Journal of Artificial Intelligence Research 54 (2015) 369–435 Submitted 9/15; published 11/15

Continuing Plan Quality Optimisation

Fazlul Hasan Siddiqui FAZLUL.SIDDIQUI@ANU.EDU.AU
Patrik Haslum PATRIK.HASLUM@ANU.EDU.AU
The Australian National University &
NICTA Optimisation Research Group
Canberra, Australia

Abstract

Finding high quality plans for large planning problems is hard. Although some current anytime planners are often able to improve plans quickly, they tend to reach a limit at which the plans produced are still very far from the best possible, but these planners fail to find any further improvement, even when given several hours of runtime.

We present an approach to continuing plan quality optimisation at larger time scales, and its implementation in a system called BDPO2. Key to this approach is a decomposition into subproblems of improving parts of the current best plan. The decomposition is based on block deordering, a form of plan deordering which identifies hierarchical plan structure. BDPO2 can be seen as an application of the large neighbourhood search (LNS) local search strategy to planning, where the neighbourhood of a plan is defined by replacing one or more subplans with improved subplans. On-line learning is also used to adapt the strategy for selecting subplans and subplanners over the course of plan optimisation.

Even starting from the best plans found by other means, BDPO2 is able to continue improving plan quality, often producing better plans than other anytime planners when all are given enough runtime. The best results, however, are achieved by a combination of different techniques working together.

1. Introduction

The classical AI planning problem involves representing models of the world (initial and goal states) 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 sequence of actions, whose application transforms the world from the initial state to a desired goal state. Thus, planning makes an intelligent system autonomous through the construction of plans of action to achieve its goals.

A key concern in automated planning is producing high quality plans. Planners using optimal or bounded suboptimal (heuristic) search methods offer guarantees on plan quality, but are unable to solve large problems. Fast planners, using 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. We seek to address this gap by proposing a new approach to continuing plan improvement, that is able to tackle large problems and works at varying time scales.

Anytime search tries to strike a balance between 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

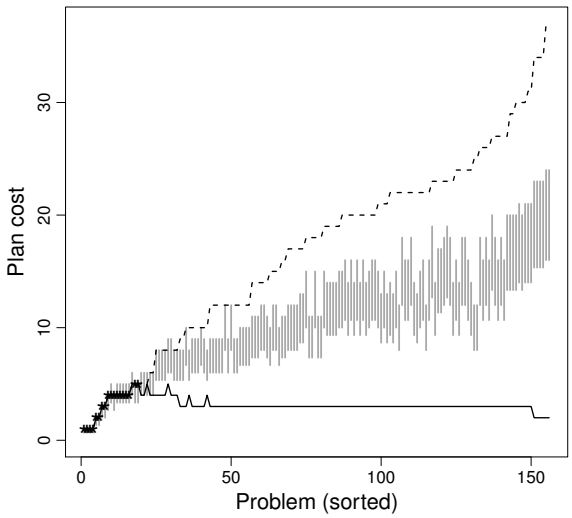

Figure 1: Illustration of the plan quality gap. The dashed line represents the best (lowest-cost) plan for 156 problems from Genome Edit Distance (GED) domain (Haslum, 2011) found by different non-optimal planners, including anytime planners. The solid line represents the corresponding highest known lower bound. The difference between these two is the optimality gap. The '⋆' points represent plans found by optimal planners, while the vertical bars show the optimality gap obtained by a problem-specific algorithm (GRIMM).

the more time they are given. Anytime search algorithms such as, for example, RWA* (Richter, Thayer, & Ruml, 2010) or AEES (Thayer, Benton, & Helmert, 2012b) have been successfully used in anytime planners. However, these planners are often not effective at making use of increasing runtime beyond the first few minutes. Xie, Valenzano, & Müller (2010) define the "unproductive time" of a planner as the amount of time remaining when it 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 problem, is more than 90%.

We have observed similar results, as shown in Figure 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 Section 3, from page 392.) LAMA finds a first solution quickly: for 92.3% of the problems it solves (within a maximum of 7 hours CPU time per problem), the first plan is found in less than 10 minutes. The quality of LAMA's plans improve rapidly early on, but the later trend is one of "flattening out", i.e., decreasing increase. (The drop at the beginning is due to the figure showing the average plan quality over solved problems: as initial, low-quality, plans for more 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 for 21.3% of solved problems. Yet for a further 51.6% of problems better plans exist, and are found by other methods. In the same time interval, LAMA's average plan quality score increases by only 2.7%, while an increase of

