algorithms, we do not immediately move to a better plan and restart neighbourhood exploration after a local improvement has been found. Instead, we use delayed restarting, which allows a better solution to be found in one local search step by destroying and repairing multiple parts of the current plan. Experimentally, we found that delayed restarting produces better quality plans, and produces them faster, than immediate restarts (cf. Section 3.7). Further to delayed restart, our LNS post-processing uses on-line adaptation which helps the process to make the right choice for the current problem (e.g., which subplanner to select for each local optimisation attempt).

In summary, our plan improvement system can be viewed as a large neighborhood search in the space of valid plans, where we move from one plan to the next by replacing one or more subplans with the improved subplans. Since every successive plan is of better quality, the local search performs a hill climbing with low possibility of local minima due to the large neighborhood.

3.2 Overview of the Plan Optimisation System

BDPO2 is a post-processing-based plan quality optimisation system. Starting with an initial plan, it seeks to optimise parts of the plan, i.e. subplans, replacing them with lower-cost subplans. We refer to the subplans that are candidates for replacement as *windows*. When a better plan has been found and certain conditions are met, BDPO2 starts over from the new plan. This can be viewed as a local search, using the large neighborhood search strategy, in which the neighborhood of a plan is defined as the set of plans that can be reached by replacing a window with a new subplan. The local search is plain hill-climbing: each move is to a strictly better neighbouring plan. As in other LNS algorithms, searching for a better plan in the neighbourhood is done by formulating local optimisation problems, which are solved using bounded-cost subplanners.

Block deordering, described in Chapter 2, helps identify candidate windows by providing a large set of possible plan linearisations; the block decomposition is also used by some of our windowing strategies. Each window is a subsequence of some linearisation of the block deordered input plan. However, we represent a window in a slightly different way, by a partitioning of the blocks into the part to be replaced (w), and those ordered before (p) and after (q) that part.

Definition 8. *Let $\pi_{\text{bdp}} = (\mathcal{S}, \mathcal{B}, \prec)$ be a block decomposed p.o. plan. A* window *in π_{bdp} is a partitioning of $\mathcal{B}$ into sets p, w, q, such that π_{bdp} has a linearisation consistent with $\{b_p \prec b_w \prec b_q \mid \forall b_p \in p, b_w \in w, b_q \in q\}$.*

Each window defines a subproblem, which is the problem of finding a plan that can fill the gap left by removing the steps in w from a linearisation of π_{bdp} consistent with the window. This problem is formally defined as follows.

Definition 9. *Let $\pi_{\text{bdp}} = (\mathcal{S}, \mathcal{B}, \prec)$ be a block decomposed p.o. plan for planning problem Π, $\langle p, w, q \rangle$ a window in π_{bdp}, and $s_1, \ldots, s_{|p|}, s_{|p|+1}, \ldots, s_{|p|+|w|}, s_{|p|+|w|+1}, \ldots, s_n$ a linearisation of π_{bdp} consistent with that window. The* subproblem corresponding to $\langle p, w, q \rangle$, *Π_{sub}, has the same atoms and actions as Π. The initial state of Π_{sub}, I_{sub}, is the result of progressing the initial state of Π through $s_1, \ldots, s_{|p|}$ (i.e., applying $s_1, \ldots, s_{|p|}$ in I), and the goal of Π_{sub}, G_{sub}, is the result of regressing the goal of Π through $s_n, \ldots, s_{|p|+|w|+1}$.*

Theorem 7. *Let $\pi_{\text{bdp}} = (\mathcal{S}, \mathcal{B}, \prec)$ be a block decomposed p.o. plan for planning problem Π, $\langle p, w, q \rangle$ a window in π_{bdp}, Π_{sub} the subproblem corresponding to the window, and $s_1, \ldots, s_{|p|}, s_{|p|+1}, \ldots, s_{|p|+|w|}, s_{|p|+|w|+1}, \ldots, s_n$ the linearisation that Π_{sub} is constructed from. Let $\pi'_w = s'_1, \ldots, s'_k$ be a plan for Π_{sub}. Then $s_1, \ldots, s_{|p|}, s'_1, \ldots, s'_k, s_{|p|+|w|+1}, \ldots, s_n$ is a valid sequential plan for Π.*

Proof. The proof is straightforward. The subsequence $s_1, \ldots, s_{|p|}$ is applicable in the initial state of Π, I, and, by construction of Π_{sub}, results in the initial state of Π_{sub}, I_{sub}. Hence $s_1, \ldots, s_{|p|}, s'_1, \ldots, s'_k$ is applicable in I, and, again by construction of Π_{sub}, results in a state s_G that satisfies the goal of Π_{sub}, G_{sub}. Since G_{sub} is the result of regressing the goal of Π, G, through $s_{|p|+|w|+1}, \ldots, s_n$ in reverse, it follows that this subsequence is applicable in s_G, and applying it results in a state satisfying G. (For the relevant properties of regression, see, for example, [Ghallab et al., 2004], Section 2.2.2) □

The subproblem corresponding to a window $\langle p, w, q \rangle$ always has a solution, in the form of a linearisation of the steps in w. To improve plan quality, however, the replacement subplan must have a cost that is strictly lower than the cost of w, $\mathcal{C}(w)$. This amounts to solving *bounded-cost* subproblems. The subplanners we have used for this in BDPO2 are described in Section 3.6. We return to the question of when and how multiple windows within the same plan can be simultaneously replaced in Section 3.8.

Algorithm 2 describes how BDPO2 performs one step of the local search, by exploring the neighbourhood of the current plan. The first step is to block deorder the current plan (line 3). Next, optimisation using a bounded-cost subplanner is tried systematically on candidate windows (lines 4–19), until a restart condition is met (line 18), until no more local improvements are possible, or until a time limit is reached. A point of difference with other LNS algorithms is that we have used *delayed restart*, meaning that exploration of the neighbourhood can continue after a better plan has been found. This helps avoid local minima, by driving exploration to different parts of the current plan. The restart conditions, and the impact they have on the local search, are described in Section 3.7.

A key design goal of the procedure is to avoid unproductive time, meaning spending too much time in one step or trying to optimise one window while other options that could lead to an improvement are left waiting. Therefore, all steps are done incrementally, with a time limit on any step that could take an unbounded time.

A database (windowDB) stores each unique window extracted from the block deordered plan, and records its status (how many times optimisation of this window has been tried and the result), and structural summary information about the

Algorithm 2 The neighbourhood exploration procedure in BPDO2.

1: **procedure** BDPO2($\pi_{\text{in}}, t_{\text{limit}}$, banditPolicy, rankPolicy, optSubprob)
2: Initialize: $t_{\text{elapsed}} = 0$, $\pi_{\text{last}} = \pi_{\text{in}}$, trialLimit$[1...n] = 1$, windowDB $= \emptyset$
3: $\pi_{\text{bdp}} = \textsc{BlockDeorder}(\pi_{\text{in}})$
4: **while** $t_{\text{elapse}} < t_{\text{limit}}$ and π_{last} is not locally optimal **do**
5: **if** more windows needed **then**
6: $\textsc{ExtractMoreWindows}(\pi_{\text{bdp}}, \text{windowDB}, \text{optSubprob})$
7: $p = \textsc{SelectPlanner}(\text{banditPolicy})$
8: $w = \textsc{SelectWindow}(p, \text{rankPolicy}, \text{trialLimit}, \text{windowDB})$
9: **if** $w = null$ and no more windows to extract **then** trialLimit$[p]$ += 1
10: **if** $w = null$ **then continue**
11: w_{new}, searchResult $= \textsc{OptimiseWindow}(p, w)$
12: $\textsc{UpdateWindowDB}(p, w, w_{\text{new}}, \text{optSubprob}, \text{searchResult}, \text{windowDB})$
13: **if** $\mathcal{C}(w_{\text{new}}) < \mathcal{C}(w)$ **then**
14: $\pi_{\text{new}} = \textsc{Merge}(\pi_{\text{bdp}}, \text{windowDB})$
15: **if** $\mathcal{C}(\pi_{\text{new}}) < \mathcal{C}(\pi_{\text{last}})$ **then** $\pi_{\text{last}} = \pi_{\text{new}}$
16: $\textsc{UpdateBanditPolicy}(p, w, w_{\text{new}}, \text{searchResult}, \text{banditPolicy})$
17: $\textsc{UpdateRankPolicy}(p, \text{searchResult}, \text{rankPolicy})$
18: **if** $\mathcal{C}(\pi_{\text{last}}) < \mathcal{C}(\pi_{\text{in}})$ and restart condition is true **then**
19: **return** BDPO2 $(\pi_{\text{last}}, t_{\text{limt}} - t_{\text{elapse}}, \text{banditPolicy}, \text{rankPolicy}, \text{optSprob})$
20: **return** π_{last}

window. The window database is populated incrementally (lines 5–6), by applying different windowing strategies with a limit on the time spent and the number of windows added. The limits we have used are 120 seconds and 20 windows, respectively. This balances time between window extraction and optimisation, to prevent the procedure spending unproductive time. The windowing strategies are described in Chapter 4. We also compute a lower bound on the cost of any replacement plan for the window, using the admissible LM-Cut heuristic [Helmert and Domshlak, 2009]. A window is proven optimal if the current subplan cost equals this bound, or a previous attempt to optimise the window exhausted the bounded-cost search space. Already optimal windows are, of course, excluded from further optimisation. More windows are added to the database when the number of windows eligible to be selected for optimisation by any one subplanner (defined in the next paragraph) drops below a threshold. We have used 75% of the current window database size as the threshold.

The subplanner to use is selected using the UCB1 multi-armed bandit policy [Auer et al., 2002], which learns over repeated trials to select more often the subplanner that succeeds more often in finding improvements. The next window to try is chosen, among the eligible ones in the database, according to a ranking policy. Windows eligible for optimisation by the chosen subplanner are those that (1) are not already proven optimal; (2) have not been tried with the chosen subplanner up to its current trial limit; and (3) do not overlap with any improved window already

found. The ranking policy is a heuristic aimed at selecting windows more likely to be improved by the chosen subplanner. We use several ranking policies and switch from one to the next when the subplanner fails to find an improvement for a number of consecutive tries, since this indicates the current ranking policy may not be recommending the right windows for the current problem. The threshold we have used for switching the ranking policy is 13. (This is $2/3$ of the maximum number of windows added to the window database in each call to ExtractMoreWindows.) The ranking policies are described in Section 4.7. The subplanner is given a time limit, which is increased each time it is retried on the same window. We have used a limit of 15 seconds, increasing by another 15 seconds for each retry. A limit on the number of times it can be retried on the same window is kept for each subplanner. Initially set to 1, the limit is increased only when the subplanner has been tried on every window in the database (excluding windows that have already been proven optimal or that overlap with windows for which a better replacement has been found) and no strategy can generate more new windows (line 9). If a lower-cost replacement subplan for the window is found, this together with all improvements already found in the current neighbourhood are fed into the Merge procedure, which tries to combine several replacements to achieve a greater overall plan cost reduction. The Merge procedure is described in Section 3.8.

When the procedure restarts with a new best plan, the learned bandit policy for subplanner selection and the current ranking policy (for each subplanner) are carried over to the next iteration. We also keep a database of the subproblems (defined by their initial state and goal) whose plan cost has been proven optimal, to avoid trying fruitlessly to optimise them further. The window database, which contains only information specific to the current input plan, is reset.

3.3 Experiment Setup

Before presenting the overview of results, we outline below the three different experimental setups that we have used. For experiment setup 2 and 3 we used 182 large-scale instances from 21 IPC domains. The selection of domains and instances is described below. For experiment 1, we included additional medium-sized instances for a total of 219 instances from the same 21 domains. We used all domains from the sequential satisficing track of the 2008, 2011, and 2014 IPC, except for the CyberSec, CaveDiving and CityCar domains. (The CyberSec domain is too slow for our system to parse. The other two have conditional effects, which our implementation does not handle.) We also used the Alarm Processing for Power Networks (APPN) domain [Haslum and Grastien, 2011]. The plans used as input to BDPO2 are the plans produced by IBaCoP2 [Cenamor et al., 2014] in the 2014 IPC for the problems from that competition, and the best plan found by LAMA [Richter and Westphal, 2010, IPC 2011 version] in 1 hour CPU time for all other problems. We refer to these as the *base plans*. For experiments 2 and 3, we selected from each domain the 10 last instances for which a base plan exists. For domains that appeared in more than one

competition, we used instances only from the IPC 2011 set.

All experiments were run on 6-Core, 3.1GHz AMD CPUs with 6M L2 cache, with an 8 GB memory limit for every system. When comparing the anytime performance of BDPO2 and other systems that require an input plan, we count the time to generate each base plan as 1 hour CPU time. This is the maximum time allocated to generating each base plan; most of them were found much more quickly (by LAMA or IBaCoP2), but were not improved further during the remaining of the allotted 1 hour CPU time.

In our first experiment, we did not use the BDPO2 system. Instead, we ran each of two subplanners, PNGS and IBCS, for up to 30 seconds on every subproblem corresponding to a window extracted (by our six windowing strategies) from all base plans, excluding only subproblems for which the window was proven optimal by the lower bound obtained from the admissible LM-Cut heuristic [Helmert and Domshlak, 2009]. This experiment provided information to inform the design of the combined window extraction procedure, the window ranking policies, and other aspects of the system. We do not present results of this experiment here, but will refer to it in later chapters where we discuss these system components in detail.

In experiment 2, we compare BDPO2 and eight other anytime planners and plan optimisation systems: LAMA [Richter and Westphal, 2010, Fast Downward implementation]; AEES [Thayer et al., 2012b, Fast Downward implementation]; IBCS (as described in Section 3.6); Beam-Stack Search (BSS) [Zhou and Hansen, 2005]; PNGS (including "Action Elimination", [Nakhost and Müller, 2010]); IBaCoP2 [Cenamor et al., 2014]; LPG [Gerevini and Serina, 2002]; and Arvand [Nakhost and Müller, 2009]. BDPO2 uses PNGS and IBCS as subplanners, and is configured as described above. The other systems are described further in Section 3.10.

Each system was allowed 7 hours or more CPU time per problem. BDPO2 and PNGS both use the base plans as input, and IBCS and Beam-Stack Search both use the base plan cost as the initial cost bound. As mentioned above, we allocated 1 hour CPU time for generating each base plan. Therefore, when comparing these systems with planners starting from scratch (LAMA, AEES, IBaCoP2, LPG and Arvand), we add a 1 hour "start up" delay to their runtime. Beam-Stack Search, in our implementation, is much slower than the other planners used in the experiment (mainly because of how the LM-Cut heuristic is computed). Therefore, we ran it for up to 24 hours CPU time, and in reporting its results we divide its runtime by 4. In other words, the results shown are for a hypothetical implementation of Beam-Stack Search that does the same amount of search, but faster by a constant factor of 4.

Experiment 3 uses the same setup as experiment 2, except that the input to BDPO2 is the best plan found by running PNGS for up to 1 hour CPU time, with an 8 GB memory limit, on the base plans. (As mentioned previously, in the vast majority of cases PNGS runs out of memory in much less time than that, but in a few cases it does run up to the 1 hour limit.) We use this setup primarily to run different configurations of BDPO2 to analyse the impact of different designs (e.g., the planner selection and window ranking policies, immediate vs. delayed restart, and so on) in a setting where input plans are already of good quality. When comparing the anytime result of BDPO2 in this configuration to the other systems, we add 2 hours to its

runtime.

3.4 Overview of Results

Figure 3.1 shows the headline result, in the form of the average plan quality achieved by BDPO2 and other systems as time-per-problem increases. The IPC quality score of a plan is calculated as c^{ref}/c, where c is the cost of the plan and c^{ref} is the cost of the best plan for the problem instance found over all runs of all systems used in our experiments. Thus, a higher score reflects a lower-cost plan. c^{ref} is kept constant, i.e., that it is the cost of the best plan for the instance found by any system at the end of all experiments. This means that the conversion from plan cost to quality score is only a constant re-scaling (and inversion). The results in Figure 3.1 are from experiment 2 and 3, described in the previous section. It is the same as shown in Figure 1.2 (on page 5), but including results for all the compared anytime planning systems. None of the planners starting from scratch find a solution to all 182 problems: LAMA solves 155 problems, IBaCoP2 144, Arvand 134, AEES 87 and LPG 49. For these planners, the average quality score shown in Figure 3.1 is the average over only those problems for which they find at least one plan. (As previously mentioned, this is also the reason why the average quality sometimes falls: when a first plan, of low quality, for a previously unsolved problem is found, the average can decrease.) With this metric, a planner is not penalised for low coverage. Also, for the plan improvement systems, there is no "coverage" difference, since we use the quality of the base plan for any problem that a system has not improved on. Like the planners starting from scratch, none of the post-processing or bounded cost search methods improve on all base plans: BDPO2 finds a plan of lower cost than the base plan for 147 problems, PNGS for 133, IBCS for 66 and Beam-Stack Search for 24. For these systems, the average quality shown in Figure 3.1 is taken over all 182 problems, using the base plan quality score for those problems that a system has not improved on.

The majority of the compared systems show a trend similar to that of LAMA, i.e., improving quickly early on but then flattening out and ultimately stagnating. The reasons vary: Memory is a limiting factor for algorithms that perform exhaustive search, notably PNGS, which exhausts the 8 GB available memory before reaching the 7 hour CPU time limit for 93.7% of problems. LAMA and AEES do the same, respectively, for 67% and 50% of problems. On the other hand, planners that use limited-memory algorithms, such as Beam-Stack Search, LPG and Arvand (both of which use local search), never run out of memory and thus could conceivably run indefinitely. However, the rate at which they find plan quality improvements is small: From 4 to 7 hours, the average quality produced by LPG and Arvand increases by 0.0049 and 0.0094, respectively. (The latter excludes three problems that were solved by Arvand for the first time between 4 and 7 hours; including those brings the average down, making the increase less than 0.002.) The increase in average quality achieved by BDPO2, starting from the high-quality plans generated by PNGS from the base plans, over the same time interval is 0.0115.

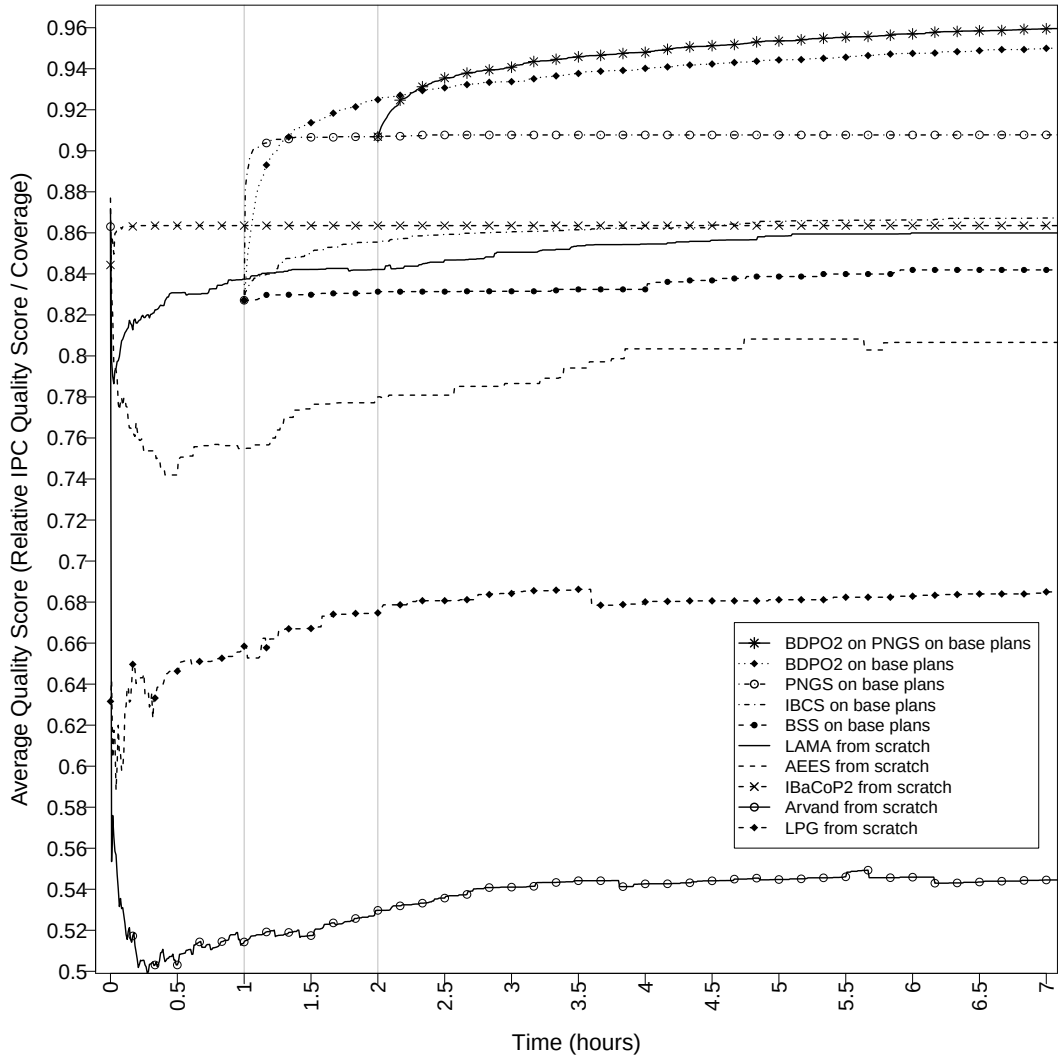


Figure 3.1: Average IPC quality score as a function of time per problem, on a set of 182 large-scale planning problems. The quality score of a plan is c^{ref}/c, where c is the cost of the plan and c^{ref} is the cost of the best plan for the problem found by any system at the end of all experiments. Hence a higher score represents better plan quality. The LAMA, AEES, LPG, Arvand and IBaCoP2 planners start from the scratch, whereas the post-processing (PNGS, BDPO2) and bounded cost search (IBCS, Beam-Stack Search) methods start from a set of base plans; their curves are delayed by 1 hour, which is the maximum time allocated to generating each base plan. The experimental setup is described in detail in Section 3.3.

We draw two main conclusions: First, BDPO2 achieves the aim of continuing quality improvement even as the time limit grows. In fact, it continues to find better plans, though at a decreasing rate, even beyond the 7 hour time limit used in this experiment. Second, the combination of PNGS and BDPO2 achieves a better result than either does alone. Partly this is because they work well on different sets of problems and the figure is showing an average, but BDPO2 sometimes produces a better result when started with the best plan found by PNGS also in domains where BDPO2 already outperforms PNGS when both start from the same base plans (e.g., Elevators and Transport). However, we have also seen the opposite in some domains (e.g., Floortile and Hiking), where starting BDPO2 with a worse input plan often yields a better final plan. This phenomenon has also been observed by Xie et al. [2013]. Figure 3.2 provides a more detailed view of the above observations. It shows for each problem the cost of the best plan found by each system at the 7 hour total time limit, scaled to the interval between the base plan cost and the highest known lower bound (HLB) on any plan for the problem. (Lower bounds were obtained by a variety of methods, including several optimal planners; cf. [Haslum, 2012].) 18 of the 182 problems are excluded from Figure 3.2: in 3 cases, the base plan cost already matches the lower bound, so no improvement is possible; for another 15 problems, no method improves on the base plans within the stipulated time. (The Pegsol domain does not appear in the graph, because all base plans but one are optimal, and no method improves on the cost of the last one.)

Table 3.1 provides a different summary of the information in Figure 3.2, showing for each domain and system the percentage of instances for which it found a plan with a cost (1) matching the best plan for that instance; (2) strictly better than any other method; and (3) matching the lower bound, i.e., known to be optimal. In aggregate, the combination of BDPO2 after PNGS over the base plans achieves the best result on all three measures. However, in 5 domains (GED, Hiking, Openstacks, Parking, and Tidybot), LAMA finds more plans that are strictly better than any other method. We have tried using LAMA as one of the subplanners in BDPO2, but this does not lead to better results overall. In some domains, such as Openstacks and GED, the smallest improvable subplan is often the whole, or almost the whole, plan, and LAMA finds an improvement of the plan only after searching for a longer time. Although BDPO2 increases the time limit given to subplanners with each retry, the average time limit, across all local optimisation attempts in this experiment, is only 18.48 seconds. Thus, our strategy of searching for quick improvements of plan parts does not work in these domains.

3.5 Combining Plan Optimisation Methods

Recall from Figure 3.1 that BDPO2 achieves its best result when coupled with other planning techniques, where the input to BDPO2 is the best plan found by running PNGS for up to 1 hour CPU time on the base plans. (The IBaCoP2 and LAMA planners have been used to generate the base plans, as mentioned in Section 3.3.)

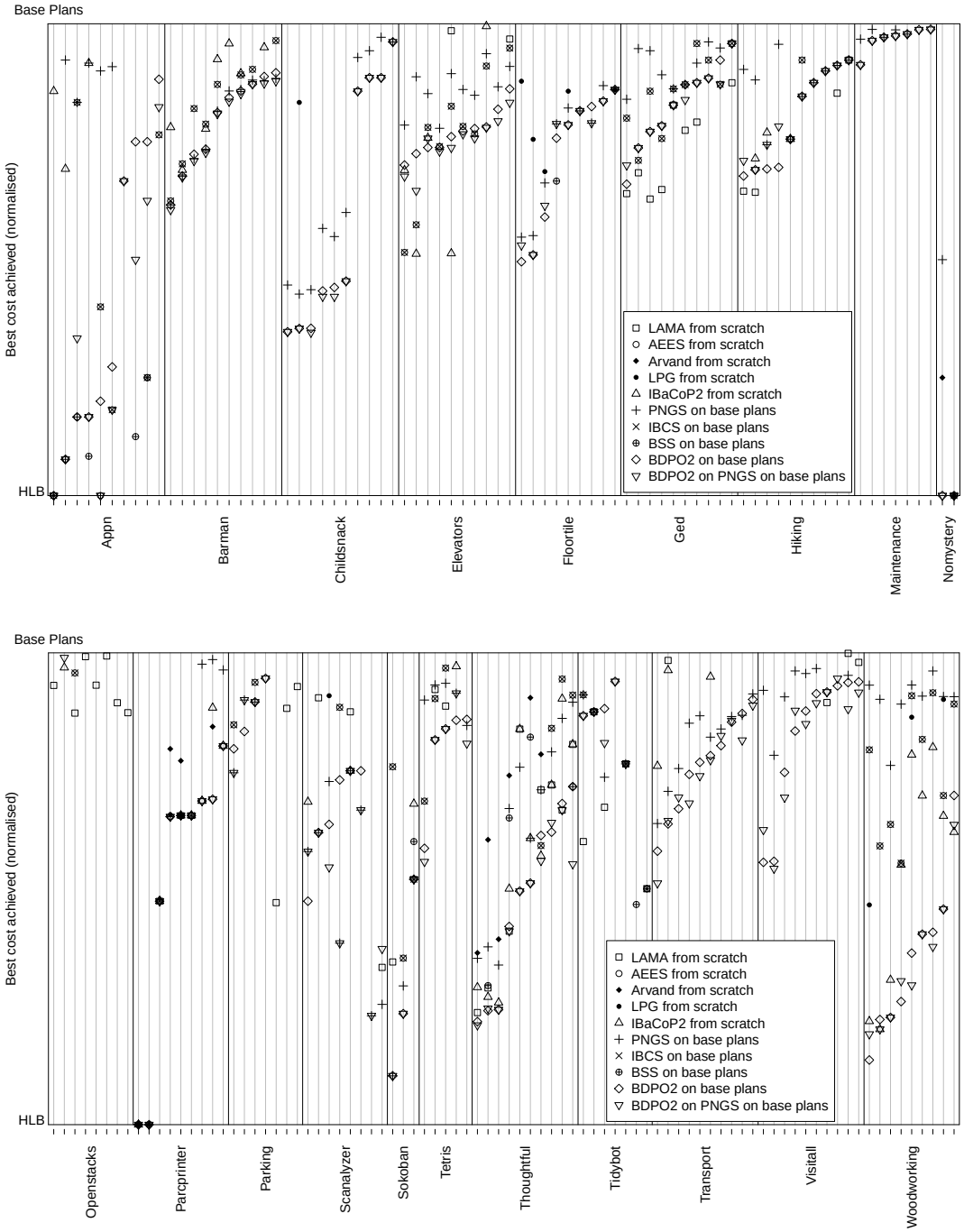


Figure 3.2: Best plan cost, normalised to the interval between the cost of the base plan and the corresponding highest known lower bound, achieved by different anytime plan optimisation methods in experiment 2, and by BDPO2 in experiments 2 & 3 (see Section 3.3).

Domains	BDPO2 on PNGS			BDPO2			LAMA			AEES			Arvand			LPG			IBCS			BSS			PNGS			IBaCoP2		
	=	<	⋆	=	<	⋆	=	<	⋆	=	<	⋆	=	<	⋆	=	<	⋆	=	<	⋆	=	<	⋆	=	<	⋆	=	<	⋆
Appn	50		20	40		10				40		10							40		10	70	20	20						
Barman	100	90		10																										
Childsnack	100	30		70																					10					
Elevators	60	60		10	10					10									10									20	20	
Floortile	67			78	22											11						11	11		33					
GED	30			20			80	70		10									10											
Hiking	50			70	20		40	30																	50			10		
Maintenance	100			100																					29					
Nomystery	100		100	100		100				50		50	50		50				50		50	50		50						
Openstacks							88	88																				12	12	
Parcprinter	100		22	100		22				33			11		11	33		22	33			67		22				67		11
Parking	43			43	14		43	43																	29					
Scanalyzer	75	12		38	12																				75	12		12		
Sokoban	100			100						33									33						67					
Tetris	80	40		60	20																									
Thoughtful	80	30		50	20		20																							
Tidybot	43			29			71	29		43									43			29	14		14					
Transport	60	60		40	40																									
Visitall	60	60		30	30		10	10																						
Woodworking	70	30		50	20																							20	10	
Overall	66	24	4	47	12	3	18	14		8		1	1		1	2		1	8		1	12	2	3	13	1		8	2	1

Table 3.1: For each plan improvement method, the percentage of instances where it found a plan of cost matching the best plan (=); found a plan strictly better than any other method (<); and found a plan that is known to be optimal, i.e., matched by the highest lower bound (⋆). The percentage is of the same instances in each domain shown in Figure 3.2. (Zeros are omitted to improve readability.) "BDPO2 on PNGS" is the result of BDPO2 in experiment 3; the other results are from experiment 2 (see Section 3.3).

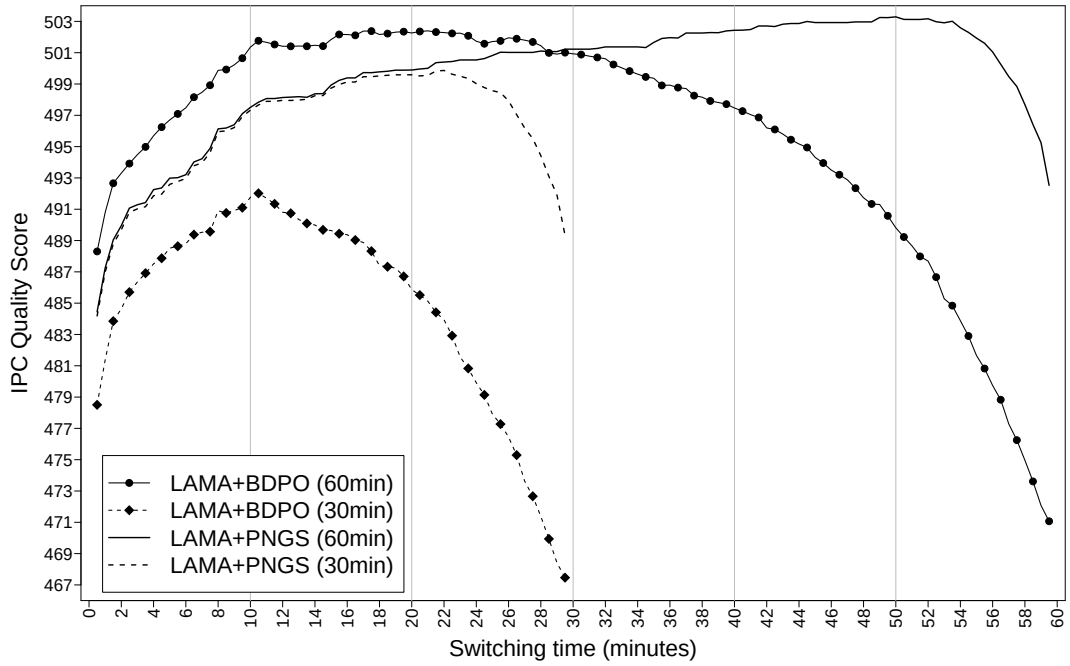


Figure 3.3: Impact of switching from LAMA to BDPO and LAMA to PNGS at different times during 30 and 60 minutes total runtimes [Siddiqui and Haslum, 2013a].

Given a bound on total runtime, it is critical to decide the best point to switch from one planning method to BDPO2 (or any other planning method).

We conducted a different experiment with an earlier version of BDPO2, called BDPO [Siddiqui and Haslum, 2013a], to observe the impact of switching plan optimisation methods on overall plan quality optimisation. In that experiment, we switched from LAMA to BDPO and LAMA to PNGS at different times during the total runtimes of 30 and 60 minutes under 3Gb memory per problem. The BDPO system uses IBCS and rule-based as the only subplanner and windowing strategy, respectively, and does not use any on-line adaptation. We used problems from all the domains (except Tidybot and Cybersec) from the 2008 and 2011 IPC. For domains that appeared in both the 2008 and 2011 IPC, we used only the instances from 2011 that were new in that year. Also the problem set included the data set 2-nd of the Genome Edit Distance (GED) domain [Haslum, 2011], and the Alarm Processing for Power Networks (APPN) domain [Haslum and Grastien, 2011].

The result of the above experiment is shown in Figure 3.3. It is apparently clear from the figure that allocating relatively more time to BDPO than LAMA leads to a better result (as measured by the summed IPC quality score). The LAMA+BDPO combination achieves its best score when switching at around 10 and 20 minutes during the 30 and 60 minutes total runtimes, respectively. In contrast, running PNGS for relatively shorter time (less than 10 minutes) after a long run of LAMA gives the best score in their combination for both runtimes. In 30 minutes total runtime, switching to PNGS is better than switching to BDPO. However, if the total runtime extends to 1 hour, the LAMA+BDPO combination improves its best score by 10.38 units compared to its best score in 30 minutes total runtime, whereas LAMA+PNGS improves that by 3.43 units only. This suggests that the best result within a stipulated amount of time could be achieved by a sequential portfolio of LAMA-PNGS-BDPO, where PNGS is run for a short time, and BDPO for relatively longer time. The overall experiment shown in Figure 3.1 also confirms that BDPO2 achieves the best result by using this portfolio.

3.6 Subplanners Used for Window Optimisation

The subplanners used by BDPO2 are used to find a plan for the window subproblem, as stated in Definition 9, with a cost less than the cost of the current window, $\mathcal{C}(w)$. We have considered three subplanners:

(1) Iterated bounded-cost search (IBCS), using a greedy search with an admissible heuristic for pruning.

(2) Plan neighbourhood graph search (PNGS), including the "action elimination" technique [Nakhost and Müller, 2010].

(3) Restarting weighted A* [Richter et al., 2010], as implemented in the LAMA planner.

However, in the experimental setups described in the previous section, BDPO2 uses only two subplanners, IBCS and PNGS. There are two reasons for choosing these two: First, they show good complementarity across domains. For example, IBCS is significantly better than PNGS in the APPN, Barman, Floortile, Hiking, Maintenance, Parking, Sokoban, Thoughtful, and Woodworking domains, while PNGS is better in the Elevators, Scanalyzer, Tetris, Transport, and Visitall domains. Second, the learning policy that we use for subplanner selection learns faster with a smaller number of options. Therefore, adding a third subplanner will only improve the overall performance of BPDO2, given a limited time per problem, if that subplanner complements the other two well, i.e., it performs well on a significant fraction of instances where both the other two do not. On the set of benchmark problems used in our experiment, this was not the case. (A different set of benchmarks could of course yield a different outcome.) An experiment comparing the effectiveness of all three subplanners, individually as well as the combination of IBCS and PNGS under the learning policy, in BDPO2 is presented in Chapter 5.

To solve the bounded-cost problem, IBCS uses a greedy best-first search guided by the unit-cost FF heuristic, pruning states that cannot lead to a plan within the cost bound using the f-value based on the admissible LM-Cut heuristic [Helmert and Domshlak, 2009]. It is implemented in the Fast Downward planner. The search is complete: if there is no plan within the cost bound, it will prove this by exhausting the search space, given sufficient time and memory. The bounded-cost search can return any plan that is within the cost bound. To get the best subplan possible within the given time limit, we iterate it: whenever a plan is found, as long as time remains, the search is restarted with the bound set to be strictly less than the cost of the new plan.

PNGS [Nakhost and Müller, 2010] is a plan improvement technique. It searches a subgraph of the state space around the input plan, limited by a bound on the number of states, for a lower cost plan. If no better plan is found the exploration limit is increased (usually doubled); this continues until the time or memory limit is reached. Like with IBCS, we iterate PNGS to get the best subplan possible within the given time limit. If it improves the current subplan, the process is repeated around the new best plan.

LAMA [Richter and Westphal, 2010] finds a first solution using greedy best-first search. It then switches to RWA* [Richter et al., 2010] to search for better quality solutions.

3.7 Restart

The restart condition determines a trade-off between exploring the neighbourhood of the current solution and continuing the local search into different parts of the solution space. The most obvious choice, and the one used in other LNS algorithms, is to restart with the new best solution as soon as one is found. We call this *immediate* restart. However, we have found that continuing to explore the neighbourhood of

the current plan even after a better plan has been found, and adding together several subplan improvements as described in Section 3.8 below, often produces better results. We call this *delayed* restart.

Setting the right conditions for when to make a delayed restart is critical to the success of this approach. We have used a disjunction of two conditions: First, if the union of improved windows found in the neighbourhood covers 50% of the steps in the input plan. Recall that when we continue the exploration loop (Algorithm 2) after an improvement has been found, windows that overlap with any already improved window are excluded from further optimisation. This drives the procedure to search for improvements to different parts of the current plan, and helps avoid a certain "myopic" behaviour that can occur with immediate restarts: When restarting with the new best plan, we get a new block decomposition and a new set of windows; this can lead to attempting to re-optimise the same part of the plan that was just improved, even over several restarts, which may lead to a local optimum that is time-consuming to escape. The second condition is that 39 consecutive subplanner calls have failed to find any further improvement. The threshold of 39 is three times the threshold for switching the ranking policy. This means that after 39 attempts we have tried to optimise the top 13 windows, among the remaining eligible ones, recommended by all ranking policies, without success. This suggests there are no more improvable windows to be found, or that none of our ranking policies are good in the current neighbourhood. Making a restart at this point allows the exploration to return to parts of the plan that intersect already improved windows, thus increasing the set of eligible windows.

The average plan quality, as a function of time-per-problem, achieved by BDPO2 using immediate restart and delayed restart based on the conditions above is shown by the top two lines in Figure 3.4 (page 56). In this experiment, both configurations were run using the same setup as experiment 3, described in Section 3.3 (page 42). As can be seen, delayed restart yields better results overall. However, we found immediate restart to work better for a few instances, especially in the Visitall and Woodworking domains, where BDPO2 with immediate restart found a better final plan for nearly 20% of the instances.

The average number of iterations (i.e., steps in the LNS) done by BDPO2 using the delayed restart condition is 3.48 per problems across all the domains considered in the experiment; the highest average in a single domain is 8.7, in Thoughtful solitaire. With immediate restart the average over all domains increases to 4.87. In other words, both configurations of BDPO2 spend significant time exploring the neighbourhood of each plan. The anytime performance curve in Figure 3.4 (page 56) shows that the additional time spent in each neighbourhood when using delayed restarts pays off.

3.8 Merging Improved Windows

Delayed restarting would not have any benefit without the ability to simultaneously replace several improved windows in the current plan. The improved windows are

Algorithm 3 Merge Improved Windows

1: **procedure** MERGE(π_{bdp}, windowDB)
2: Initialise $\hat{\pi}_{\text{bdp}} = \pi_{\text{bdp}}$
3: W = improved windows from windowDB sorted by cost reduction ($\mathcal{C}(w) - \mathcal{C}(w_{\text{new}})$)
4: **while** $W \neq \emptyset$ **do**
5: $(\langle p, w, q\rangle, w_{\text{new}})$ = pop window with highest $\mathcal{C}(w) - \mathcal{C}(w_{\text{new}})$ from W
6: $\hat{\pi}_{\text{bdp}} = \text{REPLACEIFPOSSIBLE}(\hat{\pi}_{\text{bdp}}, \langle p, w, q\rangle, w_{\text{new}})$
7: $W = \text{REMOVECONFLICTINGWINDOWS}(W, \hat{\pi}_{\text{bdp}})$
8: **return** $\hat{\pi}_{\text{bdp}}$

always non-overlapping (because once a better subplan for a window is found, windows that overlap with it are no longer considered for optimisation) but their corresponding subproblems may have been generated from different linearisations of the block deordered plan. Because of this, the replacement subplans may have additional preconditions or delete effects that the replaced windows did not, or lack some of their add effects. Thus, there may not be a linearisation that permits two or more windows to be simultaneously replaced.

The MERGE procedure shown in Algorithm 3 is a greedy procedure. It maintains at all times a valid block deordered plan ($\hat{\pi}_{\text{bdp}}$), meaning that each precondition of each block is supported by a causal link with no active threat. (Recall that a "block" in this context can be block that consists of a single step.) Initially, this is the input plan (π_{bdp}), for which causal links, and other ordering constraints, are computed by block deordering. The procedure gets the improved windows (W) from the window database, and tries to replace them in the current plan $\hat{\pi}_{\text{bdp}}$ in order of their contribution to decreasing plan cost, i.e., the cost of the replaced window ($\mathcal{C}(w)$) minus the cost of the new subplan ($\mathcal{C}(w_{\text{new}})$). The first replacement always succeeds, since, by construction of the subproblem, there is a linearisation of the input plan in which w_{new} is valid (cf. Theorem 7). Subsequent replacements may fail, in which case MERGE proceeds to the next improved window in W. Since replacing a window with a different subplan may impose new ordering constraints, any remaining improved windows that conflict with partial order of the current plan are removed from W.

The REPLACEIFPOSSIBLE function takes the current plan ($\hat{\pi}_{\text{bdp}}$), and returns an updated plan (which becomes the current plan), or the same plan if the replacement is not possible. The replacement subplan (w_{new}) is made into a single block whose steps are totally ordered. The preconditions and effects of this block, and those of the replaced window (w), are computed according to Definition 5 (page 19). For any atom in $\text{pre}(w_{\text{new}})$ that is also in w, the existing causal link is kept; likewise, causal links from an effect in $\text{add}(w)$ that are are also in $\text{add}(w_{\text{new}})$ are kept. These links are unthreatened and consistent with the order, since the plan is valid before the replacement. For each additional precondition of the new subplan, $m \in \text{pre}(w_{\text{new}}) \setminus \text{pre}(w_i)$, and for each causal link $\langle b_p, m, b_c\rangle$ in $\hat{\pi}_{\text{bdp}}$ where the producer is in the replaced window ($b_p \in w$), the consumer is not ($b_c \notin w$), and the atom of the link is

not produced by the replacement subplan ($m \notin \text{add}(w_{\text{new}})$), a new causal link must be found. Given a consumer (b_c) and an atom it requires ($m \in \text{pre}(b_c)$), the procedure tries the following two ways of creating an unthreatened causal link:

(C1) If there is a block $b' \prec^+ b_c$ with $m \in \text{add}(b')$, and for every threatening block (i.e., b'' with $m \in \text{del}(b'')$), either $b'' \prec b'$ or $b_c \prec b''$ can be added to the existing plan ordering without contradiction, then b' is chosen, and the ordering constraints necessary to resolve the threats (if any) are added.

(C2) Otherwise, if there is a block b' with $m \in \text{add}(b')$ that is unordered w.r.t. b_c, and for every threatening block either $b'' \prec b'$ or $b_c \prec b''$ can be enforced, then b' is chosen, and the causal link (implying the new ordering $b' \prec b_c$) and threat resolution ordering constraints (if any) are added to the plan.

The two are tried in order, C1 first and C2 only if C1 fails. If neither rule can find the required causal link, the replacement fails. w_{new} may also threaten some existing causal links in $\hat{\pi}_{\text{bdp}}$ that w did not. For each threatened link, $\langle b_p, m, b_c \rangle$, the procedure tries to resolve the threat in three ways:

(T1) If the consumer b_c was ordered before w in the linearisation of the corresponding subproblem ($b_c \in p$), and $b_c \prec w_{\text{new}}$ is consistent, the threat is removed by adding this ordering.

(T2) If the producer b_p was ordered after w in the linearisation of the corresponding subproblem ($b_p \in q$), and $w_{\text{new}} \prec b_p$ is consistent, the threat is removed by adding this ordering.

(T3) If a new, unthreatened causal link supplying m to b_c can be found by one of the two rules C1 or C2 above, the threatened link is replaced with the new causal link.

The rules are tried in order, and if none of them can resolve the threat, the replacement fails.

Some non-basic ordering constraints between blocks not in w may disappear when w is replaced with w_{new}; likewise, some ordering constraints between w and the rest of the plan may become unnecessary, because w_{new} may not delete every atom that w deletes and may not have all preconditions of w, and thus can be removed. This may make pairs of blocks b, b' in the plan that were ordered before the replacement unordered, and thus create new threats. All such new threats are checked by ReplaceIfPossible, and if found are resolved by restoring the ordering constraint that was lost.

Lemma 8. *If the current plan $\hat{\pi}_{bdp}$ is valid, and w_{new} solves the subproblem corresponding to window $\langle p, w, q \rangle$, the plan returned by* ReplaceIfPossible *is valid.*

Proof. The procedure ensures that every precondition of every step is supported by a causal link with no active threat: such a link either existed in the plan before replacement (and any new threats to it created by the replacement are resolved by ordering constraints), or was added by the procedure. Thus, if the replacement succeeds, the resulting plan is valid according to Theorem 2. If the replacement fails, the plan returned is the current plan, $\hat{\pi}_{\text{bdp}}$, unchanged, which is valid by assumption. □

Theorem 9. *If the input plan, π_{bdp} is valid, then so is the plan returned by* MERGE.

Proof. Immediate from Lemma 8 by induction on the sequence of accepted replacements. □

3.9 Impact of Plan Decomposition in Plan Optimisation

The neighbourhood explored in each step of the LNS in BDPO2 is defined by substituting improved subplans into the current plan. Each subplan considered for local optimisation is a subsequence of some linearisation of the block deordering of the current plan. Obviously, we can also restrict windows to be consecutive subsequences of the totally ordered input plan; in fact, similar approaches to plan optimisation have adopted this restriction [Ratner and Pohl, 1986; Estrem and Krebsbach, 2012; Balyo et al., 2012a]. In this section, we address the question of how much the block deordering contributes to the performance of BDPO2.

In the preliminary experiment (using setup 1, as described in Section 3.3 on page 42) we observed that more than 75% of the subproblems for which an improved subplan was found correspond to a non-consecutive part of the sequential input plan. However, this in itself does not prove that optimising only the 25% of subplans that can be found without deordering would not lead to an equally good end result. Therefore, we conducted another experiment, using the same setup as experiment 3 (described in Section 3.3). In this experiment, we ran BDPO2 separately with different degrees of plan decomposition: (1) With block deordering (as in the default BDPO2 configuration, and the one used in experiments 2 and 3 presented in Section 3.4 on page 44). (2) With standard, i.e., step-wise, plan deordering only. In this configuration, we used Kambhampati and Kedar's [1994] algorithm (described in Section 2.2.3) for plan deordering. (3) Without any deordering, i.e., passing the totally ordered input plan directly to the LNS process. In addition, each of these configurations was run once with immediate restarting and once with delayed restarting, as described in Section 3.7.

Figure 3.4 shows the average IPC plan quality score as a function of time-per-problem achieved by each of these configurations of BDPO2. It shows a simple and clear picture: With immediate restart, LNS applied to block deordered plans outperforms LNS applied to step-wise deordered plans, which in turn outperforms its use on totally ordered plans. The total improvement, as measured by the increase in the average IPC plan quality score, achieved by BDPO2 without deordering is 28.7% less than what is achieved by the best configuration. We can also see that deordering is an enabler for delayed restarting: With either block or step-wise deordering, delayed restarting further boosts the performance of the LNS-based plan optimisation approach, while on totally ordered plans it has no significant effect.

Deordering increases the number of linearisations and therefore enables many more distinct candidate windows to be created. However, recall that BDPO2's neighbourhood exploration procedure (Algorithm 2) interleaves incremental window generation with optimisation attempts; many of the windows that could be generated

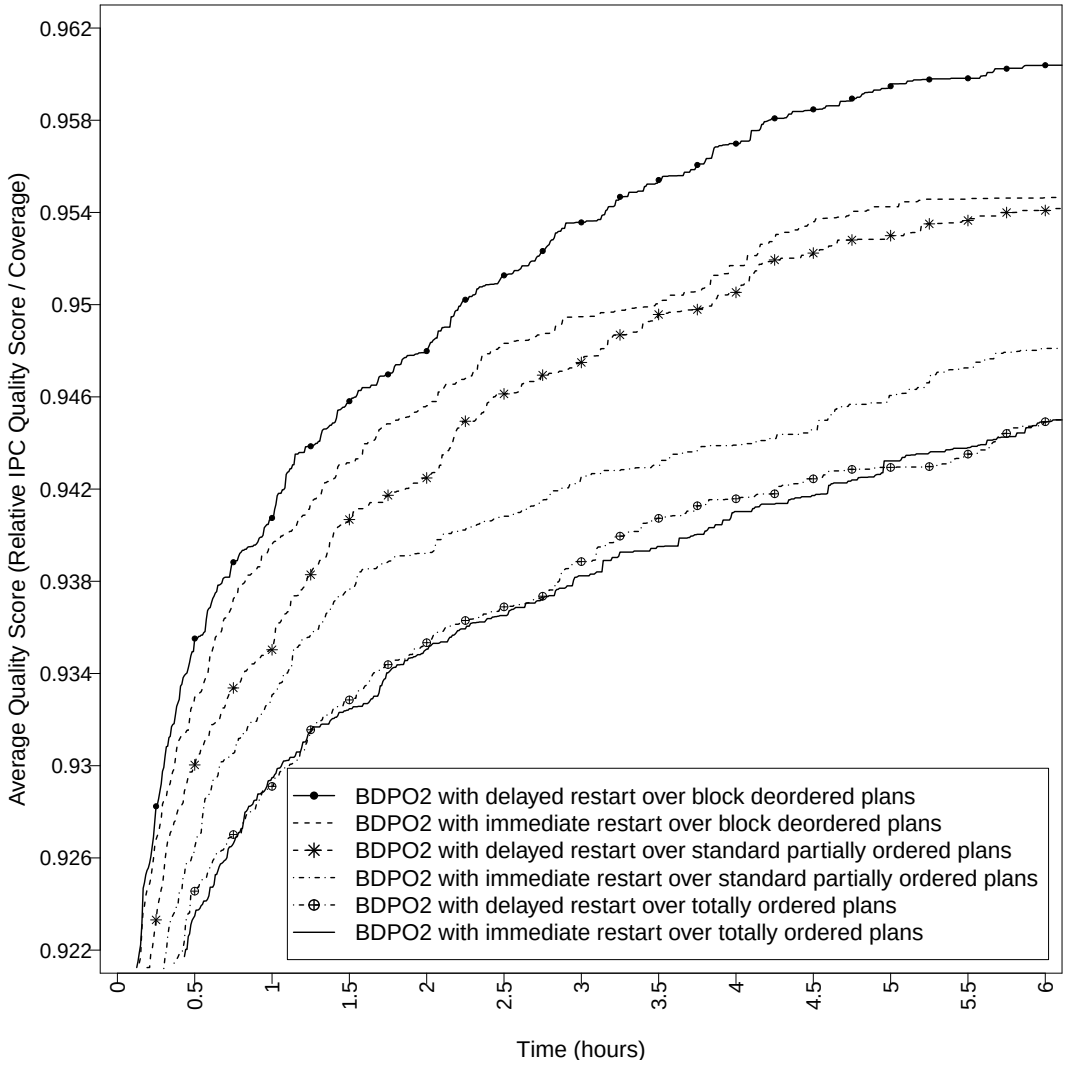


Figure 3.4: Average IPC quality score as a function of time per problem for BDPO2 applied to the totally ordered input plan; the standard (step-wise) deordering of the plan; and the block deordering of the plan. For each plan type, the system was run in two configurations: once with delayed restarting and once with immediate restarting (cf. Section 3.7 on page 51). This experiment was run with setup 3, as described in Section 3.3 on page 42. The quality score of a plan is calculated as before (see page 44). c^{ref} (as mentioned) is kept constant, i.e., that it is the cost of the best plan for the instance found by any system at the end of all experiments. The time shown here is only the runtime of BDPO2 (i.e., without the 2 hour delay for generating the input plans, as shown in Figure 3.1). Note also that the y-axis is truncated: All curves start from the average quality of the input plans, which is 0.907.

from the current plan may never be generated before a restart occurs. Thus, the average number of windows generated in each iteration does not reflect the difference in performance. (With block deordering, the average number of windows generated is 277.23, of which 183.19 remain after filtering, while on the totally ordered plans it is 376.8, and 149.94 after filtering; both are using immediate restart.) But deordering helps the windowing strategies generate windows that are more easily optimised. Recall that neighbourhood exploration will retry the same subplanner on the same window (with a higher time limit) only after all windows have been tried by that subplanner. The average number of optimisation attempts, using either subplanner, on each window selected for optimisation at least once, is around 1.7 when either block deordering or standard deordering is used on the input plan. Without any deordering, however, the average number of attempts is higher, and very high in a few domains: leaving out the highest 5% of neighbourhoods encountered, the average is slightly more than 2; in more than 10% of the plan neighbourhoods the average number of attempts is over 5, and the highest values are over 20 attempts, occurring in the APPN, Childsnack, ParcPrinter and Scanalyzer domains. In other words, generating windows from a totally ordered plan causes the procedure to spend, on average, more time on each window before an improving plan is found.

On the other hand, as noted in Section 3.4 on page 44, in some domains subplanners need more runtime to find better plans for improvable windows. In the current BDPO2 system, the subplanner time limit is increased only when a window is retried. A procedure that either attempts candidate windows more likely to be improved (for example, as indicated by the window ranking policies described in Section 4.7) more frequently, or varies the amount of time given to optimise each window may perform better. The optimal amount of deordering to do on each plan may well be different from problem to problem. But averaged across the set of benchmark problems, more deordering is unarguably better than none.

3.10 Related Work

We survey four areas of related work: Anytime search algorithms and post-processing approaches, which have in common with our approach the aim of continuing plan quality improvement; uses of local search in planning; and finally, uses of algorithm portfolios in planning.

3.10.1 Anytime Search

Large state-space search problems, of the kind that frequently arise in planning problems, often cannot be solved optimally because optimal search algorithms either exhaust memory before finding a solution or require a very long time to find a solution (when limited-memory algorithms are used). Anytime search algorithms try to deal with such problems by finding a first solution quickly, possibly using a greedy or suboptimal heuristic search, then continue (or restart) searching for a better quality

solution. Anytime algorithms are attractive because they allow users to stop computation at any time, i.e., after a "good enough" solution has been found, or after "too long" a wait. This contrasts with algorithms that require the user to decide in advance on a deadline, a suboptimality bound, or some other parameter that fixes the trade-off between time and solution qualty.

Bounded suboptimal search is the problem of finding a solution with cost less than or equal to a user specified factor w of optimal. Weighted A* (WA*) search [Pohl, 1970] and Explicit Estimation Search (EES) [Thayer and Ruml, 2011] are two algorithms of this kind that have been most used in planning. Iteratively applying any bounded suboptimal search algorithm with a lower value of w whenever a new best solution is found provides an anytime improvement of plan quality. Restarting WA* [Richter et al., 2010] does this, using a schedule of decreasing weights. RWA* is used by LAMA [Richter and Westphal, 2010] – the best-performing planner in the 2008 and 2011 IPC. LAMA finds the first plan using a greedy best-first search [Bonet and Geffner, 2001]. It also uses several search enhancements, like preferred operators and deferred evaluation [Richter and Helmert, 2009]. AEES [Thayer et al., 2012b] is an anytime version of EES. EES conducts a bounded suboptimal best-first search restricted to expanding nodes that may lead to a solution with a cost no more than a given factor w times optimal. Among the open nodes in this set, it expands the one estimated to have the fewest remaining actions between it and a goal. To achieve an anytime behavior, AEES lowers the value of w whenever a new best solution is found. The algorithm was shown to be particularly effective in domains with non-uniform costs, due to its use of estimates of both solution length and solution cost.

The bounded-cost search [Stern et al., 2011] problem, of which the subproblems solved in our approach are an example, requires finding a solution with a cost less than or equal to a user-specified cost bound; however, the aim of a bounded-cost search algorithm is to find such a solution as quickly as possible. The BEES and BEEPS algorithms [Thayer et al., 2012a] adapt EES to the setting of bounded-cost search. Iteratively applying any bounded-cost search algorithm with a bound less than the cost of the best solution found so far provides anytime quality improvement. This is what the IBCS algorithm, used as one of the subplanners in BDPO2, does. The main difference between anytime BEES and BEEPS, and AEES is how these algorithms determine if a solution is likely to be improved upon the current incumbent. Neither BEES nor BEEPS expand a node whose admissible estimate of solution cost is larger than the cost of current best solution, whereas AEES may well expand the node. Beam-stack search (BSS) [Zhou and Hansen, 2005] is an extension of breadth-first branch-and-bound search. It expands nodes in breadth-first order, in which only the b (referred to as beam width) most promising nodes in each level of the search space are expanded. Like branch-and-bound search, it uses upper and lower bounds to prune the search space, and updates the upper bound each time it finds an improved solution. BSS remembers which nodes have not yet been expanded and returns to them later; it explores the entire search space below the chosen states before it backtracks on its decision. This may be time consuming, if, for example, a set of successor states is chosen from which no near-optimal solution can be reached.

In our experiment, we use Beam-stack search as a bounded-cost search by giving an upper bound of the cost.

Anytime search planners aim to provide continuing improvement of plan quality given more time, and often succeed in doing that in the early stages of the search. However, as we have observed in the results of our experiments, these algorithms often "stagnate", reaching a point where they do not find any better plans even after several hours of CPU time. (cf. Figure 3.1 and Section 3.3.) LAMA and AEES, for example, improved plans for 8.7% and 6.1%, respectively, of the total number of problems in our experiments during the second half of the runtime (i.e., from 3 hours to 6 hours), while BDPO2 found improved plans for 30.4% of problems in the same time. The reason for this is usually that these planners, which use systematic state-space searches, run out of memory. BSS, on the other hand, never exhausted the available memory in our experiment (using setup 2, as described in Section 3.3 on page 42). However, because of exploring the entire search space, it often fails to find any improvement within a long 6 hours time bound. It found plan quality improvements of 13.2% of the problems considered in that experiment, compared to that of 91.2% found by different other methods in the experiment.

3.10.2 Local Search

Local search explores a space by searching only a small neighbourhood of a current element in the search space for one that is, in some way, better, then moving to the neighbour and repeating the process. Compared to systematic search algorithms, the advantage of local search is that it needs much less memory: only one element of the search space, and its neighbours, are stored at any time. Therefore, local search algorithms are widely used to solve hard optimisation problems. However, local search algorithms cannot offer any guarantees of global optimality, or bounded suboptimality. In planning, local search has been used mainly to find plans quickly, and rarely to improve plan quality, though some of the post-processing methods discussed in the next section can be viewed as local searches.

FF [Hoffmann and Nebel, 2001] is a forward-chaining heuristic state space search planner. The heuristic used by FF estimates the distance from a state to the nearest goal state. FF uses a local search strategy, called enforced hill-climbing, that in each state uses a breadth-first search to find a "neighbour" state (which may be several steps away from the current state) with a strictly better heuristic value, i.e., that is believed to be closer to the goal. It then commits to that state and starts a new search for a neighbour with a better yet heuristic value. If the local search fails, due to getting trapped in a dead end, FF falls back on a complete best-first search algorithm. The RW-LS planning algorithm [Xie et al., 2012] is similar to FF's hill-climbing approach, but uses a combination of greedy best-first search and exploration by random walks to find a better next state in each local search step.

Nakhost and Müller [2009] developed a planning system called Arvand that uses random walk-based local exploration in conjunction with the FF search heuristic. They showed that Arvand outperforms FF for hard problems in many domains. The

execution of Arvand consists of a series of search episodes. Each episode starts by performing a set of random walks from the initial state. The endpoint of each of these random walks are evaluated using the heuristic function to choose the next state. The search episode then continues with a set of random walks from this state. This process then repeats until either a goal is found, or enough transitions are made without heuristic progress, in which case the process makes a restart. Arvand (in the IPC 2011 and 2014 versions) uses a plan post-processing system called Aras [Nakhost and Müller, 2010]. Aras iterates between two post-processing techniques (described in the next subsection): Action Elimination (AE) and Plan Neighborhood Graph Search (PNGS) until some time or memory limit is reached; the limit on the nodes in the neighbourhood graph is increased each time the neighbourhood graph phase begins. The 2014 version of Arvand uses "aggressive restarting" that performs restarts much more often then the earlier versions of Arvand.

The LPG planner [Gerevini and Serina, 2002] is based on local search in the space of "action graphs", which represent partial plans. The neighbourhood is defined by operators that modify an action graph, such as inserting or removing actions. The function that evaluates nodes in the neighbourhood combines terms that estimate both how far an action graph is from becoming a valid plan, termed "search cost", and the expected quality of the plan it may become. The choice of neighbour to move to also involves an element of randomness. LPG also performs a continuing search for better plans; in this, it is similar to the anytime search algorithms discussed in the last subsection. Whenever it finds a plan, the local search restarts with a partial plan obtained by removing some randomly selected actions from the current plan. A numerical constraint forcing the cost of the next plan to be lower is also added. This provides some guidance towards a better quality next plan.

A key difference between our use of local search and its previous uses in planning is that we carry out a local search (in particular a large neighborhood search) only in the space of valid plans. This permits the neighbourhood evaluation to focus exclusively on plan quality. Searching a space of partial plans (represented by states) as done in FF, or incomplete (invalid) plans, as done by LPG, requires neighbourhood evaluation to consider how close an element is to becoming a valid plan, and balancing that with quality.

3.10.3 Plan Post-Processing

By a post-processing method, we mean one that takes a valid plan as input and attempts to improve it, by making some modifications.

Nakhost and Müller [2010] proposed two post-processing techniques – Action Elimination (AE) and Plan Neighborhood Graph Search (PNGS). Action elimination identifies and removes some unnecessary actions from the given plan. PNGS constructs a plan neighborhood graph, which is a subgraph of the state space of the problem, built around the path through the state space induced by the current plan by expanding a limited number of states from each state on that path. It then searches for the least-cost plan in this subgraph. If this finds a plan better than the current,

the process is repeated around the new best plan; otherwise, the exploration limit is increased, until a time or memory limit is exceeded. Iterative Tunneling Search with A* (ITSA*) [Furcy, 2006] is similar to PNGS. ITSA* explores an area, called a tunnel, of the state space using A* search, restricted to a fixed distance from the current plan. These methods can be seen as creating a neighborhood that includes only small deviations from the current plan, but anywhere along the plan. In contrast, BDPO2 focuses on one section of the decomposed plan at a time, often grouping together different parts of the input plan, but puts no restriction on how much that section changes; hence, it creates a different neighbourhood. Our experiments show that the best results are obtained by exploring both neighbourhoods. For example, PNGS often finds some plan improvements quickly, but running it for an additional 6 hours improves its average IPC quality score, over that of the best plans it finds in the first hour, only by 0.08%. Running instead BDPO2, using PNGS as the only subplanner and taking the best plans found by PNGS in 1 hour as input, improves the quality score by 2.86% in 6 hours.

Ratner and Pohl [1986] used local optimisation for shortening solutions to sequential search problems. Their approach to subproblem selection is a simple sliding window over consecutive segments of the current path. Balyo et al. [2012b] also used a sliding window to minimise parallel plan length (that is, "makespan", assuming all actions have unit duration). We use block deordering to create many more candidate windows for local optimisation. This is very important for the success of BDPO2: In the experiment described in Section 3.9, where the input plans were of high quality (found by running PNGS with 1 hour runtime cutoff on the best plans found by running LAMA or IBaCoP2 with 1 hour runtime cutoff) as described in the experiment setup 3 in Section 3.3, the total quality improvements (measured by the average IPC quality score) found by BDPO2 without performing any block deordering (i.e., based on sequential input plans) is 28.7% less than that found by BDPO2 based on block deordered input plans.

The planning-by-rewriting approach [Ambite and Knoblock, 2001] also uses local modifications of partially ordered plans to improve their quality. Plan modifications are defined by domain-specific rewrite rules, which have to be provided by the domain designer or learned from many examples of both good and bad plans. Hence, this technique can be effective for solving many problem instances from the same domain. Using a planner to solve subproblems may be more time-consuming than applying pre-defined rules, but makes the process automatic. However, if we consider solving many problems from the same domain it may be possible to reduce average planning time by learning (generalised) rules from the subplan improvements we discover and using these where applicable to avoid invoking a subplanner.

3.10.4 Portfolio Planning and Automatic Parameter Tuning

A portfolio planning system runs several subplanners in sequence (or in parallel) with short timeouts, in the hope that at least one of the component planners will find a solution in the time allotted to it. Portfolio planning systems are motivated

by the observations that no single planner dominates all others in all domains, and that if a planner does not solve a planning task quickly, often it does not solve at all. Therefore, many of today's most successful planners run a sequential portfolio of planners [Coles et al., 2012].

Gerevini et al. [2009] introduced the PbP planner, which learns a portfolio over a given set of planners for a specific domain, as well as domain-specific macro-actions. Fast Downward Stone Soup (FDSS) [Helmert et al., 2011] uses a fixed portfolio, computed to optimise performance on a large sample of training domains, for all domains. IBaCoP2 [Cenamor et al., 2014] dynamically configures a portfolio using a predictive model of planner success.

Another recent trend is the use of automatic algorithm configuration tools, like the ParamILS framework [Hutter et al., 2009], to enhance planner performance on a specific domain. ParamILS does a local search in the space of configurations, using a suite of training problems to evaluate performance under different parameter settings. The combinatorial explosion caused by many parameters with many different values is managed by varying one parameter at a time. ParamILS has been used to configure the LPG planner [Vallati et al., 2011] and the Fast Downward planner [Fawcett et al., 2011]. The PbP2 portfolio planner [Gerevini et al., 2011], successor to PbP, includes a version of LPG customised to the domain with ParamILS in the learned portfolio.

BDPO2, of course, uses a portfolio of subplanners, and, as we have shown, selecting the right subplanner for the current problem is important (cf. Chapter 5). Much more important, however, is the focus on subproblems that our approach brings: comparing Figures 3.1 (page 45) and 5.3 (page 91), it is clear that using even a single subplanner within BDPO2 is more effective than using any of the subplanners on its own. The multiple window ranking policies used in BDPO2 (cf. Section 4.7) can also be viewed as a simple sequential portfolio. Compared to previous portfolio planners, the iterated use of subplanners, windowing strategies and other components in our approach offers a possibility to learn the best portfolio or configuration on-line; that is, rather than spend time on configuring the system using training problems, we can learn from the experience of solving several subproblems, while actually working on optimising the current plan.

Finally, although we have not explored it in great depth, the experimental results described in Section 3.5 suggest that combining different anytime search and post-processing methods, in what is effectively a kind of sequential portfolio (such as running BDPO2 on the result of running PNGS on the result of LAMA, as shown in that experiment), often achieves better quality final plans than investing all available time into any single method.

3.11 Summary

Plan quality optimisation, particularly for large problems, is one of the key research areas in automated planning. Anytime planning, which aims to generate a sequence

of successively higher quality plans given more time, is an attractive idea, offering the flexibility to stop the process at any point, such as when the best plan found is "good enough" or the wait for the next plan becomes "too long".

This chapter presents an approach to anytime plan improvement, and its realisation in the BDPO2 system. This approach uses large neighborhood search to improve a plan, one segment at a time. In doing so, a set of windowing strategies pairs with a set of off-the-shelf bounded-cost planning techniques to create a neighborhood of better quality plans. At the end of each search step, we restart with the best found neighboring plan. Windowing strategies extract plan segments (from a block deordering of the current plan) that are more likely to be improved, and queue those for optimisation (using bounded-cost planners) according to different ranking policies. Within each search step, the re-optimised segments are merged to form a better quality plan for the underlying problem. We enhance our LNS-based plan improvement system by incorporating (1) delayed restart to help finding a better solution in one local search step by optimising multiple parts of the current plan, and (2) on-line adaptation to help the system make the right choice (from a set of alternatives) for the current problem.

Experiments demonstrate two important observations: (1) BDPO2 achieves continuing plan quality improvement even at large time scales (several hours CPU time), when other anytime planners stagnate, and (2) BDPO2 achieves its best result when coupled with other planning techniques.

While this chapter introduces the concept of windowing, ranking, and on-line adaptation techniques, necessary to explain the overall plan improvement system, the next two chapters describe these techniques in more details with necessary background and experiments.

Chapter 4

Windowing Strategies and Ranking of Windows

The previous chapter described our overall plan optimisation system, BDPO2, that improves a block deordered plan one segment, termed as window, at a time using a large neighborhood search strategy. This chapter describes the generation and ranking strategies for these windows. For window generation, we use some heuristics to group together (successive or unordered) blocks into a window so that the formulated window is more likely to be improved. A window formed from a random subsequence of blocks of a block deordered plan is less likely to be improvable. In an experiment, where the input plans were of high quality (as described in experiment setup 3 in Section 3.3), the total quality improvements (measured by the average IPC quality score) found by formulating windows based on random subsequences of blocks is 17.1% less than that found by the heuristics-based windowing strategy. This chapter describes three different strategies to formulate windows, and experimentally evaluates their individual and combined impacts on the overall system performance. BDPO2 also uses different window ranking policies to rank the candidate windows generated by the windowing strategies, such that the more promising ones are optimised first. Like windowing strategies, we also justify the usefulness of window ranking policies with experimental results.

This chapter is structured as follows: First we describe separately the three windowing strategies in Sections 4.2-4.4, which is followed by a discussion of experimental results in Section 4.5 showing the individual and combined effects of the strategies on BDPO2. We discuss a possible extension of these strategies in Section 4.6, and the window ranking policies with corresponding experiments in Section 4.7.

4.1 Introduction

A window is a subplan of some linearisation of the block deordered plan, extracted in order to attempt local optimisation. Recall from Definition 8 (page 39) that a window is represented by a triple $\langle p, w, q \rangle$, where w is the set of blocks to be replaced, and p and q are the sets of blocks ordered before and after w, respectively, in the linearisation. A block decomposed p.o. plan can have many linearisations, producing

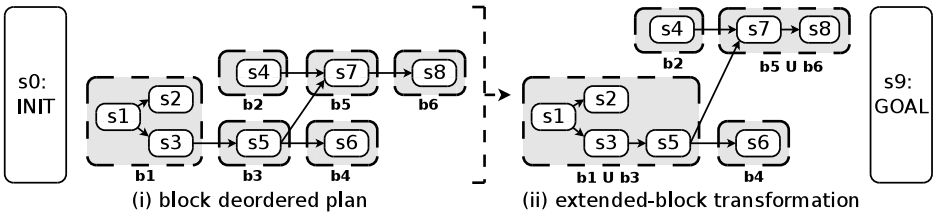


Figure 4.1: A block deordered plan and its transformation into extended blocks: blocks b1 and b3 are merged into a single block, as are blocks b5 and b6.

Algorithm 4 Computing extended blocks.

1: $\mathrm{B_{ext}} \leftarrow \mathrm{B_{basic}}$
2: **while** $\exists b_i, b_j \in \mathrm{B_{ext}} : \mathrm{IP}(b_j) = \{b_i\}, \mathrm{IS}(b_i) = \{b_j\}$ **do**
3: $\quad \mathrm{B_{ext}} \leftarrow \mathrm{B_{ext}} \cup \{b_i \cdot b_j\} \setminus \{b_i, b_j\}$

many possible windows – typically far too many to attempt to optimise them all. A *windowing strategy* is a heuristic procedure that extracts a reduced set of windows, hopefully including the most promising ones, in a systematic way.

We propose three windowing heuristics, called the *rule-based*, *cyclic thread*, and *causal followers* heuristics. Each of them is described in detail in the following subsections. Each procedure is applied over two types of block – *basic* and *extended*, one at a time. Basic blocks $\mathrm{B_{basic}}$ are the blocks generated by a block deordering. (For the purpose of windowing, any step that is not included in a block created by block deordering is considered to be a block on its own.) Extended blocks $\mathrm{B_{ext}}$ are created by merging basic blocks in the block deordered plan that form complete non-branching subsequences. If block b_i is the only immediate predecessor of block b_j, and b_j the only immediate successor of b_i, they are merged into one extended block. Algorithm 4 shows the formal procedure of extended block formation, which is further illustrated by an example shown in Figure 4.1. Note that blocks b5 and b2 are not merged into one extended block. This is because although b5 is the only immediate successor of b2, b2 is not the only immediate predecessor of b5. Extended blocks are useful because they allow some windowing heuristics to capture larger windows. It should be noted that if no deordering is possible, $\mathrm{B_{ext}}$ will consist of a single (extended) block encapsulating the whole plan. Our experimental results show that windows of different sizes are more useful in different domains. (Windows are characterized as large, small, or medium-sized with respect to plan length.) Larger windows, for example, are more likely to be improved in the Pegsol, Openstacks and Parcprinter domains, while optimising smaller windows is better in the Elevators, Transport, Scanalyzer and Woodworking domains.

A windowing strategy is a windowing heuristic applied to a block type. Thus, we use a total of six different strategies. Each of these strategies contributes some improvable windows that are not generated by any of the other strategies (cf. Section 4.5, and in particular Table 4.2). Thus, all of them are, in this sense, useful. On

Windowing heuristics	Generated		Not filtered out		Improved		Impr./Gen.	
	Basic	Ext.	Basic	Ext.	Basic	Ext.	Basic	Ext.
Rule-based	108	35	59	22	23	9	0.21	0.26
Cyclic thread	59	47	45	31	15.5	11	0.26	0.23
Causal followers	72	45	41	20	15.5	7	0.22	0.16

Table 4.1: The total number (in thousands) of windows that were generated, not filtered out, and finally improved by either of the two subplanners (IBCS and PNGS) using different windowing heuristics over different block types (basic and extended). The rightmost pair of columns shows the rate of success, meaning the fraction of improved windows out of the generated windows by the different heuristics. This statistics is extracted from the results of the first experiment described in Section 3.3 (on page 42).

the other hand, the size of the set of windows that each generates and the fraction of improvable windows in this set varies between the strategies, and in that sense some are more useful than others. Table 4.1 shows the total number (in thousands) of windows that were generated, not filtered out, and finally improved by either of the two subplanners (IBCS and PNGS) using different windowing heuristics over different block types (basic and extended). In this experiment, windows are filtered out only if the window cost is matched by the admissible LM-Cut heuristic [Helmert and Domshlak, 2009]. The experiment uses setup 1 as described in Section 3.3 (on page 42). In the experiment, there were a total of 219 instances (out of 272 instances) for which we found some improvements. We measured each strategy's rate of success, meaning the fraction of windows generated by the strategy that were improved by either of the two subplanners used in the experiment, and used this to order them. The order is as follows:

1. Rule-based heuristic over extended blocks.
2. Cyclic thread heuristic over basic blocks.
3. Cyclic thread heuristic over extended blocks.
4. Causal followers heuristic over basic blocks.
5. Rule-based heuristic over basic blocks.
6. Causal followers heuristic over extended blocks.

The neighbourhood exploration procedure (Algorithm 2 on page 41) adds windows to the database incrementally, by calling the ExtractMoreWindows procedure shown in Algorithm 5. This procedure selects the next strategy to try, cycling through them in the order above, and asks this strategy to generate a specified number of windows, in a limited time. Each strategy keeps its own state (what part of the heuristic has been applied and up to what part of the plan), so that the next time it is queried it can resume generating new windows. When all windows that are possible under a given strategy have been generated, we say the strategy is *exhausted*. The windowing strategies discard (1) windows that are known to be optimal, either because their

Algorithm 5 Extract More Candidate windows.

```
   /* global array strategy[1..6] stores the state of each windowing strategy */
 1: procedure ExtractMoreWindows(π_bdp, windowDB, optSubprob)
 2:     W = ∅
 3:     t_limit = initial time limit T_increment
 4:     while t_elapsed < t_limit or |W| < nWindowsLimit do
 5:         i = NextWindowingStrategy()
 6:         if i = null then break   /* all windowing strategies are exhausted */
 7:         W = strategy[i].GetWindows(π_bdp, windowDB, optSubprob,
                                       nWindowsLimit − |W|, t_limit − t_elapsed)
 8:         if t_elapsed ≥ t_limit and W = ∅ then t_limit += T_increment
 9:     windowDB.Insert(W)
```

cost matches the lower bound given by the admissible LM-Cut heuristic [Helmert and Domshlak, 2009], or because they are in the stored set of optimally solved subproblems, and (2) windows that overlap with an already improved window. These windows are not eligible for optimisation (cf. Chapter 3), so generating them is redundant. If the selected strategy finishes without generating enough windows and time remains, the next not-yet-exhausted strategy in the order is queried, and so on, until either $|W|$nWindowsLimit or time is up. If no windows are generated, and some strategies are still not exhausted, the time limit is increased.

4.2 Rule-Based Windowing Heuristic

Rule-based windowing uses a fixed set of rules to formulate windows over π_{bdp}. Each rule selects a set of blocks to go into the replaced part (w). To ensure that the window is consistent with the block deordering (i.e., has a consistent linearisation, as stated in Definition 8 on page 39), any blocks that are constrained to be ordered between blocks in the window must also be included. We call these the *intermediate blocks*, which are formally defined as follows:

Definition 10. *Let* $\pi_{bdp} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ *be a block decomposed p.o. plan. The* intermediate blocks *of* $B \subseteq \mathcal{B}$ *are* $\text{IB}(B) = \{b \mid \exists\, b', b'' \in B : b' \preceq b \preceq b''\}$.

Let b be a block in π_{bdp}, and let $\text{Un}(b)$ be the set of blocks that are not ordered w.r.t. b, $\text{IP}(b)$ the immediate predecessors of b, and $\text{IS}(b)$ its immediate successors. The rules used by the windowing heuristic are:

1. $w' \leftarrow \{b\}$.
2. $w' \leftarrow \{b\} \cup \text{IP}(b)$.
3. $w' \leftarrow \{b\} \cup \text{IS}(b)$.
4. $w' \leftarrow \{b\} \cup \text{Un}(b)$.
5. $w' \leftarrow \{b\} \cup \text{Un}(b) \cup \text{IP}(b)$.

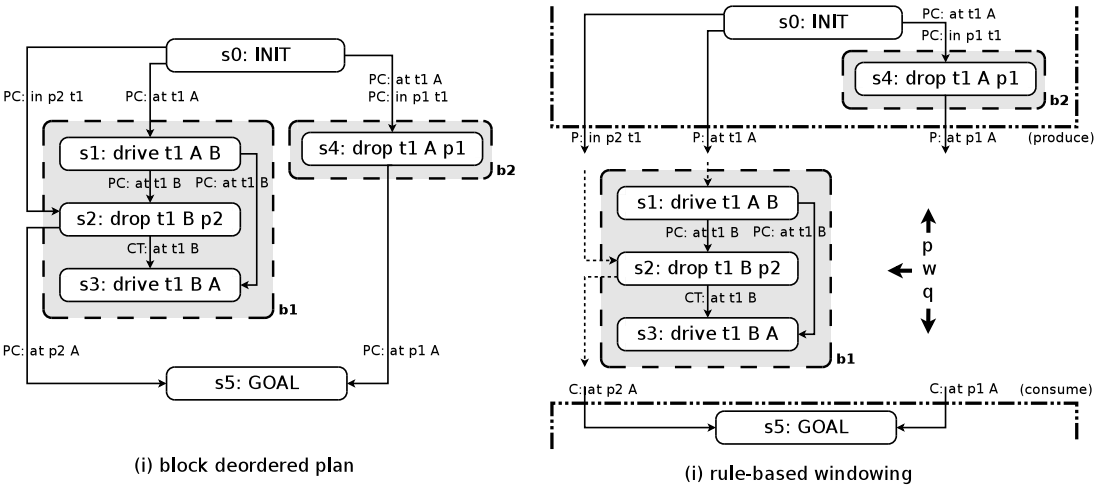


Figure 4.2: Window formation by applying the 1st rule of the rule-based windowing heuristic over the block b1, i.e., $w \leftarrow \{\text{b1}\}, p \leftarrow \text{Un}(\text{b1})$. Therefore, the unordered block b2 (with respect to b1) is placed in its predecessor list. Interestingly, this window can be optimised by simply removing s3 because this step has no causal link to its successors.

6. $w' \leftarrow \{b\} \cup \text{Un}(b) \cup \text{IS}(b)$.
7. $w' \leftarrow \{b\} \cup \text{Un}(b) \cup \text{IP}(b) \cup \text{IS}(b)$.
8. $w' \leftarrow \{b\} \cup \text{Un}(b) \cup \text{IP}(\{b\} \cup \text{Un}(b))$.
9. $w' \leftarrow \{b\} \cup \text{Un}(b) \cup \text{IS}(\{b\} \cup \text{Un}(b))$.
10. $w' \leftarrow \{b\} \cup \text{Un}(b) \cup \text{IP}(\{b\} \cup \text{Un}(b)) \cup \text{IS}(\{b\} \cup \text{Un}(b))$.

Recall that a window is a partitioning of blocks into $\langle p, w, q \rangle$. Given the blocks selected by one of the rules above, the window is formed by setting $w = w' \cup IB(w')$ and assigning to p any block that is ordered before or unordered with w, and to q any block ordered after w. Figure 4.2 shows an example of rule-based windowing, where the 1st rule is applied to block b1.

Applied to all blocks, the above rules can produce duplicates; of course, only unique windows are kept. The rules generally produce different sizes of windows. Larger windows are mostly produced by the higher indexed rules, while the lower indexed rules produce smaller windows (though there is no exact relation, since the number of steps in a block varies).

The rule-based windowing heuristic generates windows in two steps: First, it sorts all the blocks in descending order of their sizes (measured by the number of constituent steps). The ordering of two blocks b_x and b_y having equal size depends on their type. When b_x and b_y are of extended type, if b_x contains a step that is ordered before all other steps in $b_x \cup b_y$ in the (sequential) input plan, then b_x is ordered before b_y. In contrast, when b_x and b_y are of basic type, b_x is ordered after

b_y for the same condition. Since the ordering of basic blocks is opposite to that of extended blocks, alternating between the block types allows exploring different parts of the decomposed plan. In the second step, the heuristic applies all rules to each block in the block deordered plan π_{bdp} in turn. For each block (according to the sorted order), the rules are applied in the following order: $1, 10, 2, 9, 3, 8, 4, 7, 5, 6$, i.e., the lowest, the highest, the second lowest, the second highest, and so on. Thus, the rule-based windowing generates a smaller window followed by a larger window and so on.

Recall that ExtractMoreWindows repeatedly asks each windowing strategy to generate a limited number of windows. The ordering of blocks and rules described above helps to ensure that the heuristic generates a varied set of windows, including both small and large, covering different parts of the current plan, each time it is queried.

4.3 The Cyclic Thread Windowing Heuristic

To discover new windowing heuristics, we noted some key changes in the decomposed plan structure that frequently occur when a plan is improved. One significant observation is that if multiple steps of an input plan have the same add effects, then those steps together with the steps necessarily ordered in between them form a subplan that can often be improved. We call this *cyclic behavior*. We found in one experiment that cycles of this type are either removed from the plan or replaced with different cycles in more than 87% of the improvements across most domains. The definition of cyclic behavior is based on an individual atom. Intuitively, an atom has cyclic behavior if it has multiple producers (as defined below).

Definition 11. *Let $\pi_{bdp} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a block decomposed p.o. plan, and $P_m \subseteq \mathcal{S}$ be the set of producers of an atom m, i.e., $\forall_{s \in P_m} m \in add(s)$. m has* cyclic behavior *iff $|P_m| > 1$.*

Note that P_m contains the init step s_{I} iff $m \in add(s_{\text{I}})$. However, a candidate window is formulated from extended producers that never contain s_{I}. A step $s \notin \{s_{\text{I}}, s_{\text{G}}\}$ is an extended producer of an atom m iff s produces m, or s consumes m and there is no $s' \neq s_{\text{I}}$ that produces m and ordered before s in the block deordered plan. The formal definition of extended producer is as follows:

Definition 12. *Let $\pi_{bdp} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a block decomposed p.o. plan. A step $s \in \mathcal{S}$ is an* extended producer *of an atom m iff $s \notin \{s_I, s_G\}$ and:*
1. $m \in add(s)$ or
2. $m \in pre(s)$ and $\forall_{k \in \mathcal{S} \setminus s_I}$ if $m \in add(k)$ then $s \prec^+ k$.

In order to form candidate windows with respect to an atom m having cyclic behavior, we first extract all the blocks that contain at least one extended producer of an atom m. A cyclic thread (stated in Definition 14) is then formed by taking a linearisation of those blocks, consistent with the input plan.

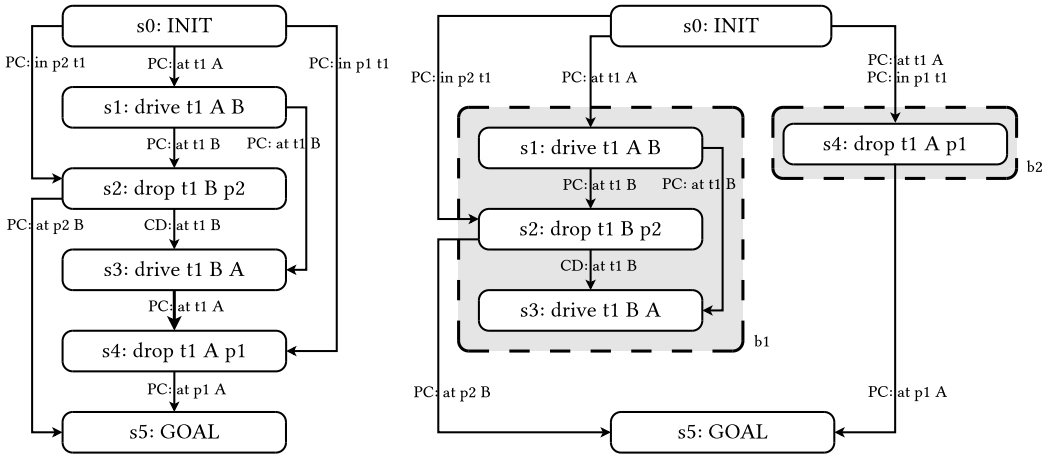


Figure 4.3: A sequential plan and a block deordering of this plan with two unordered blocks b1 and b2. The atom $m = (\text{at t1 A})$ has cyclic behavior since m is produced by $s0$ and $s3$, and $|\{s0, s3\}| > 1$ (as defined in Definition 11). The extended producers of m are $\text{EP}_m = \{s1, s3, s4\}$ according to Definition 12, which are captured by the blocks $b1$ and $b2$. The cyclic thread of m, according to Definition 14, is $T_m = \langle b1, b2 \rangle$, which yields three cyclic thread-based windows (according to Definition 15): $\langle \{b2\}, \{b1\}, \{\emptyset\} \rangle$, $\langle \{b1\}, \{b2\}, \{\emptyset\} \rangle$, and $\langle \{\emptyset\}, \{b1, b2\}, \{\emptyset\} \rangle$ in the format $\langle p, w, q \rangle$.

Definition 13. *Let $\pi_{bdp} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a block deordering of a sequential plan π_{seq}, and $b_x, b_y \in \mathcal{B}$ are two blocks such that $b_x \cap b_y = \emptyset$. Let $\langle b_x, b_y \rangle$ be a linearisation of $\{b_x, b_y\}$. $\langle b_x, b_y \rangle$ is consistent with π_{seq} if at least one step in b_x appears before a step in b_y in π_{seq}.*

The way we linearise the blocks so that it is consistent with the input plan is clarified by the following example. Assume $b_x : \{s_a, s_c\}$ and $b_y : \{s_b, s_d\}$ are two blocks that we have to linearise, and that the orderings of their constituent steps in the input plan is $s_a \prec_{\text{in}} s_b \prec_{\text{in}} s_c \prec_{\text{in}} s_d$. The linearisation starts with the block that contains the first element of $\prec_{\text{in}}$, i.e., b_x in this case (since it contains s_a); $\prec_{\text{in}}$ is then updated to $\prec_{\text{in}} \setminus b_x$, and the linearisation continues in the same fashion until $\prec_{\text{in}}$ is empty. The resulting linearisation of the example blocks will be $\langle b_x, b_y \rangle$. If multiple (nested) blocks contain the first element of $\prec_{\text{in}}$, the innermost one is picked. The formal definitions of thread and cyclic thread are as follows.

Definition 14. *Let $\pi_{bdp} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a block deordering of a sequential plan π_{seq}, $EP_m \subseteq \mathcal{S}$ be the set of extended producers of an atom m, and $B_m \subseteq \mathcal{B}$ be the set of blocks, where each element of B_m contains at least one element of EP_m. The* thread *of m, T_m, is the linearisation of blocks in B_m such that the linearisation is consistent with π_{seq}. The thread is called* cyclic *iff m has cyclic behavior.*

Finally, the candidate windows are formed by taking a consecutive subsequence of blocks (and their intermediate blocks) from a cyclic thread. Like in rule-based

windowing, blocks that are unordered with respect to the window are assigned to the set of blocks that will precede the window. Figure 4.3 shows an example of cyclic thread windowing.

Definition 15. *Let $T_m = b_1, ..., b_k$ be a cyclic thread of an atom m. The* cyclic thread-based windows *over the cyclic thread T_m are $W_{l,m} = \{B \cup \mathrm{IB}(B) \mid B = b_i, ..., b_{i+l}$ is a consecutive subsequence of $T_m\}$, while the unordered blocks are always placed in its predecessor set.*

Also like the rule-based windowing heuristic, the cyclic thread heuristic generates windows in an order that aims to ensure it returns a varied set of windows each time it is called. It first identifies all cyclic threads in the block deordered plan and then generates a stream of candidate windows from one cyclic thread after another (in an arbitrary order). As mentioned, each candidate window is formed by taking a consecutive subsequence of blocks (and the intermediate blocks as required to form a consistent window) from the cyclic thread. Given a thread of $|T_m|$ blocks, subsequences are generated according to the following order of sizes: $1, |T_m|, 2, |T_m| - 1, \ldots, |T_m|/2$. In other words, the subsequence lengths are ordered as the smallest, the biggest, the second smallest, the second biggest, and so on. For each size in this order, all windows are generated moving from the beginning to the end of the thread.

4.4 Causal Followers Windowing Heuristic

The third strategy that we have use to obtain a broader range of potentially improvable windows is similar to the cyclic thread heuristic in that it creates windows that are subsequences of a linearisation of blocks connected by a particular atom, but different in that these connections are via causal links.

Definition 16. *Let $\pi_{bdp} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a block decomposed p.o. plan, and $\prec_c$ be the set of causal links in $\prec$. The* causal followers *of an atom m for a producer $p \in \mathcal{S}$ are $\mathrm{CF}_{\langle m,p \rangle} = \{p, s_j, ..., s_k | \{\langle p, m, s_j \rangle, ..., \langle p, m, s_k \rangle\} \subseteq \prec_c\} \setminus \{s_I, s_G\}$. The* causal followers *of m (for all producers), CF_m, is the sequence $\langle \mathrm{CF}_{\langle m,p_1 \rangle}, ..., \mathrm{CF}_{\langle m,p_n \rangle} \rangle$, where $p_1, ..., p_n$ is a linearisation of all the producers of m.*

In other words, the causal followers of an atom m is a list of sets of steps. In each set of steps, one is the producer s and the others are the consumers s_j of m, and s has a causal link to every s_j for m, i.e., $\mathrm{PC}(m) \in \mathrm{Re}(s \prec s_j)$. For example, the atom (at t1 B) in the block deordered plan in Figure 4.3 appears in two causal links, both with the same producer $s1$: $\langle s1, (\text{at t1 B}), s2 \rangle$ and $\langle s1, (\text{at t1 B}), s3 \rangle$. Thus the causal followers are $\mathrm{CF}_{\langle \text{at t1 B} \rangle} = \langle \{s1, s2, s3\} \rangle$.

From the block deordered plan we extract the sequence of sets of blocks corresponding to the causal follower steps, according to the definition below. For example, the causal follower blocks of $\mathrm{CF}_{\langle \text{at t1 B} \rangle}$ in the plan in Figure 4.3(ii) is $\mathrm{CFB}_{\langle \text{at t1 B} \rangle} = \langle \{b1\} \rangle$, since all the steps in $\mathrm{CF}_{\langle \text{at t1 B} \rangle}$ are contained in block $b1$.

	Basic block	Ext. block	Rule-based	Cyclic thread	Causal followers
Exclusive	66.52	8.14	24.50	6.09	17.78
All	91.86	33.48	63.22	34.01	66.34

Table 4.2: Percentage of improvable windows found using the two block types and three windowing heuristics, out of the total number of improvable windows found using all blocks types and windowing heuristics. The first row gives the percentage of improvable windows found by one block type but not the other (or by one windowing heuristic but not the others), while the second row gives the percentage of all improvable windows found by one block type (or windowing heuristic). The results are from the first experiment, described in Section 3.3 on page 42.

Definition 17. *Let $\pi_{bdp} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a block decomposed p.o. plan, and $\mathrm{CF}_{\langle m,p \rangle}$ be the causal followers of an atom m with respect to a producer $p \in \mathcal{S}$. The* causal follower blocks *with respect to a producer $p \in \mathcal{S}$ of an atom m, $\mathrm{CFB}_{\langle m,p \rangle}$, is the set of blocks, where each block contains at least one element of $\mathrm{CF}_{\langle m,p \rangle}$. The* causal follower blocks *of m (for all producers), CFB_m, is the sequence $\langle \mathrm{CFB}_{\langle m,p_1 \rangle}, ..., \mathrm{CFB}_{\langle m,p_n \rangle} \rangle$, where $p_1, ..., p_n$ is a linearisation of all the producers of m in π_{bdp}.*

Candidate windows are formed by taking consecutive subsequences of the sequence of causal follower blocks (with intermediate blocks, as necessary). The formal definition is given below. Like in the other windowing heuristics, blocks that are unordered with respect to the window are assigned to the set of blocks that will precede the window. As an example, $\langle \{b2\}, \{b1\}, \{\emptyset\} \rangle$, in the format $\langle p, w, q \rangle$, is the only window that can be formed from the causal follower blocks $\mathrm{CFB}_{\langle \text{at t1 B} \rangle} = \langle \{b_1\} \rangle$ in the plan in Figure 4.3(ii).

Definition 18. *Let $\pi_{bdp} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a block decomposed p.o. plan, and $\mathrm{CFB}_m = \langle \mathrm{CFB}_{\langle m,p_1 \rangle}, ..., \mathrm{CFB}_{\langle m,p_n \rangle} \rangle$ be the causal follower blocks of m. The* causal followers-based windows *over CFB_m are $W_{l,m} = \{B \cup IB(B) \mid B = \mathrm{CFB}_{\langle m,p_i \rangle} \cup ... \cup \mathrm{CFB}_{\langle m,p_{i+l} \rangle}$ is a consecutive subsequence of CFB_m of length $l\}$, while the unordered blocks are always placed to its predecessor list.*

The order in which windows are generated by the causal followers heuristic is based on the same principle as in the cyclic thread heuristic. It generates a stream of candidate windows from the causal follower blocks CFB_m associated with each atom m in turn, where the atoms are processed in an arbitrary order. These windows are consecutive subsequences of sets of blocks from CFB_m, of lengths chosen according to the pattern $1, l, 2, l-1, ..., (l/2)$, where l is the length of CFB_m.

4.5 The Impact of Windowing Heuristics

No one single windowing heuristic or block type, nor any combination of those, is guaranteed to find all the improvable windows. The first row of Table 4.2 shows the percentage of improvable windows found using one block type but not the other

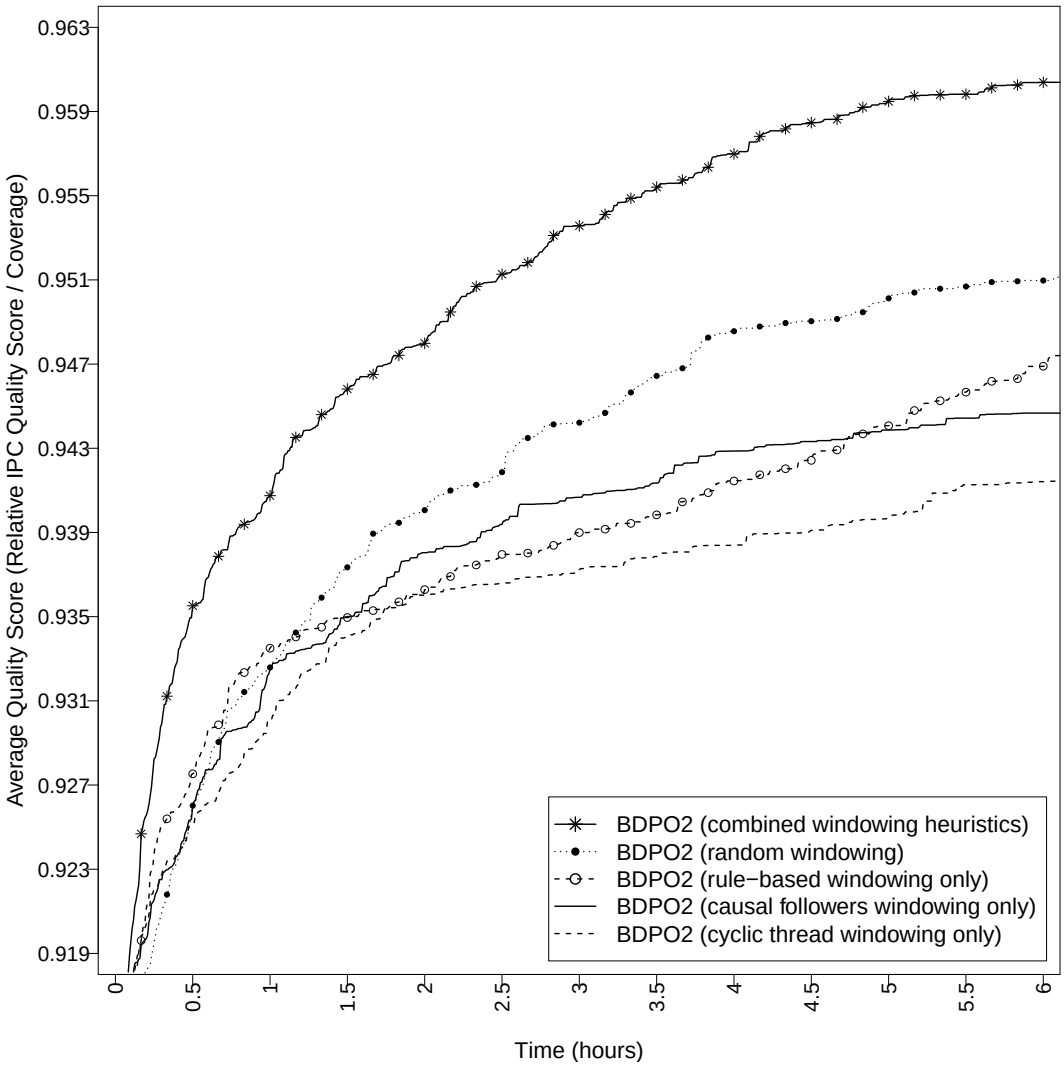


Figure 4.4: Average IPC quality score as a function of time for separate runs of BDPO2 using each of the three windowing heuristics alone, all three of them combined via three windowing heuristics, and random windowing (without using any heuristic). Each run is done using the experimental setup 3 as described in Section 3.3 (on page 42). Note that the y-axis is truncated (started at 0.907) and the x-axis shows the runtime of BDPO2 (which started at 2 hour in the overall experiment shown in Figure 3.1).

(or by one windowing heuristic but not the others), out of the total number of improvable windows found using all blocks types and windowing heuristics. It clearly shows that every windowing heuristic and block type contributes some improvable windows that are not found by other strategies. For example, 24.5% of improvable windows are found only by the rule-based windowing heuristic (using both basic and extended blocks), i.e., those windows are not found by the other two windowing heuristics. On the other hand, 36.78% of improvable windows are not found by the rule-based heuristic, since 63.22% of improvable windows are found in total by this heuristic (as shown in the second row). Each of the windowing heuristics has its strengths and limitations. The rule-based heuristic, for example, can only generate windows up to a fixed length of extended blocks. The cyclic thread heuristic, on the other hand, is not limited to a fixed window length, but the windows it can form are restricted to the threads of a single atom, i.e., it may not find an improvable window which can otherwise be found by merging cyclic threads of two different atoms. The causal followers heuristic likewise considers only blocks connected by a single atom, but usually generates many more candidate windows, including some not found by the others.

We have seen in an experiment that a combined technique that uses the individual heuristics in portfolio (as described at the beginning of this chapter) contributes more to our plan quality improvements than any single heuristic alone. The portfolio of strategies also shows better results than generating windows randomly. In the random windowing strategy, a window is formulated by taking a random subsequence of blocks from a random linearisation of the current block deordered plan. While generating windows using the random windowing strategy, we maintain the distribution of the quantity of windows of different sizes similar to that of the combined windowing in the original BDPO2. Figure 4.4 shows the average IPC quality score achieved by the combined technique compared to that achieved using each windowing heuristic alone. In this experiment we ran BDPO2 five times: three separate runs using individual windowing heuristics, one run using a portfolio of these heuristics which is used in original BDPO2, and the last run using random windowing strategy. In each run, BDPO2 used as input the best plans found by PNGS applied to the base plans, same as experimental setup 3 described in Section 3.3 on page 42. In this experiment, where the input plans were of high quality, the total quality improvements (measured by the average IPC quality score) found by the random windowing strategy is 17.1% less than that found by the portfolio of windowing strategies.

4.6 Possible Extensions to the Windowing Strategies

Since a window is formulated by partitioning plan steps into three disjoint sets of blocks, the total number of possible windows is exponential. Therefore, it is challenging to extract windows that are more likely to be improved from this large set of all possible windows. Each of the windowing strategies has its strength in finding many of these improvable windows, but also has limitations in some domains.

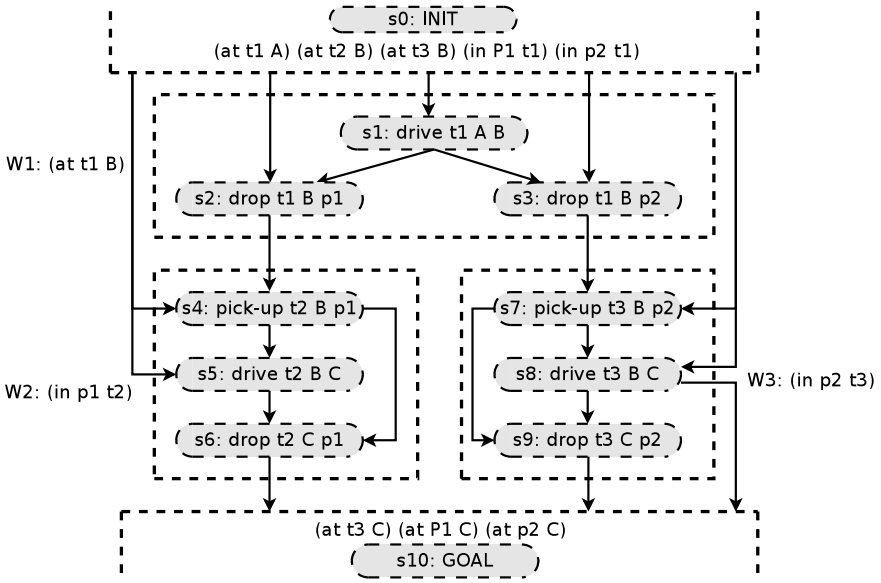


Figure 4.5: An example of forming a composite window. Three separate candidate windows: W1, W2, and W3, are found by the causal followers windowing heuristic for the atoms (at t1 B), (in p1 t2), and (in p2 t3) respectively. None of them are separately improvable. However, the composite window, formulated by merging W1 and W2, is improvable by substituting the delivery of package p1 (from location B to C) provided by truck t2 with truck t1. This is because the atom (at t2 C) is not required by any of its successors (i.e., the goal in this example).

Therefore, all together they do not guarantee to find all improvable windows from this extremely large set of all possible windows. Hence, there is always a scope for developing new windowing heuristics or extending the existing ones; one such extension is discussed in this section.

The combined windowing technique that alternates between the individual windowing heuristics is very useful in the overall plan improvement process (as Figure 4.4 shows), but, as mentioned, it may fail to find some improvable windows. For example, if an improvable window has a long length without any cyclic thread or causal followers with respect to a single atom, then it may not be captured by any of the aforementioned windowing heuristics. Forming composite windows by combining one or more candidate windows (and their intermediate blocks) found by the individual windowing heuristics may allow generating improvable windows that are not found by any heuristic. An example of a composite window is shown in Figure 4.5, where the three candidate windows, W1, W2, and W3, found by the causal followers windowing heuristic, are not improvable separately, but the union of W1 and W2 is improvable. This type of composite windows could be formed in later stages of the plan improvement process, after all the individual windowing heuristics have been exhausted. However, note that forming composite windows from a large set of candidate windows is a combinatorial problem and thus optimising all of them will

take a long time.

4.7 Window Ranking

Windowing strategies often extract a large number of candidate windows from a block decomposed plan. In order to speed up the plan improvement process, it is important to find windows that are more likely to be improved so that we can optimise those first. Providing an ordering of the candidate windows that prioritises the more promising windows is the role of window ranking.

Determining whether a window is likely to be improved is difficult. One of the reasons is that the properties of improvable windows vary from one to another, and a lot from domain to domain. For example, as mentioned at the beginning of this chapter, larger windows are more likely to be improved in the Pegsol, Openstacks, and Parcprinter domains, while smaller windows are better for the Elevators, Transport, Scanalyzer, and Woodworking domains. The Sokoban domain, on the other hand, is good for medium-sized windows. Moreover, an improvable window may not be improved within a given time bound. We have noted that in some domains, such as, e.g., Pegsol or Scanalyzer, subplanners require, on average, more time to find a lower-cost plan. Therefore, deciding whether a window is likely to be improved within a given time bound is even more difficult.

We have developed a set of window ranking policies by examining structural properties of the candidate windows generated by the windowing strategies. In order to examine the properties, we used the data from our first experiment (cf. Section 3.3 on page 42) in which we systematically tried two subplanners (IBCS and PNGS) on each generated window with a 30 second time limit, excluding only windows whose cost is already shown to be optimal by the admissible LM-Cut heuristic [Helmert and Domshlak, 2009]. Investigating the properties of improved and unimproved windows, we identify four structural metrics (as ranking policies) that work relatively well across domains:

(1) The total number of causal links whose producers reside in a window and whose consumers are outside the window, divided by the length of the window – the lower the value the higher the rank. We call this property "outgoing causal links per length".

(2) The total number of causal links whose consumers reside in a window and whose producers are outside the window, divided by the length of the window – the lower the value the higher the rank. We call this property "incoming causal links per length".

(3) The gap between the cost of a window and the lower bound on the cost of any plan for the corresponding subproblem given by the admissible heuristic – the higher the value the higher the rank.

(4) The number of pairwise ordering (of steps) disagreements between a window $\langle p, w, q \rangle$ and the sequential input plan – the lower the value the higher the rank. To calculate this we first take the linearisation of $\langle p, w, q \rangle$ that is used to generate

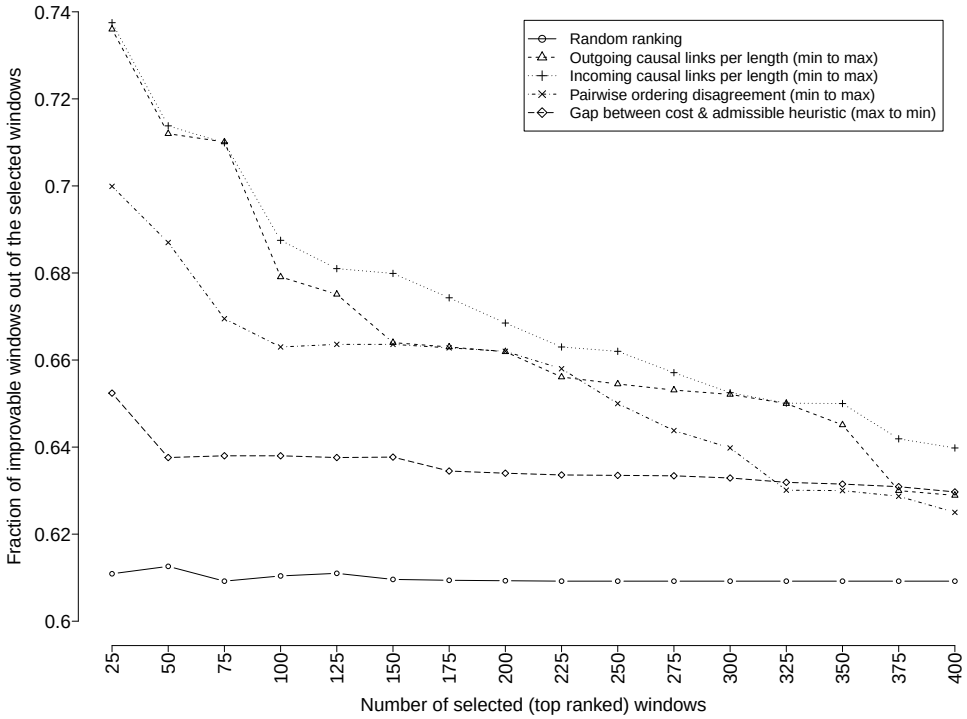


Figure 4.6: Fraction of improvable windows, across all domains, out of the selected top windows from the ranked orders generated by each of the ranking policies (see text).

the corresponding subproblem. Then, for every pair of plan steps, if the ordering between them in the linearisation is not the same as in the input plan we call this a pairwise ordering disagreement. The lower the total number of such disagreements is for a window, the higher its rank. In other words, if the ordering of steps in a window is very different from the input plan then it is less likely to be improved.

We can infer from the first two ranking policies that the more disconnected a window is from other blocks in the decomposed plan the more likely it is to be improved. Figure 4.6 compares these ranking policies with the performance of a random ranking. On average across all domains, all four ranking policies are good at picking out improvable windows. For example, if we pick the top 25 windows from the order generated by the "incoming causal links per length" policy, nearly 74% of those windows are improvable (by at least one subplanner), while the top 25 windows from the random order contains only 61% improvable windows. The random ranking in Figure 4.6 is the best result out of three separate random rankings for each of the values on the x-axis. As expected, it exhibits more or less the same ratio of improvable windows over all ranges (from 25 to 400). Nearly 61% of the selected windows, across all domains, are improvable. However, the performance

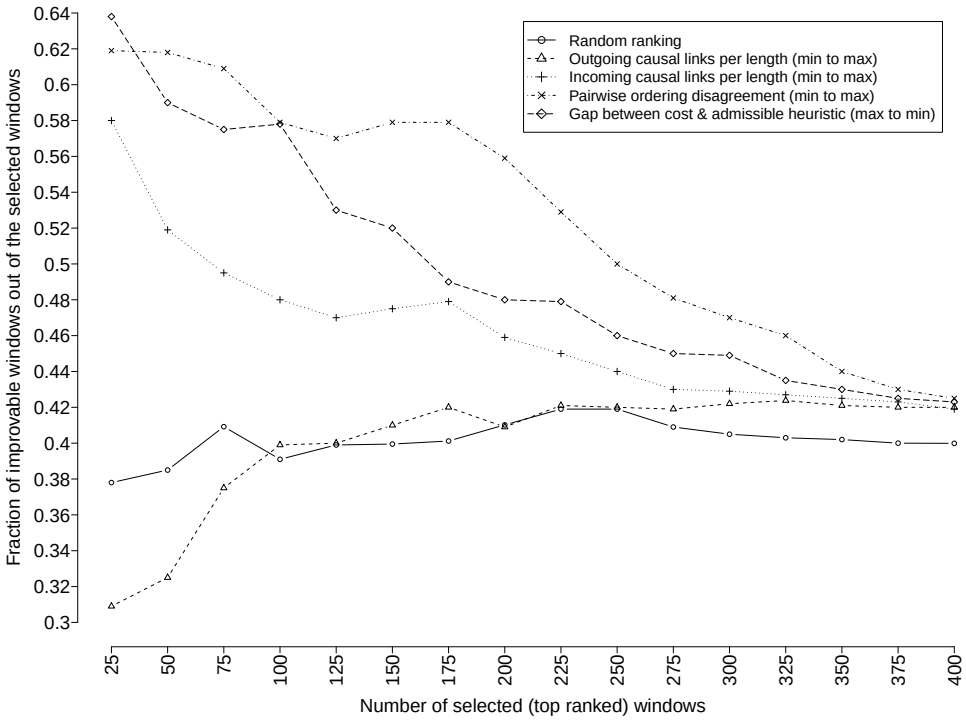


Figure 4.7: Fraction of improvable windows in the Parking domain, out of the selected top windows from the ranked orders generated by each of the ranking policies (see text).

of individual ranking policies varies by domain, and for each policy we find some domain in which it is not good. For example, Figure 4.7 shows ranking results for instances of the Parking domain only: Here, the "outgoing causal links per length" policy does not work well. Considering the top 90 windows in the ranked order, it is even worse than random. However, the other ranking policies are quite beneficial in this domain.

Among the four ranking policies, the first three work relatively better than the fourth in most of the domains, and therefore, we use the first three ranking policies in BDPO2. Since the strength of the individual ranking policies varies from domain to domain, we use them in a portfolio (as explained in Chapter 3). For each subplanner, BDPO2 uses one current ranking policy to select the next window for the chosen subplanner (from those eligible for optimisation by that subplanner). If no improvement is found by that subplanner in a certain number of attempts (13, in our current configuration), the system switches a different ranking policy, to produce a different ordering of candidate windows for being optimised by that subplanner.

We also tried different methods of combining the ordered lists generated by different ranking policies, in order to achieve a ranking with more stable performance across domains. The problem of combining rankings, often called rank aggregation, has been studied in many disciplines, such as social choice theory, sports and competitions, machine learning, information retrieval, database middleware, and so on. Rank aggregation techniques range from quite simple (based on rank average or number of pairwise wins) to complex procedures that in themselves require solving an optimisation problem. We tried five simple but popular rank aggregation techniques, namely Borda's [1781] method, Kemeny's [1978] optimal ordering, Copeland's [1951] majority graph, MC4 [Dwork et al., 2001], and multivariate Spearman's rho [Bedo and Ong, 2014]. The result of those experiments, however, is that rank aggregation does not produce better, or more stable, window rankings, especially in cases where one individual policy is relatively bad. Hence we prefer using the individual ranking policies in a cyclic order than any of the rank aggregation techniques in BDPO2.

The use of window ranking has a beneficial effect on the overall plan improvement process, as shown in Figure 4.8. We achieve higher quality scores, and in particular, achieve them faster, when using window ranking compared to random ranking. This experiment used the same setup as experiment 3 (described in Section 3.3 on page 42). We ran BDPO2 once with the ranking policies, and once with random ranking. The quality score of a plan is calculated as before (see page 44).

4.8 Summary

This chapter describes three strategies for formulating windows over a block deordered plan: (1) rule-based windowing that uses a fixed set of rules, (2) cyclic thread windowing that is based on the cyclic behavior of atoms, and (3) causal followers windowing that is based on the atom-wise causal dependency of blocks. We

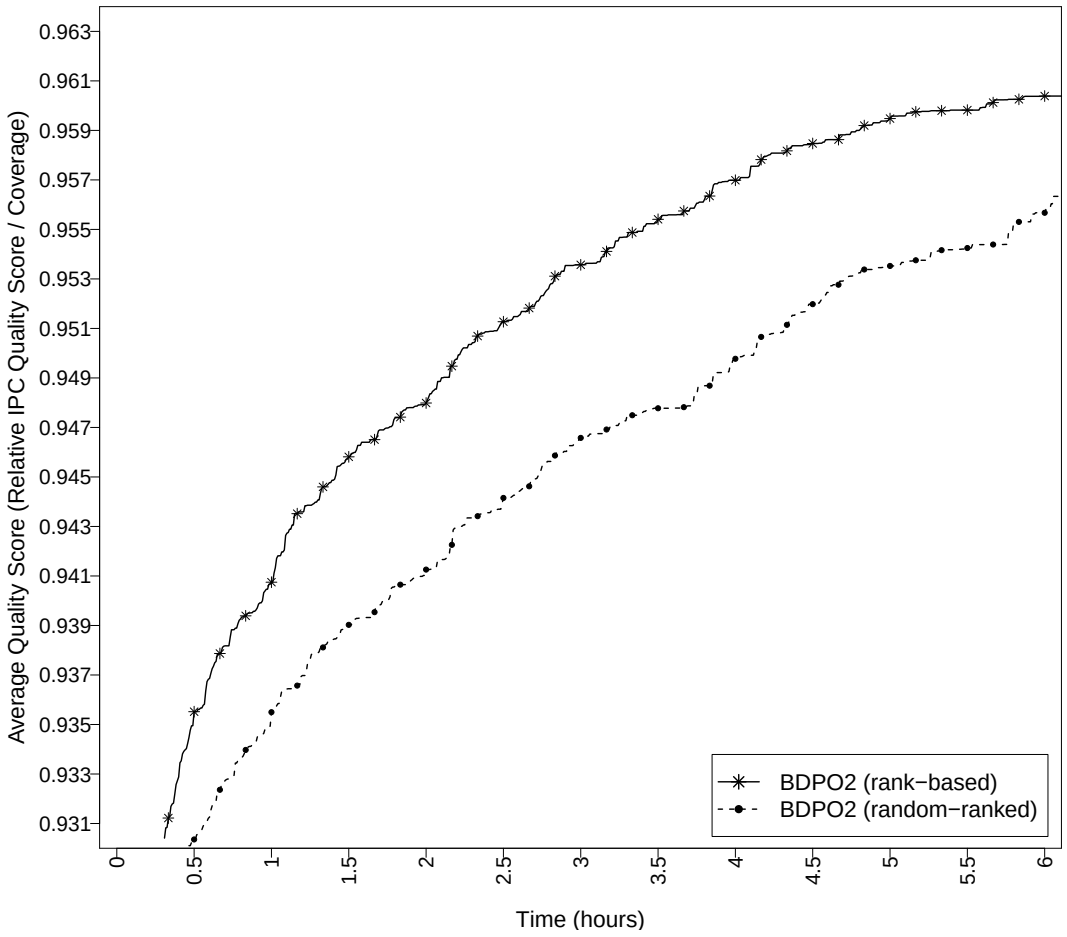


Figure 4.8: Average IPC plan quality score of BDPO2 as a function of time in two separate runs: with and without considering the ranking policies. In the second case, the ranks of the candidate windows are randomised. This experiment is run with the setup 3 as described in Section 3.3. Note that the y-axis is truncated (started at 0.907) and the x-axis shows the runtime of BDPO2 (which started at 2 hour in the overall experiment shown in Figure 3.1).

alternate between the windowing strategies, since no one single strategy, or any combination of those, is guaranteed to find all improvable windows. However, these strategies all together may fail to find some improvable windows. Combining some candidate windows, found by the individual windowing strategies, in a systematic way may allow generating as yet undiscovered improvable windows.

The windowing strategies often extract a large number of candidate windows; ordering these windows based on how likely they are to be improved is important to speed up the plan improvement process. This chapter also describes a set of window ranking policies to prioritize the more promising windows by examining structural properties of the candidate windows.

While this chapter describes the windowing component of BDPO2 in detail, the next chapter explains the on-line adaptation component of BDPO2 (briefly introduced in the previous chapter).

Chapter 5

On-Line Adaptation

We have a range of different policies for selecting subplans for optimisation, and subplanners for doing the optimisation in our plan improvement system. Picking the right one from many alternatives is very important and challenging to provide a continuous flow of plan quality improvement. We cast the problem of picking the most useful subplanner for the current subproblem from a group of alternative subplanners as a multi-armed bandit (MAB) problem. MAB, in brief, is a popular machine learning framework used for balancing between exploration and exploitation. We use this formulation to exploit the potential subplanners as much as possible. This chapter describes the MAB problem, how we use this for subplanner selection, and the impact of this on our overall plan improvement system. The process of adapting the window ranking policies (discussed in the previous chapter) on-line with the current problem is also discussed in this chapter.

This chapter is organized as follows: Section 5.2 presents the background and formulation of multi-armed bandit problem, while Section 5.3 describes the bandit policy used for selecting subplanners in BDPO2, and other alternative policies from the literature. The impact of the bandit policy on our overall plan improvement system is discussed in Section 5.4. Finally Section 5.5 describes the process of adapting the window ranking policies to the current problem in BDPO2.

5.1 Introduction

The approach to plan quality optimisation by repeatedly solving local subproblems gives us the opportunity for adapting the process on-line to the current problem. We have noted that different subplanners, windowing strategies, and ranking policies work better in different domains. For example, Figure 5.1 shows the fraction of local improvements found by each of the three subplanners in different domains. As can be seen, the IBCS subplanner is more productive, compared to PNGS and LAMA, in the APPN, Barman, Maintenance, Parking, Sokoban, and Woodworking domains. PNGS, on the other hand, is better in the Scanalyzer and Visitall domains, and LAMA in the Elevators and Openstacks domains. Therefore, if we can learn over the course of the local search the relative success rate of different subplanners on the current problem, the system will perform better. In a similar fashion, the choice of

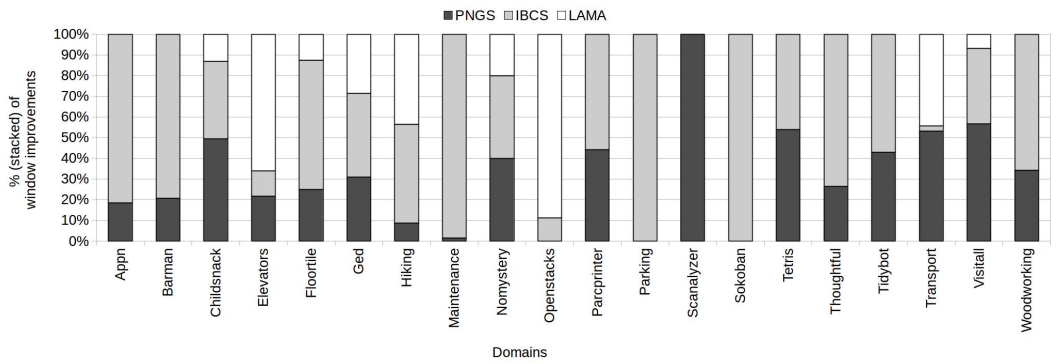


Figure 5.1: The percentage (stacked) of window improvements found by each of the subplanners (PNGS, IBCS, and LAMA), out of total number of improved windows found by all the subplanners. In this experiment, BDPO2 was run three times, each time with one subplanner. (If more than one subplanner improve on the same window, these are counted as separate improvements.) The setup was the same as for experiment 3 (described in Section 3.3 on page 42).

window generation strategy and ranking policy may also be adapted to the current problem, so that the system is more likely to select subplans for optimisation that are improvable.

5.2 The Multi-armed Bandit Problem

The *multi-armed bandit* (MAB) problem is a popular machine learning formulation for capturing the fundamental trade-off between exploration and exploitation. In a MAB problem, an algorithm is presented with a sequence of trials. In each round, the algorithm chooses one alternative from a set of alternatives (often called "arms") based on the past history, and receives a reward for this choice. The goal is to maximise the total reward over time.

The name "multi-armed bandit problem" is derived from a gambling metaphor [Awerbuch and Kleinberg, 2008]: a gambling machine, also known as one-armed bandit, is activated by means of a lever pulls (in old machines), button push, or touchscreen (in newer machines). The machine is referred to as a game of chance, where the objective is to draw the highest possible rewards from the machine. In a multi-armed bandit problem, a gambler has access to multiple different gambling machines and faces the problem of deciding which machines to play, how many times to play each machine, and in which order to play them. When played, each machine provides a random reward from a stationary distribution specific to that machine. The objective of the gambler is to maximize the sum of rewards earned through a sequence of lever pulls. In the literature of multi-armed bandit problem, the lever is often referred to as an "arm"; to be consistent, we also refer the same (i.e.,

arm) throughout this thesis.

5.2.1 Applications

A bandit learning algorithm balances exploiting the arms with the highest observed average reward with exploring poorly understood arms to discover if they can yield better reward. As a result, MAB problems can model a wide variety of real-world decision-making scenarios including those associated with searching in an uncertain environment. MAB has found numerous applications in diverse fields (e.g., control, economics, statistics, and learning theory) after the influential paper by Robbins [1952]. A classical example of a MAB problem is the evaluation of clinical trials with medical patients described by Thompson [1933]. The decision-maker is a doctor and the options are different treatments with unknown effectiveness for a given disease. Given patients that arrive and get treated sequentially, the objective for the doctor is to maximize the number of cured patients.

5.2.2 Basic Formulation

According to Auer et al. [2002], a k-armed bandit problem, in its most basic formulation, is defined by random variables $X_{i,n}$ for $1 \leq i \leq K$ and $n \geq 1$, where each i is the index of an arm and n is the index of the plays. Successive plays of machine i yield rewards $X_{i,1}, X_{i,2}, ...$ which are independent and identically distributed (i.e., each of $X_{i,1}, X_{i,2}, ...$ has the same probability distribution as the others and all are mutually independent) according to an unknown law with unknown expectation $\mu_i = \mathbb{E}[X_i]$. Independence also holds for rewards across machines; i.e., $X_{i,s}$ and $X_{j,t}$ are independent (and usually not identically distributed) for each $1 \leq i < j \leq k$ and each $s, t \geq 1$.

A *bandit policy* is an algorithm that chooses the next machine to play based on the sequence of past plays and obtained rewards $X_{i,1}, X_{i,2}, ..., X_{i,n}$. Let n_i be the number of times machine i has been played by the policy during the first t plays. Then the regret of the policy after t plays is defined by $R_t = \mu^\star t - \mu_j \sum_{j=1}^{k} \mathbb{E}[n_j]$, where μ_i is the reward expectation for arm i, $\mu^\star = \max_{1 \leq i \leq k} \mu_i$ is the reward expectation for the best arm, and $\mathbb{E}[.]$ denotes expectation. Thus, the regret is the expected loss due to the fact that the policy does not always play the best arm. The goal of the policy is to find an arm selection strategy such as to minimize R_t.

We fit the multi-armed bandit problem in the on-line adaptation of our plan optimisation process by casting the arms as subplanners with the goal of finding the best subplanner so that we can exploit that subplanner as much as possible in order to maximise the overall plan quality optimisation.

5.3 Multi-armed Bandit Policies

Many policies have been proposed for MAB problems under different assumptions, for example, with independent [Auer et al., 2002] or dependent arms [Pandey et al.,

2007], exponentially or infinitely many arms [Wang et al., 2008], finite or infinite time horizon [Jones and Gittins, 1974], with or without contextual information [Slivkins, 2014], and so on. The idea of on-line adaptation in our plan improvement system aligns best with the "finite set of independent arms" assumption of the MAB problem. A number of policies have been proposed under this assumption; some of the widely used policies are ϵ-greedy [Johnson et al., 2000], Bayesian policy (e.g., randomized probability matching [Scott, 2010], Gittins index [Gittins et al., 2011], Thompson sampling [Thompson, 1933], etc.), Softmax exploration [Johnson et al., 2000], and optimistic exploration (e.g., UCB1 [Auer et al., 2002]).

This section describes the multi-armed bandit policy (the optimistic exploration policy to be specific) used for planner selection in BDPO2, followed by a brief discussion on the other alternative policies from the literature mentioned above.

5.3.1 A MAB Policy for Subplanner Selection in BDPO2

As mentioned, we cast the problem of selecting the subplanner (invoked on each subproblem) to use as a multi-armed bandit problem. The goal of this problem is to exploit the best subplanner as much as possible in order to maximise the overall plan quality optimisation. Our planner selection strategy is based on the optimistic exploration policy, where we select an arm in the most favorable environments that has a high probability of being the best, given what has been observed so far. This strategy is often called "Optimism in the face of uncertainty". At time t, for each arm k, this strategy uses past observations and some probabilistic argument to define high-probability confidence intervals containing the expected reward μ_k. The most favorable environment for arm k is thus the upper confidence bound (UCB) on μ_k. One simple implementation of this strategy is to play the arm having the largest UCB.

A number of algorithms have been developed for optimistic exploration of bandit arms, such as UCB1, UCB2, and UCB1-NORMAL by Auer et al. [2002], UCB-V by Audibert et al. [2009], and KL-UCB by Garivier and Cappé [2011]. We use the UCB1 algorithm for planner selection. UCB1 is an upper confidence bound algorithm derived from the index-based policy of Agrawal [1995]. The intuition behind UCB1 is that despite our lack of knowledge of what alternatives are best we construct an optimistic guess with respect to how good the expected reward of each alternative is, and pick the one with the highest estimate. If our guess is wrong, then our optimistic estimate quickly decreases and we are compelled to switch to an alternative. But if we choose well, we are able to exploit that alternative and incur little regret. In this way we balance exploration and exploitation, and believe that an alternative is as good as possible given the available evidence.

In describing the basic formulation of UCB1, at each time t, the algorithm selects the arm with highest upper confidence bound $B_{k,t} = \widehat{\mu}_{k,t} + \sqrt{\frac{2 \ln t}{n_k}}$, the sum of an exploitation term and an exploration term respectively. $\widehat{\mu}_{k,t}$ is the empirical mean of the rewards received from arm k up to time t, and n_k is the number of times arm k has

been pulled up to time t. The second term of $B_{k,t}$, $\sqrt{\frac{2\ln t}{n_k}}$, is a confidence interval for the average reward (within which the true expected reward falls with almost certain probability), hence $B_{k,t}$ is a upper confidence bound. UCB1 can achieve logarithmic regret uniformly over the number of plays and without any preliminary knowledge about the reward distributions [Auer et al., 2002].

The subplanner selection process in BDPO2, according to the UCB1 algorithm, consists of two phases: First we select each subplanner once to initialise expectations. After each optimisation attempt, we give a reward of 1 to a subplanner for finding an improvement, and a reward of 0 otherwise. We could use some other method for assigning the rewards rather than simply 0 and 1, for example, associating the amount of improvement (or time taken for an improvement) of a window found by a subplanner. However, we have seen in our experiments that assigning varying rewards to subplanners after the optimisation attempts makes the bandit learning system more complicated, and does not help in achieving better overall result. In the second phase, we select each time a subplanner p that maximises the upper confidence bound of p, $U_p = \widehat{\mu}_p + \sqrt{\frac{2\ln t}{n_p}}$, as explained above. Here, n_p is the number of times p has been tried so far, and t is the total number of optimisation attempts (by all subplanners) done so far. We can see that U_p grows with t but shrinks with n_p. This ensures each alternative is tried infinitely often but still balances exploration and exploitation. In other words, the more we try p, the smaller the size of the confidence interval and the closer U_p gets to its mean value $\widehat{\mu}_p$. But p cannot be tried once U_p becomes smaller than $\widehat{\mu}_{p^*}$, where p^* is the planner with the best reward, i.e., $\widehat{\mu}_{p^*} = max_k \widehat{\mu}_k$.

5.3.2 Other MAB Policies

We use the UCB1 algorithm for subplanner selection because it is simple, widely used , and uses an optimistic exploration strategy. However, as mentioned, there are a number of other multi-armed bandit algorithms under the finite set of independent arms assumption, any of them might be considered as an alternative policy for subplanner selection (although we have not tried). Below are given a brief description of those algorithms.

(a) ϵ-greedy policy [Johnson et al., 2000]: With ϵ-greedy, on tth play, the agent chooses a random arm with probability $0 \leq \epsilon \leq 1$, or the arm with highest empirical mean with probability $1 - \epsilon$. Although ϵ-greedy policy is an effective and popular means of balancing exploration and exploitation, one drawback is that when it explores it chooses equally among all arms. This means that it is as likely to choose the worst-appearing arm as it is to choose the next-to-best arm. In problems where the worst arms are very bad, this policy may be unsatisfactory, since no distinction of sub-optimal arms is considered.

(b) Bayesian policy: Bayesian policy assigns priors to the arm distributions and select arm according to the posterior distributions (e.g., Gittins index [Gittins

et al., 2011], Thompson sampling [Thompson, 1933], etc.). This policy eliminates the drawback (choosing equally among all arms during exploration) of ϵ-greedy policy because of varying the arm probabilities.

(c) Softmax exploration [Johnson et al., 2000]: In softmax arm selection rules, the greedy arm (i.e., the arm with the highest observed mean reward) is given the highest selection probability, but all the others are ranked and weighted according to their value estimates. The most common softmax method uses a Gibbs or Boltzmann distribution. It chooses arm i on the tth play with probability $\frac{\exp(\mu_{it}/\tau)}{\sum_{j=1}^{k}\exp(\mu_{jt}/\tau)}$, where τ is a positive parameter called the temperature, and μ_{it} is the estimated value of arm i at the th play. High temperatures cause all the arms to be (nearly) equiprobable. Low temperatures cause a greater difference in selection probability for arms that differ in their value estimates. In the limit as $\tau \to 0$, softmax arm selection becomes the same as greedy arm selection.

(d) Variants of UCB1 for optimistic exploration: As described in the previous subsection, in optimistic exploration strategies, an agent believes that it can obtain extra rewards by reaching the unexplored parts of the state space. Hence it selects an uncertain arm that has a high probability of being the best based on the observation so far. UCB2 [Auer et al., 2002] is a slight variant of UCB1. In UCB2, the plays are divided in epochs. In each new epoch an arm i is picked based on some complex factor and then played $\tau(r_i+1)\tau(r_i)$ times, where τ is an exponential function and r_i is the number of epochs played by that arm so far. UCB1-NORMAL [Auer et al., 2002] is a special case of UCB1, which assumes the distribution is normal and uses the sample variance for computing the index as an estimate of the unknown variance. In contrast, the reward distribution in UCB1 is unknown, and uses Chernoff-Hoeffding bounds to compute the index. Since better confidence bounds imply smaller regret, UCB-V by Audibert et al. [2009] and KL-UCB by Garivier and Cappé [2011] use different empirical variance estimates in order to achieve better confidence bound.

The impact of using the UCB1 algorithm in our plan optimisation system for selecting subplanners on-line is described in the following section.

5.4 Impact of Bandit Learning in Planner Selection

The response of the bandit policy for subplanner selection is shown in Figure 5.2. The figure shows the fraction of the total number of optimisation attempts that one subplanner, IBCS, was selected, and the fraction of the total number of window improvements found by that subplanner. Since BDPO2 in this experiment uses only two subplanners, IBCS and PNGS, the corresponding fraction for PNGS is $1-y$. As an example, in the third problem (from the left) in the APPN domain, 100% of window improvements are found by IBCS, and the bandit policy selects this subplanner for 84% of the total number optimisation attempts. PNGS is chosen for the other

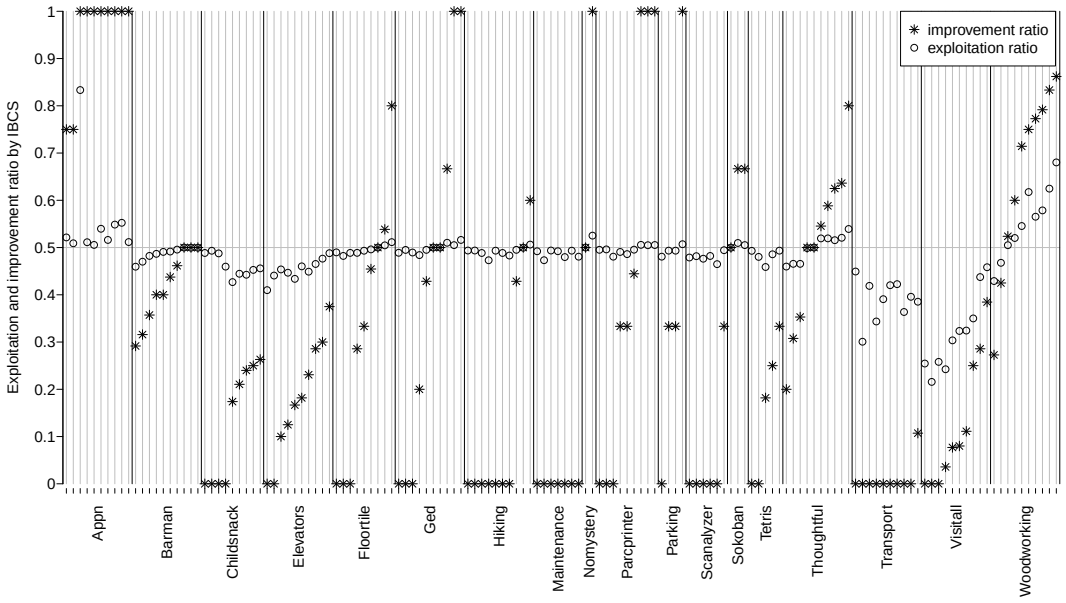


Figure 5.2: The exploitation and improvement ratio by IBCS in optimising a given plan of every problem across the domains. (Each vertical line represents a single problem.) Exploitation ratio: Fraction of the total number of optimisation attempts for which the IBCS subplanner was chosen, out of the total number of attempts by both subplanners. Improvement ratio: Fraction of the total number of improved windows found by IBCS, out of total number of improved windows found by both subplanners. Since IBCS and PNGS are the only two subplanners used in this experiment, the corresponding ratios for PNGS are the opposite (i.e., $1 - y$). The experiment was run with the same setup as experiment 2, described in Section 3.3 on page 42.

16%, but finds no improvement. We can see that the bandit policy selects the more promising subplanner more often across the problems. However, the bandit policy is somewhat conservative, because it ensures that we do not rule out any subplanners that fare poorly early on. Moreover, as the current plan is improved it becomes harder and harder to find further improvements (within the given time bound), so the average reward for both subplanners decreases. This forces the bandit policy to switch between the subplanners more often.

Figure 5.3 shows the impact of combining the subplanners using the UCB1 bandit policy, compared to using each subplanner alone, on the performance of the plan quality improvement process. In this experiment we ran BDPO2 five times, once with each subplanner as the only subplanner, once combining two of them (IBCS and PNGS) using the bandit policy, and once combining the two using a simple "alternation" policy which selects each of the two in turn. Every run of BDPO2 uses experimental setup 3 (as described in Section 3.3 on page 42). The score of a plan is calculated as before (see page 44). Clearly, combining multiple subplanners in some fashion contributes more quality improvement (measured by the average IPC quality score) to BDPO2 across the entire time scale than that achieved by running BDPO2 using any individual subplanner. The figure also shows that combining multiple subplanners using the bandit policy is a better strategy than simply alternating between them. In fact, in this experiment, where the input plans were of high quality, the total quality improvements (measured by the average IPC quality score) found by BDPO2 using the alternation policy is 6.8% less than that found by BDPO2 using the bandit policy.

5.5 Ranking Selection in BDPO2

Recall from Section 4.7 that we identify four structural metrics (called ranking policies) to rank the candidate windows according to how likely they are to be improved, after investigating the properties of a large collection of improved and unimproved windows. The candidate windows are attempted for optimisation according to the order set by the ranking policies. However, the orderings usually vary for one ranking policy to another. Apart from the subplanner selection, the role of on-line adaptation is limited to switching between alternative ranking policies. The window selected for optimisation by a subplanner is the top one in the order given by the current ranking policy for that subplanner (cf. Section 4.7). As long as improvements are found among these windows, we can consider the current policy to be useful. When a subplanner reaches a certain number of attempts with no improvements found, we switch to using the next policy for that subplanner. Since the number of windows optimised is, in most cases, relatively small compared to the number of candidate windows generated, the ranking policies have more influence over which windows are tried than the windowing strategies. Thus, adapting the ranking policy rather than the windowing strategies has a greater effect on system performance. The benefit of using window ranking policies on the overall plan improvement process has

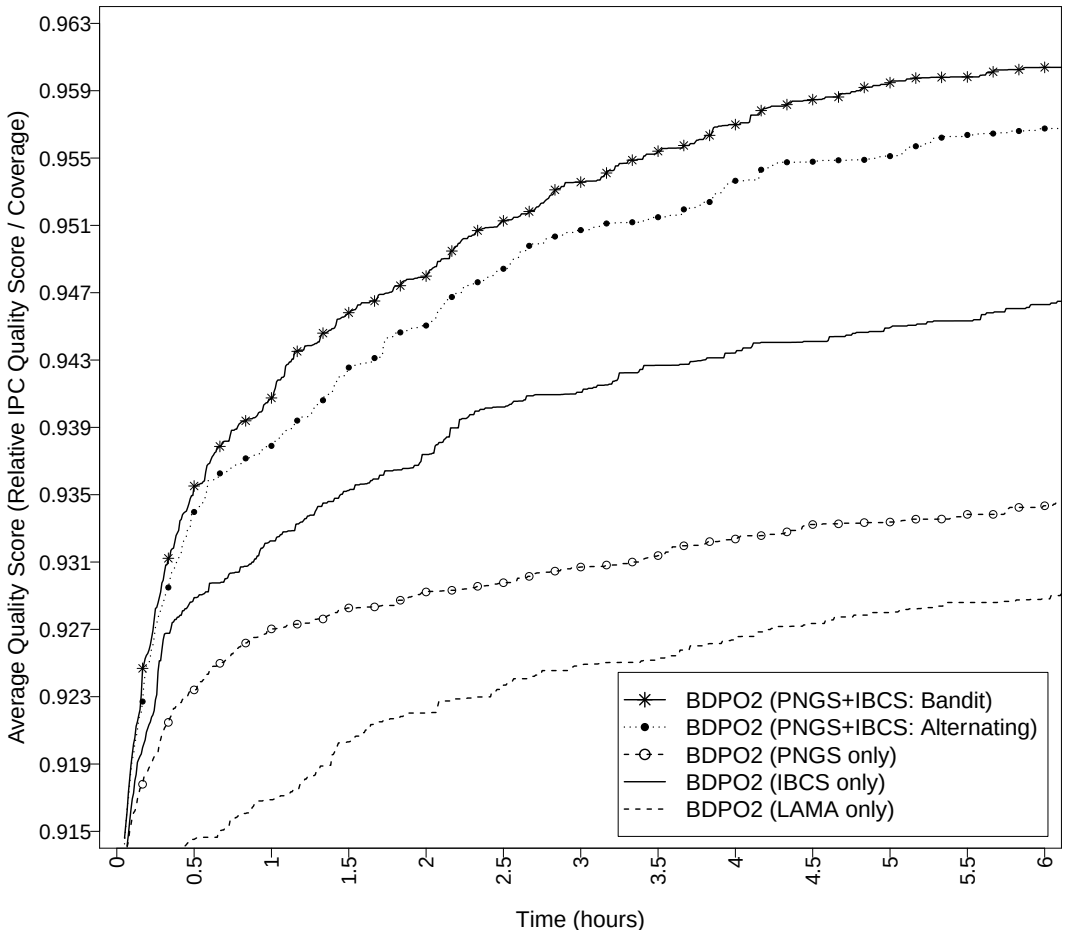


Figure 5.3: Average IPC quality score as a function of time per problem in five different runs of BDPO2: using only each one of the three subplanners, using two of them (IBCS and PNGS) combined with the UCB1 and without the UCB1 (using simple alternation policy instead). This experiment was run with setup 3 as described in Section 3.3 (on page 42). Note that the y-axis is truncated (started at 0.907) and the x-axis shows the runtime of BDPO2 (which started at 2 hour in the overall experiment shown in Figure 3.1).

been experimentally evaluated. A summary of the experimental results is given in Figure 4.8 (on page 81).

5.6 Summary

This chapter describes the on-line adaptation, an important component of our plan optimisation system, where we auto-tune the system for the current problem, over the course of optimisation, in order to achieve better quality plans. In doing so, we apply UCB1, a popular multi-armed bandit learning technique, to exploit useful subplanners as much as possible, since subplanners' productivity varies from domain to domain. Our experiments clearly suggest that the bandit policy improves the overall plan quality across the problems (measured by the average IPC plan quality score) more than what is achieved by using any one single subplanner for all problems, across the entire time scale. We also describe how we adapt the ranking policies to the current problem to prioritise the subplans that are more likely to be improvable by individual subplanners.

Recall that we have three contributions in this thesis: plan decomposition, continuing plan quality optimisation, and macro generation. We have discussed the first two contributions up to this chapter. The next chapter explains the third contribution: macro generation from block deordered plans, and how it is useful for improving planners' efficiency.

Chapter 6

Block-based Macro Generation

This chapter describes the third contribution of this thesis: block-based macro generation. Macros are a well known and widely studied kind of planning knowledge that can be easily encoded in the domain model, and thus can be exploited by standard planning engines in order to improve their efficiency and coverage.

The macro generation process, like our window generation process (described in Chapter 4) for plan optimisation, is based on block-deordered plans. In such deordering, plans are divided into meaningful subplans, called blocks, by encapsulating more effects and preconditions within a subplan, and reducing interference with steps outside the subplan. According to the nature of blocks, they can straightforwardly be transformed into purposeful macros. Note that our macro generation process, unlike the window generation process, is not concerned with plan cost, but focuses on finding macros beneficial for helping planners to find plans efficiently.

This chapter is structured as follows. Section 6.2 describes the preliminaries and terminology used in macro generation. Our overall block-based macro generation process, implemented in a system named BloMa, is described in detail in Section 6.3. This description is followed by an analysis of our experimental results in Section 6.4. Finally existing work related to our macro generation process are discussed in Section 6.5.

This chapter describes my contribution to the work published as "Exploiting Block Deordering for Improving Planners Efficiency" [Chrpa and Siddiqui, 2015] in more detail.

6.1 Introduction

Capturing and exploiting structural knowledge of planning problems has shown to be a successful strategy for making the planning process more efficient. Solutions (i.e., plans) to a planning problem that show the trajectory of achieving goals in the problem landscape are good sources of such structural knowledge. Theses solutions can be used for automatic re-engineering a domain model in order to reduce the problem complexity, and thus can help to improve planners' efficiency.

One common approach to domain remodelling is to add macro-operators ("macro", for short). A macro-operator is formulated by assembling a group of planning

operators, which is generally based on investigating training solutions, for example, by exploring groups of actions (grounded instances of these operators) that are often placed successively in the training solutions. It is, in fact, logical to recommend that if a sequence of actions frequently occur in solution plans it might be a good sequence for the planner to consider. Macros can be encoded in the same format as original planning operators in the domain model, and thus can be used in a planner-independent way.

Macros date back to 1970s where they were used, for example, in STRIPS [Fikes and Nilsson, 1971] and REFLECT [Dawson and Siklóssy, 1977]. Korf [1985] later motivates the use of macros by showing that macros can reduce the problem complexity in a number of cases. Since then many successful macro learning techniques have been developed (see Section 6.5).

Generating macros from training plans so that the macros are useful for improving planners' efficiency is not simple. Totally ordered plans, as training plans, for example, often hide some promising candidates for macros, since the corresponding actions of a useful macro may not be adjacent in these totally ordered plans. MUM [Chrpa et al., 2014] is a system that can form macros by taking non-adjacent actions of a totally ordered plan after investigating the pair-wise action dependencies in the plan. However, the technique is often unable to detect unnecessary orderings among groups of actions, which can limit the ability of the technique to capture promising macro candidates. A block deordered plan (as described in Chapter 2), on the other hand, is a decomposition into meaningful non-interfering subplans with significantly reduced number of ordering constraints among themselves. A block in a block deordered plan, because of its nature, often represents a single compound activity useful for efficient planning, and therefore is a good candidate by itself for a macro.

In this chapter, we first describe how to extract subplans that are promising candidates for macros from a block deordered plan by utilising structural relations among blocks. These macro candidates often have longer subplans representing important high level activities. Then we describe our macro generation system (BloMa) that automatically extracts domain-specific macros from the macro candidates extracted from the block-decomposed training plans. BloMa can generate useful longer macros in problems whose structure relies on repetitive application of a larger sets of actions. Traditional macro learning techniques that are based on "operator chaining" approaches (i.e. assembling operators one by one) are often not able to find such long macros. BloMa is evaluated by using the IPC benchmarks with state-of-the-art planning engines, and shows considerable improvements (in terms of IPC score and coverage) in some domains.

6.2 Background

Classical planning deals with finding a sequence of actions transforming the environment from some initial state to a desired goal state, where the environment is

described by propositions. In Chapter 2, we described the set-theoretic representation of a planning problem, which is defined by a tuple $\Pi = \langle \mathcal{M}, \mathcal{A}, \mathcal{C}, I, G \rangle$, where $\mathcal{M}$ is a set of propositions, called atoms, $\mathcal{A}$ is a set of actions, $\mathcal{C}$ is a cost function on actions, $I \subseteq \mathcal{M}$ is the initial state, and $G \subseteq \mathcal{M}$ is the goal.

A planning problem can also be represented in a classical style, where we generalize the set-theoretic representation scheme using notations derived from first-order logic. In this representation, atoms are predicates (unlike propositions in set-theoretic representation) that are true or false within some interpretation, and actions are represented by planning operators that change the truth values of these atoms. A planning operator $o = (\text{name}(o), \text{pre}(o), \text{add}(o), \text{del}(o))$ is a generalized action (i.e. the action is a grounded instance of the operator), where $\text{name}(o) = n(x_1, ..., x_k)$ (n is an unique operator name, and $x_1, ..., x_k$ are variable symbols as arguments appearing in the operator) and $\text{pre}(o)$, $\text{add}(o)$, and $\text{del}(o)$, are sets of (unground) predicates (i.e., atoms and negations of atoms), representing the preconditions, add, and delete effects of o respectively. The set-theoretic representation can be obtained from the classical representation by grounding. A planning domain represented in one of these representations can also be represented using the other representation.

Given the above preliminaries on planning operators in classical representation of planning problems, the rest of this section describes a theoretical background on macro-operators and entanglements necessary to explain our macro generation procedure.

6.2.1 Macro-operators

A macro-operator (or macro, for short) is a sequence of operators that are aggregated and added to a domain definition as a new individual operator. In the well known BlocksWorld domain [Slaney and Thiébaux, 2001], for example, we can observe that the operator $\text{pickup}(?\text{x})$ is often followed by the operator $\text{stack}(?\text{x}, ?\text{y})$. Hence, it might be reasonable to add a macro $\text{pickup-stack}(?\text{x},?\text{y})$ in the domain definition which moves a block $?\text{x}$ from a table directly to the top of the block $?\text{y}$, bypassing the situation where the block $?\text{x}$ is held by the robotic hand.

A macro is constructed by assembling a sequence of planning operators. A general procedure to do that could be as follows: First, the procedure starts with an empty macro (i.e., empty sets of operators, variables, preconditions, and effects). At each step it appends one operator from the sequence to the current macro, and fixes the variable mapping (described later in this subsection) between the new operator and the macro. Adding a new operator o to a macro m modifies the precondition, add, and delete effects of m according to the following rule.

Rule 1. *The precondition, add, and delete effects of a macro* $o_{i,j} = (\text{name}(o_{i,j}), \text{pre}(o_{i,j}), (\text{add}(o_{i,j}), \text{del}(o_{i,j}))$, *formed by assembling two operators* $\langle o_i, o_j \rangle$, *according to their sequence, where* $\text{pre}(o_j) \cap \text{del}(o_i) = \emptyset$, *are as follows:*

- $\text{pre}(o_{i,j}) = \text{pre}(o_i) \cup \{\text{pre}(o_j) \setminus \text{add}(o_i)\}$
- $\text{add}(o_{i,j}) = \{\text{add}(o_i) \cup \text{add}(o_j)\} \setminus \{\text{del}(o_j) \cup \text{pre}(o_{i,j})\}$

```
(:action pickup
   :parameters (?x1 - block)
   :precondition (and (clear ?x1) (ontable ?x1) (handempty))
   :effect
   (and
      (holding ?x1))
      (not (ontable ?x1)) (not (clear ?x1)) (not (handempty)) ))
(:action stack
   :parameters (?x2 - block ?y - block)
   :precondition (and (holding ?x2) (clear ?y))
   :effect
   (and
      (clear ?x2) (handempty) (on ?x2 ?y))
      (not (holding ?x2)) (not (clear ?y)) ))
(:action pickup_stack
   :parameters (?x - block ?y - block)
   :precondition (and (clear ?x) (ontable ?x) (handempty) (clear ?y))
   :effect
   (and (clear ?x) (on ?x ?y)
      (not (ontable ?x) (not (clear ?y)) (not (holding ?x)) ))
```

Figure 6.1: Formation of pickup_stack macro.

- $\text{del}(o_{i,j}) = \{\text{del}(o_i) \setminus \text{add}(o_j)\} \cup \text{del}(o_j)$

An example of a macro is shown in Figure 6.1, where the macro pickup_stack is formulated by assembling the operators $\langle$pickup, stack$\rangle$ in the order of their sequence according to Rule 1. Longer macros that encapsulate longer sequences of original planning operators can be constructed by this approach iteratively. However, for a macro $o_{i,j}$ to be sound, o_i must not delete any predicate required by o_j, otherwise corresponding instances of o_i and o_j cannot be applied consecutively.

A variable mapping is used to check the identity between operator's predicates and macro's predicates. Two predicates are considered identical if they have the same name and the same set of parameters. The variable mapping basically tells what variables (parameters) are common in both the macro and the new operator. In our macro generation process, a macro is constructed from a linearisation of some blocks extracted from a block deordered training plan based on some rules (described in Section 6.3.1). Since we construct a parameterised macro from a grounded subsequence (i.e., subplan) of a linearised plan, from a sequence of grounded actions which are, in fact, a plan segment (i.e., subplan), the variable mapping is established by analysing which objects the operators (i.e., actions) in the subplan share. For example, aggregating the actions of a subplan $\pi_{\text{sub}}i = \langle \text{pickup(A)}, \text{stack(A, B)} \rangle$, where the block A we picked up is stacked on the other block B, will form a macro action pickup-stack(A,B). Transforming these grounded actions into parameterised operators will construct the macro pickup-stack(?x,?y) with the preconditions and effects as shown in Figure 6.1.

Macros, because they are encoded in the same way as ordinary planning opera-

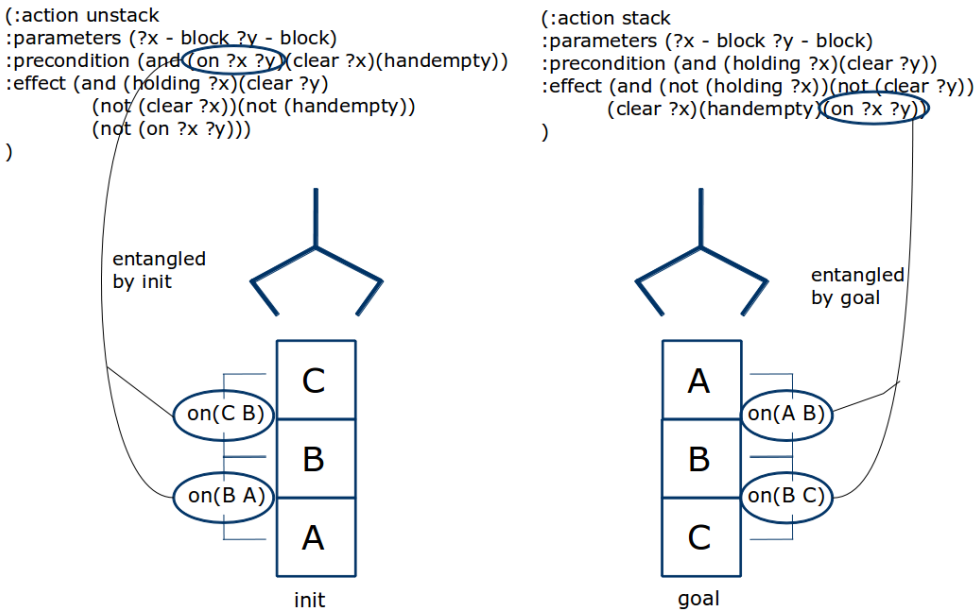


Figure 6.2: An illustrative example of outer entanglement [Chrpa et al., 2014].

tors, can be used in a planner independent way. Macros speed up planners since they can act as shortcuts to deeper states in the search tree [Botea et al., 2005; Coles et al., 2007; Newton et al., 2007]. Thus, by exploiting them it is possible to reach the goals in fewer steps, or to escape local heuristic minima. However, adding macros increase the size of grounded problem representation, and thus can be memory demanding by increasing the branching factor in the search tree. Therefore, it is important that benefits of macros' exploitation outweigh their computational cost. This problem is known as the *utility problem* [Minton, 1990].

6.2.2 Outer Entanglements

Chrpa and McCluskey [2012] introduced *outer entanglements* that define the relations between planning operators and initial or goal predicates. These relations are useful for eliminating potentially unnecessary instances of operators, as well as macros [Chrpa et al., 2014].

There are two types of outer entanglements, namely *entanglement by init* and *entanglement by goal*. An operator is entangled by init (resp. entangled by goal) with a predicate, if there exists a plan where all the operator's instances require (resp., produce) instances of the predicate that correspond to initial (resp., goal) states. In the BlocksWorld domain [Slaney and Thiébaux, 2001], for example, we might observe that unstacking blocks only occurs from their initial positions. Therefore, an entanglement by init will capture that if a predicate, say $\mathrm{on(A,B)}$, is to be achieved for a corresponding instance of the operator $\mathrm{unstack(?x,?y)}$, i.e., $\mathrm{unstack(A,B)}$, then

the predicate is present in the initial state. Similarly, we might observe that stacking blocks only occurs to their goal position. Then, an entanglement by goal will capture that a predicate on(B,A) achieved by a corresponding instance of the operator stack(?x,?y), stack(B,A), is present in the goal state. In other words, we can say that the operator unstack is entangled by init with the predicate on, and the operator stack is entangled by goal with the predicate on. This is illustrated in Figure 6.2, where we can see that if the unstack operator is entangled by init with the on predicate only the instances unstack(B,A) and unstack(C,B) follow the entanglement conditions. Thus, we can prune the rest of unstack's instances because they are not necessary to find a solution plan. Similarly, we can prune all the stack's instances except stack(A,B) and stack(B,C) if the stack operator is entangled by goal with the on predicate. Thus, outer entanglements can be used to prune potentially unnecessary instances of operators that do not follow the entanglement conditions (i.e., they are not necessary to find a solution plan). Chrpa and McCluskey [2012] showed that while in the original setting the number of operator instances grows quadratically with the number of blocks, considering outer entanglements reduces the growth to linear.

Outer entanglements have been used as a reformulation technique, as they can be directly encoded into a domain and problem model. The way outer entanglements are encoded is inspired by one of their properties: given a static predicate p_s (i.e., p_s is not present in the effects of any operator), an operator o is entangled by init with p_s if and only if $p_s \in \text{pre}(o)$ [Chrpa and Barták, 2009]. Operators involved in the outer entanglement relation with a non-static predicate are modified by putting a "static twin" of the predicate into the precondition. Instances of the "static twin" corresponding with initial or goal instances of the predicate are added into the initial state. Formally, let P be a planning problem, I be its initial state and G be its set of goal predicates. Let an operator o be entangled by init (resp. goal) with a predicate p. Then the problem P is reformulated as follows [Chrpa and McCluskey, 2012]:

1. Add a new static predicate p' having the same arguments as p to the domain model of P.
2. Add p' into o's precondition such that p' has the same arguments as p which is in precondition (resp. positive effects) of o.
3. Add instances of p' which correspond to instances of p in I (resp. in G) into I.

6.3 Block-based Macro Generation System

Our macro generation process is based on block-deordered plans, where a plan is divided into meaningful non-interleaving subplans, called blocks. Recall that a more meaningful subplan is relatively more coherent, more encapsulated, less interfering with the other parts of the plan, and often represents a purposeful activity. An example of meaningful subplans, found by the block deordering of a plan (an optimal plan, in fact) for a problem in the Gripper domain, is shown in Figure 6.3.

The Gripper problem is an easy transportation type problem with a single mobile (a robot) with a fixed capacity of holding two objects (balls) using two grippers

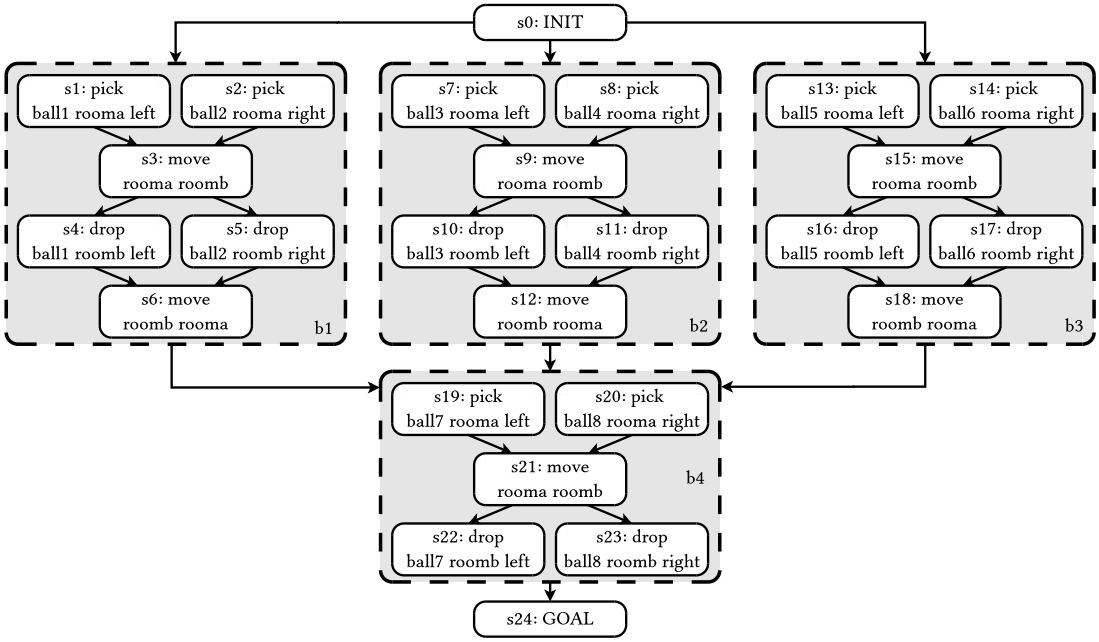


Figure 6.3: Block-deordered plan of a gripper problem with eight balls.

(which gives the domain its name). There are exactly two locations (called rooma and roomb) connected to each other. The robot can move between the rooms and pick up or drop balls with either of his two arms. Initially, all balls and the robot are in the first room. We want the balls to be in the second room.

In Figure 6.3, each of the blocks represents a purposeful activity: "comprehensive delivery", where the robot uses all its arms to perform a fully loaded delivery (of two balls) in one move. Also each block encapsulates its internal activities to perform the comprehensive delivery, and restricts the interference of the activities not within the block (which, in fact, allows the blocks b1 to b3 to be unordered). Because of this nature, each of the blocks represents a meaningful subplan, and (in this case) is a good candidate by itself for a macro. However, a block alone is not always a good candidate for a macro; a block can be too small (e.g., containing one step only) or too large (e.g., containing several component blocks) to represent a purposeful activity. Therefore, we need some principles (based on the structural properties of a block deordered plan) that group together logically connected blocks as a larger subplan resembling a single-minded activity frequently used in plans (e.g. mixing a cocktail and cleaning the shaker afterwards – as observed in the Barman domain). We call such a group of blocks a *macro candidate* (described in detail in the following subsection).

The overall procedure of our block-based macro generation is implemented in a system, called BLOMA, whose high-level design is shown in Algorithm 6. In the first step (line 1), given a set of training planning problems (simpler but not trivial),

Algorithm 6 BLOMA– a block-based macro generation system

1: $\Pi \leftarrow$ GenerateTrainingPlans()
2: $\mathcal{B} \leftarrow$ GenerateMacroCandidates(Π)
3: $M \leftarrow$ ExtractMacros($\mathcal{B}$)
4: EnhanceDomain(M)
5: $\Pi \leftarrow$ GenerateTrainingPlans()
6: FilterMacros(Π)
7: LearnEntanglements(Π)

training plans are generated using off the shelf planners. Next, macro candidates are generated from the training plans based on some rules (line 2) described in the following subsection, and the macro candidates that are generated frequently by the rules are assembled into macros (line 3). Then the formulated macros are added into the domain model (line 4), and training plans are re-generated using this macro enhanced domain model (line 5). Macros that do not appear or appear infrequently in the re-generated training plans are filtered out (line 6) from the domain description. In fact, we let the planner "decide" which macros are useful for it. The same strategy was used by MacroFF [Botea et al., 2005]. As a final step of BLOMA, like MUM [Chrpa et al., 2014], macro-specific entanglements (i.e. those where only macros are involved) are learnt (line 7) for efficient pruning of unnecessary instances of macros.

For learning entanglements, we check whether for each operator and related predicates the entanglement conditions (described in 6.2.2) are satisfied in all the training plans. Some error rate (i.e., when the entanglement conditions are violated) is allowed (typically 10%). For example, a Logistics problem might suggest that passengers can board the plane only in their initial locations and debark in their final destinations. If only a small number of passengers have to change the flights, we might still be within the "error ratio" boundary. Our entanglement learning process is similar to that of MUM (for details, see how entanglements are learnt by MUM [Chrpa et al., 2014]). Note that pruning of unnecessary instances of macros through learning entanglements does not compromise completeness because the original operators remain intact and can be used instead of incorrectly pruned instances of macros.

Whether a macro candidate or macro is "frequent" is determined relatively. Let $f^b(m)$ be a number of occurrences of a macro candidate m in the set of macro candidates $\mathcal{B}$. Then, a macro candidate $m \in M$ (M is the set of macros) is considered as frequent if $f^b(m) \geq p_b \max_{x \in M} f^b(x)$, where $0 < p_b \leq 1$. Similarly, let $f^p(o)$ be a number of occurrences of an operator or macro o in the set of macro-enhanced training plans Π. Then, a macro m is considered as frequent if $f^p(m) \geq p_p \max_{x \in O} f^p(x)$, where $0 < p_p \leq 1$ and O is a set of operators (including macros) defined in the planning domain. Clearly, setting p_b, p_p too high might cause filtering some useful macros out, while setting them too low might cause keeping useless macros.

6.3.1 Formulation of Macro Candidates

As mentioned, a block (in a block deordered plan) itself is a good candidate for a macro but not always; a block can be too small (e.g., containing one step only) to represent a purposeful combined activity. Hence we extend the blocks to larger blocks, called macro candidates, by capturing different structural relations among them in order to provide larger subplans that can capture more complex activities that are frequently occurred in plans.

We use a fixed set of rules based on structural relations among the blocks of a block deordered plan π_{bdp} in order to formulate macro candidates. Like windowing strategies (described in Chapter 4), we define relations of *immediate predecessor* and *immediate successor* of a block in π_{bdp}. Let $\text{IP}(b)$ and $\text{IS}(b)$ be sets of blocks in π_{bdp} being ordered, respectively, immediately before and after b with respect to the transitively reduced orderings of blocks. Orderings between blocks are determined by any of the necessary reasons (PC, CD, and DP) as described in Section 2.2.

In macro candidate formulation, we also make use of the *causal follower blocks* of a producer p of an atom m, $\text{CFB}_{\langle m,p\rangle}$ (according to Definition 17 on page 73). In brief, the *causal followers* of a producer p with respect to an atom m, $\text{CF}_{\langle m,p\rangle}$, are a set of steps $\{p, s_j, ..., s_k\} \setminus \{s_\text{I}, s_\text{G}\}$ (s_I and s_G are the "init" and "goal" steps respectively) such that $\{\langle p, m, s_j\rangle, ..., \langle p, m, s_k\rangle\}$ are causal links. The causal follower blocks with respect to a producer p of an atom m, $\text{CFB}_{\langle m,p\rangle}$, is the set of blocks, where each block contains at least one element of $\text{CF}_{\langle m,p\rangle}$.

We use both *basic* and *extended* blocks of a block deordered plan for macro candidate generation. Recall that basic blocks are the blocks generated by the block deordering of a plan, while extended blocks are generated by encapsulating non-branching subsequences of basic blocks (as described in Algorithm 4 on page 66). In other words, extended blocks put together blocks that are strictly consecutive in block deordered plans. Thus, they are supposed to capture larger activities (e.g. shaking a cocktail and cleaning the shaker). It should be noted that if no deordering is possible, there will be a single extended block encapsulating the whole plan.

Having described the structural relations (among blocks) above, we use the following eight rules to construct macro candidates based on those relations.

$$\begin{aligned}
&\text{R1} : \{b\} && \text{R5} : \text{R2}(\text{R4})\\
&\text{R2} : \{b\} \cup \text{IP}(b) && \text{R6} : \text{R3}(\text{R4})\\
&\text{R3} : \{b\} \cup \text{IS}(b) && \text{R7} : \text{R4}(\text{R4})\\
&\text{R4} : \{b\} \cup \text{IP}(b) \cup \text{IS}(b) && \text{R8} : \text{CFB}_{\langle m,p\rangle}
\end{aligned}$$

A rule is applied over each basic block b to construct one single macro candidate. Rule R1, for example, constructs a macro candidate by simply taking a single block b, whereas R2, R3, R4 capture sequences of neighbouring blocks of b determined by $\text{IP}(b)$, $\text{IS}(b)$, or both. Unlike windowing strategies (described in Chapter 4), we do not use any rule to capture the unordered blocks of a block b ($\text{Un}(b)$) into a macro

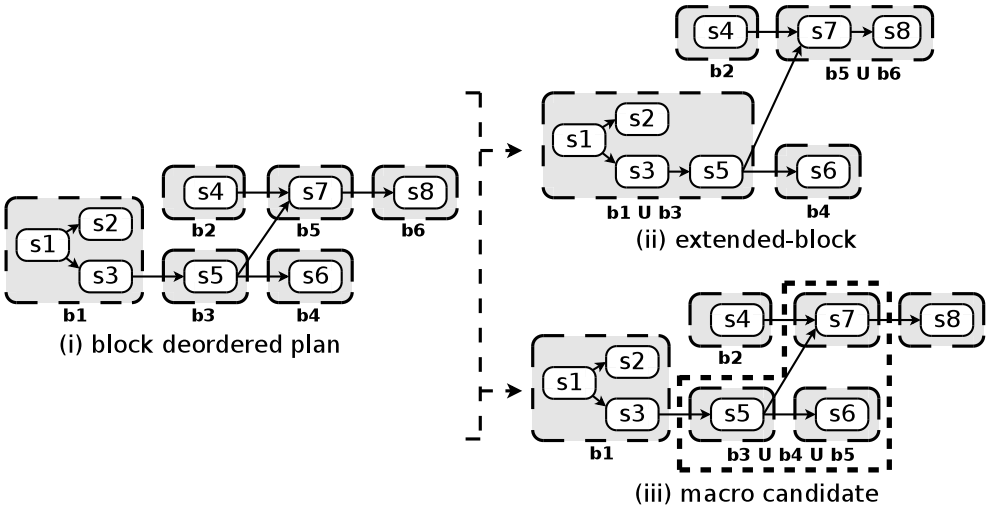


Figure 6.4: Formation of extended blocks (ii) and macro candidates (iii) from basic-blocks (1). (The macro candidate $\mathrm{b3} \cup \mathrm{b4} \cup \mathrm{b5}$ is formulated by applying the rule R3 over the basic block b3.)

candidate. Generally speaking, accumulating unordered blocks can be useful for plan optimisation, but usually not for macro generation. For example, the compound activity "delivering packages by separate trucks" can be useful to consider within a window for optimisation (which might be optimised by better use of trucks), but not as a macro because of not being internally consistent (i.e., delivering separate packages by separate trucks does not signify a single minded activity, rather increases the preconditions of the resultant macro).

The above rules can be divided into three groups, namely the primary rules (R1 to R4), the secondary rules (R5 to R7), and the causal rule (R8). The primary and secondary rules generally produce different sizes of macro candidates (measured by the number of steps they consist of). Secondary rules concatenate multiple primary rules to achieve larger macro candidates. The R5 rule (as a secondary rule), for example, says that R2 is applied over the macro candidate constructed by R4 to achieve the resultant macro candidate. The causal rule (R8) constructs macro candidates based on causal links. This rule is applied over each producer p of each atom m to group the causal followers of m (with respect to p) in a block deordered plan. The macro candidates produced by R8 are not limited to any fixed length. An example of a macro candidate formulated by applying R3 over a basic block in a block deordered plan is shown in Figure 6.4. Note that this macro candidate cannot be captured by the extended blocks.

The resultant macro candidate, after applying any rule, may not capture some intermediate blocks, i.e., blocks that are ordered in between (formally described in Definition 10 on page 68). This is illustrated in Figure 6.5 where an example of a block deordered plan is shown. In that plan, applying the R4 rule over bp results in a set of blocks not capturing the intermediately placed block bq. We incorporate

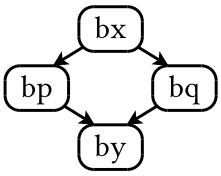


Figure 6.5: An example of intermediate blocks of a block deordered plan, where $\mathrm{IS}(\mathrm{bx}) = \mathrm{IP}(\mathrm{by}) = \{\mathrm{bp}, \mathrm{bq}\}$, and bp and bq are unordered. Applying the R4 rule over bp results in a set of blocks that do not contain the intermediately placed block bq.

the intermediate blocks after applying each rule, i.e., in this example, bq will be considered in the macro candidate constructed from bp by applying R4. Certainly if intermediate blocks are not considered, then macro candidates may represent subplans not consistent with the orderings of the block deordering. Applied to all basic blocks these rules can produce duplicates; of course, only unique macro candidates are kept.

Since too many instances of macros might hinder the benefits, BLOMA works in two phases. Initially, BLOMA tries to generate macros using extended blocks only. If no macro is generated, then BLOMA applies the rules over the basic blocks to generate macros. In order to extract macros from all the generated macro candidates, first we extract only the macro candidates that appear more frequently within the generated set. Each of these extracted macro candidates is then linearised (following the orderings of the block deordered plan), assembled into a macro (according to the principle described in Section 6.2.1), and then added into the domain model for being exploited by planners.

6.4 Experimental Analysis

We experimentally evaluated BLOMA in order to demonstrate how it improves against the original and MUM enhanced domain and problem models. The results are analysed in terms of providing insights into possible impact of generated macros to the planning process. We use IPC score as defined in the learning track of IPC-2011 [Coles et al., 2012]. For an encoding e of a problem p, $IPC(p, e)$ is 0 if p is unsolved in e, and $1/(1 + \log_{10}(T_{p,e}/T^*_p))$, where $T_{p,e}$ is the CPU-time needed to solve p in e and T^*_p is the smallest CPU-time needed to solve p in any of the considered encodings, otherwise.

6.4.1 Experiment Setup

We used all the domains from the learning track of IPC-2011, and three additional domains from the satisficing track: the Storage, Scanalyzer, and IPC-2000 version of Gripper domains. As in the learning track, the time limit was 15 minutes per

problem; all the experiments were run with satisficing planners (mentioned below) on Intel Xeon 2.53 Ghz with 2GB of RAM, CentOS 6.5.

We selected 6 training problems for each domain. The training problems were rather simple but not trivial, so the plan length was mostly within 40-80 steps; also, the training problems were not included in the test set. For learning, we used 4 state-of-the-art planners that accommodate various planning techniques: LAMA [Richter and Westphal, 2010], MpC [Rintanen, 2014], Probe [Lipovetzky et al., 2014], and Mercury [Katz and Hoffmann, 2014]. For testing, we used the same four planners that were used to generate training plans, and two more: Yahsp3 [Vidal, 2014] and Bfs-f [Lipovetzky et al., 2014]. For each domain, we considered the planner that generated the best quality (shortest) training plans. One plan was considered per each training problem. The parameters p_b and p_p (described in Section 6.3) were both set to 0.5. The learning process took from couple of seconds to couple of minutes, where generating training plans consumed the majority of the learning time.

6.4.2 Comparison

Table 6.1 presents the results of BloMa in comparison to the original problem encodings and macros generated by the benchmark approach MUM [Chrpa et al., 2014]. BloMa and MUM generated the same sets of macros in Depots and Gripper. By exploiting extended blocks, macros have been generated in the Barman, BlocksWorld, Rovers and TPP domains. In the Spanner domain, no macro has been generated. Positive results have been achieved in the Barman, Depots, Gripper and TPP domains. In the rest of domains the results were rather mixed.

Mixed results point to the fact that different planning techniques have often a different "response" to macros. This observation is, of course, not very surprising. Macros are often not supportive if the original problems are solved quickly (in a few seconds) since macros are more demanding in the pre-processing stage. Letting a planner learn macros for itself is, however, occasionally helpful, although in the Satellite domain, BloMa learnt a good macro from the training plans generated by MpC. On the other hand, when training plans are of poor quality, generated macros are very poor as well. For example, in TPP, BloMa learnt a very poor macro from the training plans generated by Probe.

In Barman, the success of BloMa rests in finding an 8-step long macro that captures an important activity – shaking a cocktail, pouring it into a shot and cleaning the shaker afterwards. In Gripper, BloMa (as well as MUM) found a useful 3-step long macro that directly delivers an object from its initial to its goal location. In other domains, BloMa found only 2-step macros capturing only partial activities (e.g. loading a truck). In TPP, BloMa generated a "recursive" macro drive-drive that a bit surprisingly contributed considerably to MpC and Probe's performance.

Apart from Barman, we identified other domains, namely Scanalyzer, Storage and the IPC-2000 version of Gripper, where BloMa found longer macros. Most problems from the Scanalyzer and Storage domains, and all problems in the IPC-2000 Gripper domain are easy to solve, that is, the planners needed at most a few

Planners	Coverage			Δ IPC		Coverage			Δ IPC		Coverage			Δ IPC	
	O	M	B	M	B	O	M	B	M	B	O	M	B	M	B
	Barman					BlocksWorld					Depots				
Lama	0	-	30	-	+30.0	23	-	25	-	+3.3	0	2	2	+2.0	+2.0
Mercury	23	-	30	-	+9.8	19	-	8	-	-11.1	0	0	0	0.0	0.0
MpC	0	-	0	-	0.0	0	-	0	-	0.0	18	24	24	+8.6	+8.6
Probe	3	-	22	-	+19.7	24	-	25	-	+3.5	30	30	30	+4.2	+4.2
Yahsp	0	-	11	-	+11.0	28	-	22	-	-7.5	21	20	20	+1.0	+1.0
Bfs-f	30	-	30	-	-6.5	0	-	10	-	+10.0	4	21	21	+17.8	+17.8
	Gripper					Parking					Rovers				
Lama	0	30	30	+30.0	+30.0	3	-	0	-	-3.0	27	29	25	+3.6	-3.3
Mercury	0	4	4	+4.0	+4.0	6	-	3	-	-3.1	24	26	29	+6.7	+9.5
MpC	0	0	0	0.0	0.0	5	-	0	-	-5.0	5	5	5	-0.1	0.0
Probe	0	5	5	+5.0	+5.0	3	-	4	-	+1.2	28	27	19	-1.0	-11.9
Yahsp	0	0	0	0.0	0.0	0	-	4	-	+4.0	30	30	30	+0.6	-0.4
Bfs-f	0	0	0	0.0	0.0	5	-	5	-	-0.9	0	0	0	0.0	0.0
	Satellite					Spanner					TPP				
Lama	3	26	18	+23.7	+15.7	0	0	-	0.0	-	16	15	17	-2.0	+1.0
Mercury	19	11	13	-10.7	-9.2	0	0	-	0.0	-	19	16	20	-4.0	+2.7
MpC	1	0	0	-1.0	-1.0	30	30	-	+1.7	-	9	11	20	+2.0	+14.0
Probe	0	2	0	+2.0	0.0	0	0	-	0.0	-	12	14	17	+2.2	+7.1
Yahsp	16	27	16	+4.7	-6.8	0	0	-	0.0	-	30	30	30	-0.7	+0.4
Bfs-f	0	0	0	0.0	0.0	0	0	-	0.0	-	15	15	8	-0.7	-8.3
	Scanalyzer					Storage					Gripper				
Lama	18	18	18	-0.8	-2.6	19	24	21	+4.9	-1.6	20	20	20	-6.7	-8.5
Mercury	20	20	20	-1.3	-1.6	20	23	21	+2.4	-1.1	20	20	20	-5.8	-7.1
MpC	16	16	17	-1.3	-1.1	30	28	24	-2.8	-12.2	20	20	20	-6.8	-6.6
Probe	17	17	17	-0.5	-3.4	21	21	28	-1.0	+4.5	20	20	20	-4.1	-9.6
Yahsp	17	18	17	0.0	-3.1	22	22	21	+2.1	-0.7	20	20	20	-1.8	-3.6
Bfs-f	18	17	17	-1.1	-2.3	23	26	25	+5.7	+3.0	20	20	20	-5.4	-11.4

Table 6.1: Comparison between original (O), MUM (M) and BLOMA (B) on the IPC-2011 learning track domains, and three more domains: the Storage, Scanalyzer, and Gripper. "-" stands for "no macros found". Δ IPC stands for a difference of IPC score of the original and the corresponding macro enhanced encodings.

seconds. With higher pre-processing requirements for macros, there is no space for improvement, although the performance was rarely considerably worse. In Storage, the learnt macro helped Probe to solve 7 more problems, Bfs-f and LAMA to solve 2 more problems, however, the macro caused MpC to solve 6 less problems. In Scanalyzer, the learnt macro was specific for "2-cycle problems" [Helmert and Lasinger, 2010]. While considering the macro, MpC solved 1 more problem, and Mercury has better performance on harder problems. On the contrary, Bfs-f solved 1 less problem.

6.4.3 Discussion

As discussed before, BLOMA is particularly useful in domains where longer macros capturing important activities can be identified. Traditional "chaining-based" ap-

proaches (e.g. MUM) often fail in these occasions. In other domains, BloMa performs with mixed results.

The number of instances of macros might cause issues with memory consumption as well as make pre-processing more difficult. A typical example is the Parking domain that represents a combinatorial problem of re-arranging cars in a parking lot. Therefore, macros despite being frequently used in plans might have detrimental effect on performance. From this perspective, approaches such as MUM that consider only macros with a relatively small number of instances are efficient. However, they are not able to generate longer macros that can be very beneficial.

On the other hand, we have observed that in some cases macros have arguments that are not necessary to be kept explicitly, so the number of instances might be reduced considerably. In particular, if a state of some object remains the same, i.e., no predicates containing this object are added or deleted, then it is not necessary to keep this object as an argument of the macro. For example, having a macro that represents an activity of delivering a package by a truck and returning the truck to the place of package's origin. The truck in fact does not change its state after applying the macro. Of course, some truck must be in the place of package's origin which has to be reflected in macro's precondition. It can be done by using existential quantifiers. Although they are supported in PDDL [McDermott, 2000], many planning engines do not support such a feature.

The impact of particular macros depends on the planning technique exploiting them. In Depots, macros bypass situations where a crate is held by a hoist. It is well known that a hoist can hold at most one crate at time, however, delete-relaxed heuristics ignore such a constraint which might lead into having many local minima in the heuristic landscape [Hoffmann, 2011]. Macros that reduce the complexity of the Planning Graph (e.g. smaller number of layers, less mutexes) seem to be beneficial for techniques such as those incorporated in MpC as observed in Depots and TPP. We believe that classifying macros according to their features will be helpful for selecting planner-specific macros that will be tailored for a given planning technique.

6.5 Related Work

Using macros has shown to be a successful strategy to speed up automated planning and problem solving processes. Hence many successful macro generation techniques have been developed, ranging from STRIPS [Fikes and Nilsson, 1971] and REFLECT [Dawson and Siklóssy, 1977] to some recent techniques, e.g., Macro-FF [Botea et al., 2005], MARVIN [Coles et al., 2007], Wizard [Newton et al., 2007], the macro generation system developed by Alhossaini and Beck [2013], and MUM [Chrpa et al., 2014]. These techniques can broadly be categorised as planner-independent or planner-specific.

Alhossaini and Beck [2013] proposed a technique for efficient selection of problem-specific macros from a set of macros learnt by some existing techniques. Marvin [Coles et al., 2007] combines off-line and on-line macro generating techniques

in order to help FF-based planners to escape plateaus in the heuristics landscape. It also learns macro-operators from plans of reduced versions of the given problems (after eliminating symmetries).

Macro-FF [Botea et al., 2005], one of the best work in the area of macro generation, has two versions of macro generation, planner (FF [Hoffmann and Nebel, 2001] in this case) dependent SOL-EP version and planner independent CA-ED version. Macro-FF is developed based on the analysis of static predicates and action sequences (found in training plans). WIZARD [Newton et al., 2007] is designed for arbitrary planners, and learns macros by exploiting genetic programming. DHG [Armano et al., 2004] generates macros from a graph of dependencies between (grounded) actions, where a macro is generated from each acyclic path in the graph. DHG filters macros that violate state invariants [Fox and Long, 2000]. Thus, DHG is an on-line and planner independent technique. A more recent macro learning method, called MUM [Chrpa et al., 2014], is based on Chrpa's [2010] method considering non-adjacent actions in training plans, and exploits outer entanglements [Chrpa and McCluskey, 2012] as a heuristics for limiting the number of potential instances of macros.

In comparison to our method BloMa, most macro learning methods, regardless of being planner independent or not, formulate macros based on a subsequence of steps from totally ordered training plans. i.e., assembling operators one by one. Such approaches, however, usually fail to find longer macros, since the "chaining" style of learning prefers shorter macros. BloMa uses macro candidates that can group together cohesive steps of a plan from different parts of the plan in a planner independent way, hence BloMa can capture longer macros that represent a purposeful compound activity.

Longer (or bigger) macros, on the hand, may generate a large number of instances in some domains, which can be memory demanding. Hence there are some works that go in the opposite direction. Haslum and Jonsson [2000], for example, proposed a method that identifies and removes "redundant actions" (i.e. actions whose effects can be achieved by sequences of other actions). Areces et al. [2014] also proposed a technique for decomposing more complex operators into simpler ones.

6.6 Summary

This chapter presents a new planner independent macro generation technique, implemented in the system BloMa. The system uses macro candidates extracted from the block deordering of training plans, which can encapsulate meaningful subplans useful for planners. Such an approach is beneficial especially for domains (e.g. Barman, Scanalyzer, Storage etc.), where an important activity repeatedly occurred in plans can be encapsulated by (longer) macros. We have shown empirically that BloMa can considerably improve the performance in many cases.

There are two major limitations. Firstly, a higher number of macro instances might be detrimental to the planning process. However, all the arguments of macros do not have to always be explicitly defined. To address this we have to use exis-

tential quantifiers in macros' preconditions; however, such a feature is not widely supported by planning engines. Alternatively, we might learn HTN methods rather than macros. Secondly, some macros are not supportive for certain planning techniques (e.g. they might "jump" into local heuristics minima). Hence, classifying macros according to their properties might reveal what kind of macros has positive or negative impact on certain planning techniques.

Chapter 7

Conclusion

Producing high-quality plans and producing them fast are two key research agendas in automated planning. Existing planners are able to find plans quickly (for example, greedy heuristic search-based planners), but usually the quality of those plans are of poor quality. Planners that guarantee solution optimality, or bounded sub-optimality, do not scale to large problems. Therefore, a gap exists between the capabilities of these two classes of planners, and it is very difficult (if not impossible) to achieve the two agendas simultaneously (i.e., achieving high-quality plans in short time) for large problems. Hence much progress has been made separately on those two agendas, and still there exists a lot of scope to progress further.

This thesis, first of all, contributes separately to those two agendas, and their realisation in two separate systems, namely BDPO2 and BloMa, respectively, for finding high quality plans and finding a plan fast. Both systems are based on block deordering – a new plan decomposition technique to identify coherent subplans useful for improving both plan quality and planners' efficiency. Figure 7.1 shows a summary view of all three contributions grouped together under one umbrella. Here, our plan optimisation and macro generation techniques, based on block-structured plan decomposition, are applied separately to achieve the two agendas of high quality plans and planners' efficiency respectively.

Our contributions, however, lie not only in achieving the above mentioned two agendas separately, but also in bridging the gap between them, meaning that in addition to the ability to find a plan quickly, we are able to achieve a continuing improvement of plan quality at any time scale (i.e., from smaller to larger time scale). Anytime planning, which aims to deliver a continuing stream of better plans given more time, is an attractive idea, offering the flexibility to stop the process at any point, such as when the best plan found is "good enough" or the wait for the next plan becomes "too long". The current best anytime planners, however, mostly remain unproductive at larger time scales; the main reason for most of the planners is that they run out of memory. LAMA, for example, between 1 and 7 hours CPU time, increases the average quality score by only 2.8%, while an increase of at least 11.9% is still possible. Our plan improvement system BDPO2 contributes to the agenda of producing high quality plans in a truly anytime fashion, using a novel approach which is based on the large neighbourhood local search strategy [Shaw, 1998], using windowing heuristics to select candidate windows from a block deordering of

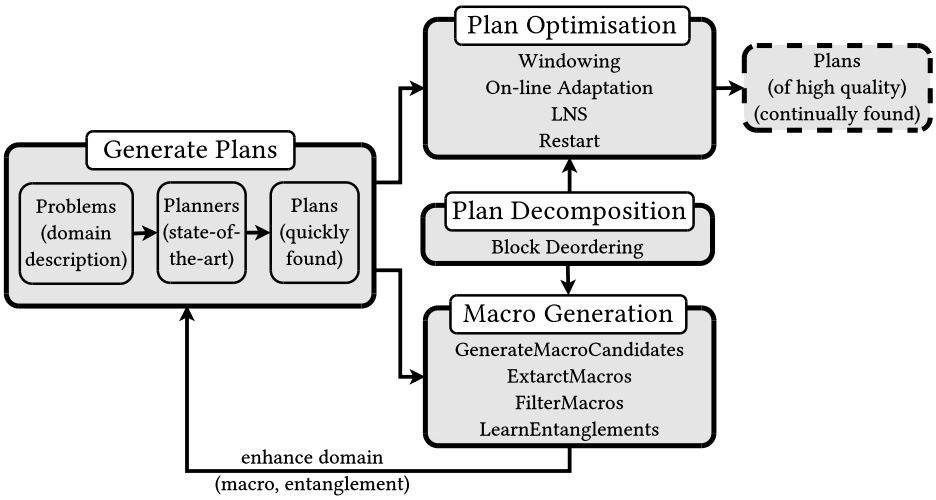


Figure 7.1: Overall contributions.

the current plan, for local optimisation using off-the-shelf bounded-cost planning techniques. We have compared our plan optimisation technique with the current best planners (e.g., LAMA, IBaCoP2, AEES, PNGS, ARVAND, and IBCS), and have shown empirically that in most domains we can achieve significant plan quality improvements in a continuous fashion at larger time scales.

In respect of improving planners' efficiency, we present a new planner independent macro generation technique, implemented in BloMa, which is based on the block deordering of training plans. Macros, in brief, are planning knowledge that can be encoded directly into the domain model as planning operators in order to be used in a planner-independent way. We have shown empirically that our macro generation technique finds macros that helps BloMa to improve planners coverage and efficiency (measured by the IPC efficiency score) in a number of domains (e.g., Barman, Depots, Gripper, TPP, Scanalyzer, and Storage) compared to MUM [Chrpa et al., 2014] – the current best macro learning technique.

Blocks in the block deordered plans, because of their nature, are very useful for both plan optimisation and macro generation. Block deordering, in fact, is interesting in its own right: it creates a decomposition of the plan into non-interleaving subplans which allow two subplans to be unordered even when their constituent steps cannot. As we have shown, a validity condition for block decomposed partially ordered plans can be stated that is almost the same as Chapman's 1987 modal truth criterion, but allowing threats to a causal link to remain unordered as long as the link is protected by the block structure (Theorem 2). Therefore, block deordering can yield less order-constrained plans, including in some cases where no conventional deordering is possible. In plan optimisation, to be specific, block deordering is essential. In an experiment, where the input plans were of high quality (found by running PNGS with 1 hour runtime cutoff on the best plans found by running LAMA or IBaCoP2 with 1 hour runtime cutoff) as described in experiment setup 3 in Section 3.3, the total

quality improvements (measured by the average IPC quality score) found by BDPO2 without performing any block deordering (i.e., based on sequential input plans) is 28.7% less than that found by BDPO2 based on block deordered input plans. In macro generation, block deordering helps to find longer macros capturing important compound activities in a number of domains (e.g., Barman, Scanalyzer, Storage, etc), which otherwise is not possible by the traditional "chaining-based" approaches (e.g. MUM).

7.1 Future Work

Experiments demonstrate that BDPO2 achieves continuing plan quality improvement even at large time scales (several hours CPU time), when other anytime planners stagnate. Key to achieving this is our focus on optimising subproblems, corresponding to windows. As mentioned in Section 4.6, our current windowing heuristics may miss some improvable windows, so extending them, and improving the on-line learning of effective window rankings, is one way to improve on this work. Also, complementing the window ranking, which estimates how "promising" a window is, with learning, on-line, an estimate of how "difficult" windows are to optimise, and using this to inform the time allocated to subplanners, which is currently uniform for all windows, may contribute to better performance. The best result, however, is achieved by chaining several techniques together (for example, applying BDPO2 to the best plan found by PNGS applied to the best plan found by LAMA). This result cannot be achieved by any of the previous anytime planning approaches alone. Thus, another area of future work is to examine, in more depth, what is the best way to combine different plan improvement methods, and how this can be learned on-line while optimising a plan.

Our macro generation technique, as mentioned in Chapter 6, may not be beneficial to the planning process if it generates a higher number of macro instances because of explicitly defining all the macro arguments. We can address this problem by using existential quantifiers in macro preconditions, or by finding and encoding the hierarchical structure of macros in some formulation technique like HTNs [Sacerdoti, 1975]. Also, we have seen that planners behave differently in different domains, and some macros are not supportive for certain planners (e.g. they might "jump" into local minima). Hence, adapting the macro generation process to the problem at hand, like the on-line adaptation of our plan optimisation system, in order to filter the macros useful for a planner at hand to solve the current problem can be an interesting future work.

OMA [Chrpa et al., 2015] is an on-line macro generation method, based on finding dependencies between planning operators extracted from the domain model, and without any offline learning from the training plans. However, it is very difficult to filter good macros using this method since there are too many possibilities to combine operators. Hence one of our ambitious in future work is the on-line generation (i.e., without using training plans) of informative blocks based on the domain knowl-

edge, which can serve as good macros.

Below are some further insights into our possible follow-up works, namely, numeric macro generation, passing improvement knowledge among symmetrical subproblems, and exploiting block deordered plans to learn HTNs.

7.1.1 Numeric Macro Generation

The plan structure uncovered by block deordering has been found useful for identifying recurrent, potentially useful, macro actions in classical planning. We believe that this block deordering technique can be extended to be used in numeric planning in order to find numeric macros. Numeric planning is used to address the real world planning problems where the execution of an action depends not only on propositional aspects but also on certain numeric conditions; the action effects can consume or renew some continuous resources, which can be represented by numeric fluents [Fox and Long, 2003] in PDDL [McDermott, 2000]. An example of such a planning problem can be given from a typical logistics domain where some numeric constraints can be associated with the problem, such as fuel consumption rate by a vehicle, money spent for refueling, etc. Recently, Scala and Torasso [2015] showed a strategy for deordering a plan for a numeric planning problem, which is an extension of the work done by Kambhampati and Kedar [1994] and Bäckström [1998] in order to deal with both propositional and numeric components (i.e., numeric fluents, numeric conditions and numeric operators). In brief, the strategy works in two phases: First it builds two validation structures, separately for propositional and numeric parts, of a given plan. Second it removes all the orderings of the input plan as long as the validity of the entailed partial order is preserved.

In block deordering, we also start with building the validation structure where the three necessary ordering reasons (described in Chapter 2): producer-consumer (i.e., causal links), consumer-deleter, and deleter-producer are taken into account. In the second phase of the block deordering, we start forming blocks using some principles that can deorder the blocks. Since the preconditions and effects of a numeric action are based on propositional and numeric terms, considering both the propositional and numeric terms while finding the necessary ordering reasons between blocks can make block deordering potentially be applicable in numeric plan decomposition. Once block decomposition of a numeric plan is achieved, we can exploit it in many other contexts of numeric planning problem, such as numeric macro generation for improving planners' efficiency (like the way we have done that for classical planning) or plan repairing (as shown by Scala and Torasso [2015]).

7.1.2 Knowledge Transfer among Symmetric Subproblems

In some domains (e.g., Barman, ChildSnack, Scanalyzer, Parcprinter, Gripper, Woodworking, etc.), we encounter many structurally similar subplans which also have symmetric improvement patterns. For instance, the blocks `b1` to `b3` in Figure 6.3 (on page 99), found by the block deordering of a plan of a Gripper problem, are symmet-

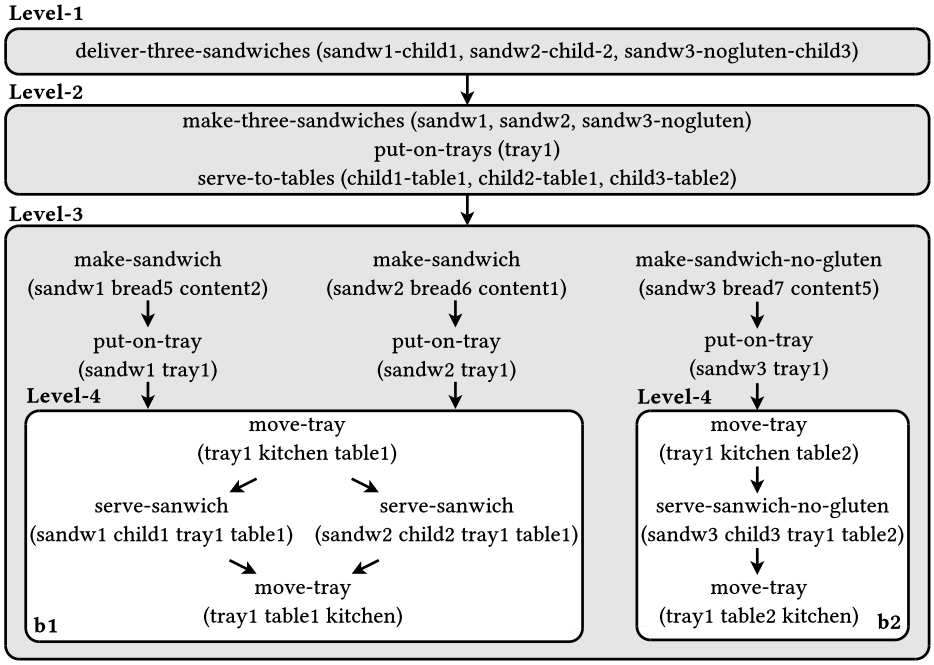


Figure 7.2: An example of hierarchical (four levels) decomposition of a task "deliver-three-sandwiches" (taken from the ChildSnack domain), where the lower two levels (Level 3 & 4) are automatically generated by the block deordering technique.

ric subplans representing the compound activity – "comprehensive delivery", where a robot uses all its arms to perform a fully loaded delivery (of two balls) in one move, and back to its original position. This type of "delivery" task can also be found in other domains, for example, delivering goods, sandwiches, and passengers, respectively, in the Logistic, ChildSnack, and Elevators domains. In these cases, transferring learned knowledge from one improved subplan to other candidate subplans, without invoking a subplanner, may be an efficient approach that can be considered in the LNS part of our plan optimisation system.

One way to apply such knowledge transfer in our plan optimisation is by first finding symmetry in the candidate subproblems (formulated from candidate windows), and then once a corresponding subplan is optimised, we can optimise the other subplans of the symmetric subproblems using the knowledge gained from the improved subplan. Abdulaziz et al. [2015] showed a sound approach for how to find isomorphic subproblems from a given planning problem, solve one instance, and then synthesize a concrete plan by concatenating instantiations of the solved one for each subproblem. This approach can be extended to be used in the above context of transferring knowledge in our plan optimisation.

7.1.3 Exploiting Blocks to Learn HTNs

Hierarchical Task Networks (HTNs) are models, mostly hand-coded, used to encode knowledge about planning domains in the form of task decompositions. The models show how abstract tasks can be decomposed into a sequence of more concrete tasks. The basic idea of HTNs dates back to the work by Sacerdoti [1975] and Tate [1977]. SHOP2 [Nau et al., 2003] is one of the popular planners that implements HTN-based domain-specific strategies for problem-solving while performing domain-independent search. Searching with HTNs explores only paths that fit into the hierarchical structure of HTNs, reducing the "fully automated" planning effort considerably. HTNs can be viewed as generalised planning with macro operators, where the domain-specific knowledge consists of not only the macro operators but also the knowledge on their hierarchical abstraction. Interestingly, the structure of block deordered plans, comprising a hierarchical (often combined with nested) decomposition of a plan into subplans, are very closely related to that of HTNs. Hence our block deordering technique can potentially be extended to generate (or help to generate) the HTN structures in a domain independent way, reducing a significant knowledge-engineering effort. Figure 7.2 shows an example of a four level hierarchical decomposition of a task "deliver-three-sandwiches", where the lower two levels (Level 3 & 4) are automatically generated by the block deordering, which leads to an easy formulation of the higher level abstractions (Level 1 & 2).

7.2 Summary

This thesis contributes to the developments of three different aspects of planning systems. First, we develop a new plan decomposition technique by extending the existing step-wise plan deordering technique to a block-wise plan deordering technique. The former is found useful by itself primarily for giving much more flexibility in plan execution than the conventional techniques. Based on this decomposition, we develop two new systems BDPO2 and BloMa, respectively, for (1) achieving continuing improvement of plan quality at any time scale (even at larger time scale when the current best techniques stagnate), and (2) macro generation for improving planners' efficiency, and empirically demonstrate their importance. Most importantly, we can achieve a significant improvement on the overall plan quality in most of the domains, as much time is given, which in general is not possible by the state-of-the-art planners.

Bibliography

Abdulaziz, M.; Norrish, M.; and Gretton, C., 2015. Exploiting symmetries by planning for a descriptive quotient. In *Proc. of the 24th International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 1479–1486. AAAI Press. (cited on page 113)

Agrawal, R., 1995. Sample mean based index policies with O(log n) regret for the multi-armed bandit problem. *Advances in Applied Probability*, (1995), 1054–1078. (cited on page 86)

Ahuja, R. K.; Goodstein, J.; Mukherjee, A.; Orlin, J. B.; and Sharma, D., 2007. A very large-scale neighborhood search algorithm for the combined through-fleet-assignment model. *INFORMS Journal on Computing*, 19, 3 (Jul. 2007), 416–428. doi:10.1287/ijoc.1060.0193. (cited on page 38)

Alhossaini, M. A. and Beck, J. C., 2013. Instance-specific remodelling of planning domains by adding macros and removing operators. In *Proc. of the 10th Symposium on Abstraction, Reformulation, and Approximation, SARA 2013, 11-12 July 2013, Leavenworth, Washington, USA*. AAAI Press. (cited on page 106)

Ambite, J. L. and Knoblock, C. A., 2001. Planning by rewriting. *Journal of Artificial Intelligence Research (JAIR)*, 15, 1 (Sep. 2001), 207–261. doi:10.1613/jair.754. (cited on page 61)

Areces, C.; Bustos, F.; Dominguez, M.; and Hoffmann, J., 2014. Optimizing planning domains by automatic action schema splitting. In *Proc. of the 24th International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*. AAAI Press. (cited on page 107)

Armano, G.; Cherchi, G.; and Vargiu, E., 2004. Automatic generation of macro-operators from static domain analysis. In *Proc. of the 16th Eureopean Conference on Artificial Intelligence, ECAI 2004, Valencia, Spain, August 22-27, 2004*, 955–956. IOS Press. (cited on page 107)

Audibert, J.-Y.; Munos, R.; and Szepesvári, C., 2009. Exploration–exploitation tradeoff using variance estimates in multi-armed bandits. *Theoretical Computer Science*, 410, 19 (Apr. 2009), 1876–1902. doi:10.1016/j.tcs.2009.01.016. (cited on pages 86 and 88)

Auer, P.; Cesa-Bianchi, N.; and Fischer, P., 2002. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47, 2-3 (2002), 235–256. doi:10.1023/A:1013689704352. (cited on pages 8, 41, 85, 86, 87, and 88)

Awerbuch, B. and Kleinberg, R., 2008. Online linear optimization and adaptive routing. *Journal of Computer and System Sciences*, 74, 1 (Feb. 2008), 97–114. doi:10.1016/j.jcss.2007.04.016. (cited on page 84)

Bäckström, C., 1998. Computational aspects of reordering plans. *Journal of Artificial Intelligence Research (JAIR)*, 9 (Sep. 1998), 99–137. doi:10.1613/jair.477. (cited on pages 16, 17, and 112)

Balyo, T.; Barták, R.; and Surynek, P., 2012a. On improving plan quality via local enhancements. In *Proc. of the 5th International Symposium on Combinatorial Search, SOCS 2012, Niagara Falls, Canada, July 19-21, 2012*. AAAI Press. (cited on page 55)

Balyo, T.; Barták, R.; and Surynek, P., 2012b. On improving plan quality via local enhancements. In *Proc. of the 5th International Symposium on Combinatorial Search, SOCS 2012, Niagara Falls, Canada, July 19-21, 2012*. AAAI Press. (cited on page 61)

Bedo, J. and Ong, C. S., 2014. Multivariate spearman's rho for aggregating ranks using copulas. *CoRR*, abs/1410.4391 (2014). (cited on page 80)

Bonet, B. and Geffner, H., 2001. Planning as heuristic search. *Artificial Intelligence*, 129, 1-2 (Jun. 2001), 5–33. doi:10.1016/S0004-3702(01)00108-4. (cited on page 58)

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J., 2005. Macro-FF: Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research (JAIR)*, 24 (2005), 581–621. doi:10.1613/jair.1696. (cited on pages 97, 100, 106, and 107)

Cenamor, I.; de la Rosa, T.; and Fernández, F., 2014. IBACOP and IBACOP2 planners. In *Proc. of the 8th International Planning Competition, IPC 2014, Deterministic Part*, 35–38. (cited on pages 6, 42, 43, and 62)

Chapman, D., 1987. Planning for conjunctive goals. *Artificial Intelligence*, 32, 3 (1987), 333–377. doi:10.1016/0004-3702(87)90092-0. (cited on pages 16 and 110)

Chrpa, L., 2010. Generation of macro-operators via investigation of action dependencies in plans. *The Knowledge Engineering Review*, 25, 03 (2010), 281–297. doi:10.1017/S0269888910000159. (cited on pages 9 and 107)

Chrpa, L. and Barták, R., 2009. Reformulating planning problems by eliminating unpromising actions. In *Proc. of the 8th Symposium on Abstraction, Reformulation, and Approximation, SARA 2009, Lake Arrowhead, California, USA, 8-10 August 2009*, 50–57. AAAI Press. (cited on page 98)

Chrpa, L. and McCluskey, T. L., 2012. On exploiting structures of classical planning problems: Generalizing entanglements. In *Proc. of the 20th European Conference on Artificial Intelligence, ECAI 2012, System Demonstrations Track, Montpellier, France, August 27-31 , 2012*, vol. 242 of *Frontiers in Artificial Intelligence and Applications*, 240–245. IOS Press. doi:10.3233/978-1-61499-098-7-240. (cited on pages 10, 97, 98, and 107)

Chrpa, L. and Siddiqui, F. H., 2015. Exploiting block deordering for improving planners efficiency. In *Proc. of the 24th International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 1537–1543. AAAI Press. (cited on page 93)

Chrpa, L.; Vallati, M.; and McCluskey, T., 2015. On the online generation of effective macro-operators. In *Proc. of the 24th International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 1544–1550. AAAI Press. (cited on page 111)

Chrpa, L.; Vallati, M.; and McCluskey, T. L., 2014. MUM: A technique for maximising the utility of macro-operators by constrained generation and use. In *Proc. of the 24th International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*, 65–73. AAAI Press. (cited on pages xviii, 9, 94, 97, 100, 104, 106, 107, and 110)

Coles, A.; Fox, M.; and Smith, A., 2007. Online identification of useful macro-actions for planning. In *Proc. of the 17th International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA, September 22-26, 2007*, 97–104. AAAI Press. (cited on pages 9, 97, and 106)

Coles, A. J.; Coles, A.; Olaya, A. G.; Celorrio, S. J.; Linares López, C.; Sanner, S.; and Yoon, S., 2012. A survey of the seventh international planning competition. *AI Magazine*, 33, 1 (2012), 83–88. (cited on pages 62 and 103)

Copeland, A. H., 1951. A reasonable social welfare function. In *University of Michigan Seminar on Applications of Mathematics to the social sciences*. (cited on page 80)

Dawson, C. and Siklóssy, L., 1977. The role of preprocessing in problem solving systems. In *Proc. of the 5th International Joint Conference on Artificial Intelligence, IJCAI 1977, Cambridge, MA, August 1977*, 465–471. William Kaufmann. (cited on pages 94 and 106)

de Borda, J. C., 1781. Memory on election ballot. *History of the Royal Academy of Sciences, Paris*, (1781), 657–664. (cited on page 80)

Dean, T. L. and McKeown, K., 1991. Systematic nonlinear planning. In *Proc. of the 9th National Conference on Artificial Intelligence, AAAI 1991, Anaheim, CA, USA, July 14-19, 1991, Volume 2.*, 634–639. AAAI Press / The MIT Press. (cited on page 16)

Dwork, C.; Kumar, R.; Naor, M.; and Sivakumar, D., 2001. Rank aggregation methods for the web. In *Proc. of the 10th International Conference on World Wide Web, WWW 2001, Hong Kong, May 1-5, 2001*, 613–622. ACM, New York, NY, USA. doi:10.1145/371920.372165. (cited on page 80)

Estrem, S. J. and Krebsbach, K. D., 2012. AIRS: Anytime iterative refinement of a solution. In *Proc. of the 25th International Florida Artificial Intelligence Research Society Conference, Marco Island, Florida. May 23-25, 2012*. (cited on page 55)

Fawcett, C.; Helmert, M.; Hoos, H.; Karpas, E.; Röger, G.; and Seipp, J., 2011. FD-Autotune: Domain-specific configuration using Fast Downward. In *Proc. of the 2011 ICAPS Workshop on Planning and Learning, PAL 2011, Freiburg, Germany, June 11-16, 2011*, 13–20. AAAI Press. (cited on page 62)

Fikes, R. and Nilsson, N. J., 1971. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2, 3/4 (1971), 189–208. doi:10.1016/0004-3702(71)90010-5. (cited on pages 94 and 106)

Fox, M. and Long, D., 2000. Utilizing automatically inferred invariants in graph construction and search. In *Proc. of the 5th International Conference on Artificial Intelligence Planning Systems, AIPS 2000, Breckenridge, CO, USA, April 14-17, 2000*, 102–111. AAAI Press. (cited on page 107)

Fox, M. and Long, D., 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)*, 20, 1 (Dec. 2003), 61–124. (cited on page 112)

Furcy, D., 2006. ITSA*: Iterative tunneling search with A*. In *Proc. of the 2006 AAAI Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications, July 16-20, 2006, Boston, Massachusetts*, 21–26. AAAI Press. (cited on page 61)

Garivier, A. and Cappé, O., 2011. The KL-UCB algorithm for bounded stochastic bandits and beyond. *CoRR*, abs/1102.2490 (2011). (cited on pages 86 and 88)

Gerevini, A.; Saetti, A.; and Vallati, M., 2009. An automatically configurable portfolio-based planner with macro-actions: PbP. In *Proc. of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*, 350–353. AAAI Press. (cited on page 62)

Gerevini, A.; Saetti, A.; and Vallati, M., 2011. PbP2: Automatic configuration of a portfolio-based multi-planner. In *7th International Planning Competition (IPC 2011), Learning Track*. http://www.plg.inf.uc3m.es/ipc2011-learning. (cited on page 62)

Gerevini, A. and Serina, I., 2002. LPG: A planner based on local search for planning graphs with action costs. In *Proc. of the 6th International Conference on Artificial Intelligence Planning and Scheduling, AIPS 2002, April 23-27, 2002, Toulouse, France*, 281–290. AAAI Press. (cited on pages 43 and 60)

Ghallab, M.; Nau, D. S.; and Traverso, P., 2004. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-856-7. (cited on page 40)

Gittins, J.; Glazebrook, K.; and Weber, R., 2011. *Multi-armed bandit allocation indices*. John Wiley & Sons. (cited on pages 86 and 87)

Godard, D.; Laborie, P.; and Nuijten, W., 2005. Randomized large neighborhood search for cumulative scheduling. In *Proc. of the 15th International Conference on*

Automated Planning and Scheduling, ICAPS 2005, Monterey, California, USA, June 5-10 2005, 81–89. AAAI Press. (cited on page 38)

Haslum, P., 2011. Computing genome edit distances using domain-independent planning. In *Proc. of the 2011 ICAPS Workshop on Scheduling and Planning Applications, SPARK 2011, Freiburg, Germany, June 11-16, 2011*. AAAI Press. (cited on pages 4, 31, 34, and 50)

Haslum, P., 2012. Incremental lower bounds for additive cost planning problems. In *Proc. of the 22nd International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*, 74–82. AAAI Press. (cited on page 46)

Haslum, P. and Grastien, A., 2011. Diagnosis as planning: Two case studies. In *Proc. of the 2011 ICAPS Workshop on Scheduling and Planning Applications, SPARK 2011, Freiburg, Germany, June 11-16, 2011*. AAAI Press. (cited on pages 31, 42, and 50)

Haslum, P. and Jonsson, P., 2000. Planning with reduced operator sets. In *Proc. of the 5th International Conference on Artificial Intelligence Planning Systems, AIPS 2000, Breckenridge, CO, USA, April 14-17, 2000*, 150–158. AAAI Press. (cited on pages 19 and 107)

Helmert, M. and Domshlak, C., 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proc. of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*, 162–169. AAAI Press. (cited on pages 41, 43, 51, 67, 68, and 77)

Helmert, M. and Lasinger, H., 2010. The scanalyzer domain: Greenhouse logistics as a planning problem. In *Proc. of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010, Toronto, Canada, May 12-16, 2010*, 234–237. AAAI Press. (cited on page 105)

Helmert, M.; Röger, G.; Seipp, J.; Karpas, E.; Hoffmann, J.; Keyder, E.; Nissim, R.; Richter, S.; and Westphal, M., 2011. Fast downward stone soup (planner abstract). In *Proc. of the 7th International Planning Competition, IPC 2011, Deterministic Part*. http://www.plg.inf.uc3m.es/ipc2011-deterministic. (cited on page 62)

Hoffmann, J., 2001. Local search topology in planning benchmarks: An empirical analysis. In *Proc. of the 17th International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, 453–458. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. (cited on page 7)

Hoffmann, J., 2011. Analyzing search topology without running any search: On the connection between causal graphs and h+. *Journal Artificial Intelligence Research (JAIR)*, 41 (2011), 155–229. doi:10.1613/jair.3276. (cited on page 106)

Hoffmann, J. and Nebel, B., 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 14 (May 2001), 253–302. doi:10.1613/jair.855. (cited on pages 59 and 107)

Hutter, F.; Hoos, H. H.; Leyton-Brown, K.; and Stützle, T., 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research (JAIR)*, 36, 1 (Sep. 2009), 267–306. (cited on page 62)

Johnson, J. D.; Li, J.; and Chen, Z., 2000. Reinforcement learning: An introduction. *Neurocomputing*, 35, 1-4 (2000), 205–206. doi:10.1016/S0925-2312(00)00324-6. (cited on pages 86, 87, and 88)

Jones, D. M. and Gittins, J., 1974. *A dynamic allocation index for the sequential design of experiments*. University of Cambridge, Department of Engineering. (cited on page 86)

Kambhampati, S. and Kedar, S., 1994. A unified framework for explanation-based generalization of partially ordered and partially instantiated plans. *Artificial Intelligence*, 67, 1 (May 1994), 29–70. doi:10.1016/0004-3702(94)90011-6. (cited on pages 17, 28, 55, and 112)

Katz, M. and Hoffmann, J., 2014. Mercury planner: Pushing the limits of partial delete relaxation. In *Proc. of the 8th International Planning Competition, IPC 2014, Deterministic Part*, 43–47. (cited on page 104)

Korf, R. E., 1985. Macro-Operators: A weak method for learning. *Artificial Intelligence*, 26, 1 (1985), 35–77. doi:10.1016/0004-3702(85)90012-8. (cited on page 94)

Lipovetzky, N.; Ramirez, M.; Muise, C.; and Geffner, H., 2014. Width and inference based planners: SIW, BFS(f), and PROBE. In *Proc. of the 8th International Planning Competition, IPC 2014, Deterministic Part*, 6–7. (cited on page 104)

McDermott, D. V., 2000. The 1998 AI planning systems competition. *AI Magazine*, 21, 2 (2000), 35–55. http://www.aaai.org/ojs/index.php/aimagazine/article/view/1506. (cited on pages 106 and 112)

Minton, S., 1990. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42, 2-3 (1990), 363–391. doi:10.1016/0004-3702(90)90059-9. (cited on page 97)

Muise, C. J.; McIlraith, S. A.; and Beck, J. C., 2012. Optimally relaxing partial-order plans with MaxSAT. In *Proc. of the 22nd International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*, 358–362. AAAI Press. (cited on pages 17, 28, and 31)

Nakhost, H. and Müller, M., 2009. Monte-Carlo exploration for deterministic planning. In *Proc. of the 21st International Joint Conference on Artificial Intelligence, IJCAI 2009, Pasadena, California, USA, July 11-17, 2009*, vol. 9, 1766–1771. (cited on pages 43 and 59)

Nakhost, H. and Müller, M., 2010. Action elimination and plan neighborhood graph search: Two algorithms for plan improvement. In *Proc. of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010, Toronto, Canada, May 12-16, 2010*, 121–128. AAAI Press. (cited on pages 6, 43, 50, 51, and 60)

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F., 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research (JAIR)*, 20, 1 (Dec. 2003), 379–404. (cited on page 114)

Nebel, B. and Bäckström, C., 1994. On the computational complexity of temporal projection, planning, and plan validation. *Artificial Intelligence*, 66, 1 (Mar. 1994), 125–160. doi:10.1016/0004-3702(94)90005-1. (cited on page 16)

Newton, M. A. H.; Levine, J.; Fox, M.; and Long, D., 2007. Learning macro-actions for arbitrary planners and domains. In *Proc. of the 17th International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA, September 22-26, 2007*, 256–263. AAAI Press. (cited on pages 9, 97, 106, and 107)

Nguyen, X. and Kambhampati, S., 2001. Reviving partial order planning. In *Proc. of the 17th International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, vol. 1, 459–464. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. (cited on page 31)

Pandey, S.; Chakrabarti, D.; and Agarwal, D., 2007. Multi-armed bandit problems with dependent arms. In *Proc. of the 24th International Conference on Machine Learning, ICML 2007, Corvallis, Oregon, USA, June 20-24, 2007*, vol. 227, 721–728. ACM. doi:10.1145/1273496.1273587. (cited on page 85)

Pednault, E. P. D., 1986. Formulating multiagent, dynamic-world problems in the classical planning framework. *Reasoning about actions and plans*, (1986), 47–82. (cited on page 17)

Pohl, I., 1970. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1, 3 (1970), 193–204. doi:10.1016/0004-3702(70)90007-X. (cited on page 58)

Policella, N.; Smith, S. F.; Cesta, A.; and Oddi, A., 2004. Generating robust schedules through temporal flexibility. In *Proc. of the 14th International Conference on Automated Planning and Scheduling (ICAPS 2004), Whistler, British Columbia, Canada, June 3-7 2004*, vol. 4, 209–218. AAAI Press. (cited on page 1)

Ratner, D. and Pohl, I., 1986. Joint and LPA*: Combination of approximation and search. In *Proc. of the 5th National Conference on Artificial Intelligence, AAAI 1986, Philadelphia, PA, August 11-15, 1986. Volume 1: Science.*, 173–177. Morgan Kaufmann. (cited on pages 55 and 61)

Régnier, P. and Fade, B., 1991. Complete determination of parallel actions and temporal optimization in linear plans of action. In *Proc. of the European Workshop*

on Planning, EWSP 1991, Sankt Augustin, FRG, March 18-19, 1991, vol. 522 of *Lecture Notes in Computer Science*, 100–111. Springer. doi:10.1007/BFb0052953. (cited on page 17)

Richter, S. and Helmert, M., 2009. Preferred operators and deferred evaluation in satisficing planning. In *Proc. of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*, 273–280. AAAI Press. (cited on page 58)

Richter, S.; Thayer, J. T.; and Ruml, W., 2010. The joy of forgetting: Faster anytime search via restarting. In *Proc. of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010, Toronto, Canada, May 12-16, 2010*, 137–144. AAAI Press. (cited on pages 3, 50, 51, and 58)

Richter, S. and Westphal, M., 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research (JAIR)*, 39 (Sep. 2010), 127–177. doi:10.1613/jair.2972. (cited on pages 42, 43, 51, 58, and 104)

Rintanen, J., 2014. Madagascar: Scalable planning with SAT. In *Proc. of the 8th International Planning Competition, IPC 2014, Deterministic Part*, 66–70. (cited on page 104)

Robbins, H., 1952. Some aspects of the sequential design of experiments. In *Herbert Robbins Selected Papers*, vol. 58, 527–535. Springer. (cited on page 85)

Ropke, S. and Pisinger, D., 2006. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40, 4 (Nov. 2006), 455–472. doi:10.1287/trsc.1050.0135. (cited on pages 7 and 38)

Sacerdoti, E. D., 1975. The nonlinear nature of plans. In *Proc. of the 4th International Joint Conference on Artificial Intelligence, IJCAI 1975, Tbilisi, Georgia, USSR, 3-8 September 1975*, 206–214. (cited on pages 111 and 114)

Scala, E. and Torasso, P., 2015. Deordering and numeric macro actions for plan repair. In *Proc. of the 24th International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 1673–1681. AAAI Press. (cited on page 112)

Schrimpf, G.; Schneider, J.; Stamm-Wilbrandt, H.; and Dueck, G., 2000. Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, 159, 2 (Apr. 2000), 139–171. doi:10.1006/jcph.1999.6413. (cited on pages 7 and 38)

Scott, S. L., 2010. A modern bayesian look at the multi-armed bandit. *Applied Stochastic Models in Business and Industry*, 26, 6 (Nov. 2010), 639–658. doi:10.1002/asmb.874. (cited on page 86)

Shaw, P., 1998. Using constraint programming and local search methods to solve vehicle routing problems. In *Proc. of the 4th International Conference on Principles and Practice of Constraint Programming, CP 1998, , Pisa, Italy, October 26-30, 1998*, vol. 1520 of *Lecture Notes in Computer Science*, 417–431. Springer. doi:10.1007/3-540-49481-2_30. (cited on pages 38 and 109)

Siddiqui, F. H. and Haslum, P., 2012. Block-structured plan deordering. In *Proc. of the 25th Australasian Joint Conference on Advances in Artificial Intelligence, AI 2012, Sydney, Australia, December 4-7, 2012*, vol. 7691 of *Lecture Notes in Computer Science*, 803–814. Springer, Berlin, Heidelberg. doi:10.1007/978-3-642-35101-3_68. (cited on pages 2, 13, 14, 21, and 23)

Siddiqui, F. H. and Haslum, P., 2013a. Local search in the space of valid plans. In *Proc. of the 2013 ICAPS Workshop on Evolutionary Techniques in Planning and Scheduling, EVOPS 2013, Rome, Italy, June 10-14, 2013*, 22–31. http://icaps13.icaps-conference.org/wp-content/uploads/2013/05/evops13-proceedings.pdf. (cited on pages 38, 49, and 50)

Siddiqui, F. H. and Haslum, P., 2013b. Plan quality optimisation via block decomposition. In *Proc. of the 23rd International Joint Conference on Artificial Intelligence, IJCAI 2013, Beijing, China, August 3-9, 2013*, 2387–2393. AAAI Press. http://ijcai.org/papers13/Papers/IJCAI13-351.pdf. (cited on page 38)

Siddiqui, F. H. and Haslum, P., 2015. Continuing plan quality optimisation. *Journal of Artificial Intelligence Research*, 54 (nov 2015), 369–435. doi:10.1613/jair.4980. (cited on page 38)

Slaney, J. and Thiébaux, S., 2001. Blocks world revisited. *Artificial Intelligence*, 125, 1-2 (jan 2001), 119–153. doi:10.1016/s0004-3702(00)00079-5. (cited on pages 95 and 97)

Slivkins, A., 2014. Contextual bandits with similarity information. *Journal of Machine Learning Research*, 15, 1 (Jan. 2014), 2533–2568. (cited on page 86)

Stern, R. T.; Puzis, R.; and Felner, A., 2011. Potential search: A bounded-cost search algorithm. In *Proc. of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany June 11-16, 2011*, 234–241. AAAI Press. (cited on pages 6 and 58)

Tate, A., 1977. Generating project networks. In *Proc. of the 5th International Joint Conference on Artificial Intelligence IJCAI 1977, Cambridge, MA, August 1977*, 888–893. William Kaufmann. (cited on page 114)

Thayer, J.; Stern, R.; Felner, A.; and Ruml, W., 2012a. Faster bounded-cost search using inadmissible heuristics. In *Proc. of the 22nd International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*, 270–278. AAAI Press. (cited on page 58)

Thayer, J. T.; Benton, J.; and Helmert, M., 2012b. Better parameter-free anytime search by minimizing time between solutions. In *Proc. of the 5th International Symposium on Combinatorial Search, SOCS 2012, Niagara Falls, Canada, July 19-21, 2012*, 120–128. AAAI Press. (cited on pages 3, 43, and 58)

Thayer, J. T. and Ruml, W., 2011. Bounded suboptimal search: A direct approach using inadmissible estimates. In *Proc. of the 22nd International Joint Conference on Artificial Intelligence, IJCAI 2011, Barcelona, Catalonia, Spain, July 16-22, 2011*, 674–679. AAAI Press. doi:10.5591/978-1-57735-516-8/IJCAI11-119. (cited on page 58)

Thompson, W. R., 1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, (1933), 285–294. (cited on pages 85, 86, and 88)

Vallati, M.; Fawcett, C.; Gerevini, A.; Holger, H.; and Saetti, A., 2011. ParLPG: Generating domain-specific planners through automatic parameter configuration in LPG. In *Proc. of the 7th International Planning Competition, IPC 2011, Deterministic Part*. http://www.plg.inf.uc3m.es/ipc2011-deterministic. (cited on page 62)

Veloso, M. M.; Pérez, A.; and Carbonell, J. G., 1990. Nonlinear planning with parallel resource allocation. In *Proc. of the DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control, San Diego, California, November 5-8, 1990*, 207–212. Morgan Kaufmann. (cited on page 17)

Vidal, V., 2014. YAHSP3 and YAHSP3-MT planners. In *Proc. of the 8th International Planning Competition, IPC 2014, Deterministic Part*, 64–65. (cited on page 104)

Wang, Y.; Audibert, J.; and Munos, R., 2008. Algorithms for infinitely many-armed bandits. In *Proc. of the 22nd Annual Conference on Neural Information Processing Systems, NIPS 2008, Vancouver, British Columbia, Canada, December 8-11, 2008*, 1729–1736. Curran Associates, Inc. (cited on page 86)

Xie, F.; Nakhost, H.; and Müller, M., 2012. Planning via random walk-driven local search. In *Proc. of the 22nd International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*, 315–322. AAAI Press. (cited on page 59)

Xie, F.; Valenzano, R. A.; and Müller, M., 2013. Better time constrained search via randomization and postprocessing. In *Proc. of the 23rd International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*, 269–277. AAAI Press. (cited on pages 3 and 46)

Young, H. P. and Levenglick, A., 1978. A consistent extension of condorcet's election principle. *SIAM Journal on Applied Mathematics*, 35, 2 (1978), 285–300. (cited on page 80)

Zhou, R. and Hansen, E. A., 2005. Beam-stack search: Integrating backtracking with beam search. In *Proc. of the 15th International Conference on Automated Planning*

and Scheduling, ICAPS 2005, Monterey, California, USA, June 5-10, 2005, 90–98. AAAI Press. (cited on pages 6, 43, and 58)