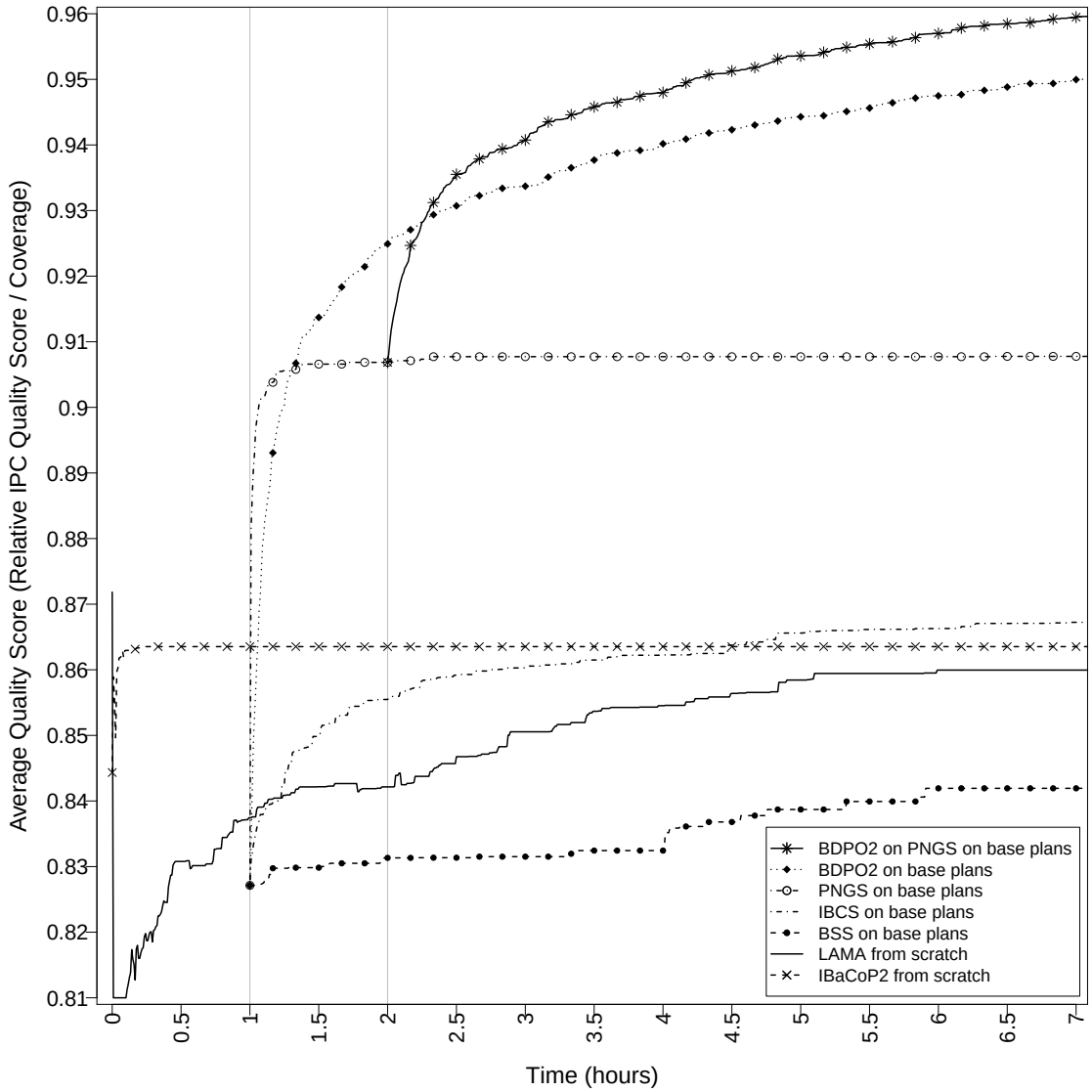

Figure 2: Average IPC quality score as a function of time per problem, on a set of 182 large-scale planning problems. The quality score of a plan is c^{ref}/c, where c is the cost of the plan and c^{ref} the "reference cost" (least cost of all plans for the problem); hence a higher score represents better 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 3.1 (page 392).

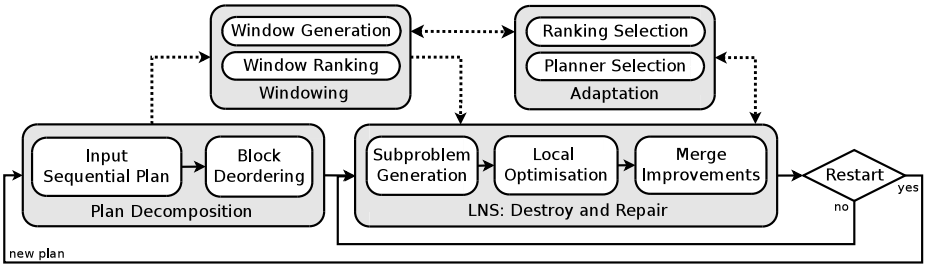

Figure 3: General framework of BDPO2

at least 14.6% is possible. Memory-limited branch-and-bound algorithms, like Beam Stack Search (Zhou & Hansen, 2005) may run indefinitely, but find improvements very slowly. The increase in average plan quality made by BSS over the entire time depicted in Figure 2 is only 1.8%.

Plan optimisation approaches based on *post-processing* start with a valid plan and seek to improve it. Figure 2 shows results for Plan Neighbourhood Graph Search (Nakhost & Müller, 2010). PNGS searches for 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 stops, as it runs out of memory.

In summary, this experiment shows that current anytime plan optimisation methods become unproductive as runtime increases, or suffer from a very slow rate of plan quality improvement.

We present a post-processing approach to plan optimisation, and its implementation in a system called BDPO2. (The source code for BDPO2 is provided as an on-line appendix to this article.) As a post-processor, BDPO2 does not work on its own: it depends on other methods providing an initial plan. 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. What Figure 2 shows is that switching to our approach after some time can overcome the limitation of current anytime planning techniques, and continue to improve plan quality as allotted time increases. 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 be achieved by previous anytime planning approaches alone.

BDPO2 uses Large Neighborhood Search (LNS), a local search technique. The local search explores a neighbourhood around the current solution plan for a better quality valid 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. The destroy methods used in LNS algorithms often contain an element of randomness, and the local search may accept moves to lower-quality solutions (Ropke & Pisinger, 2006; Schrimpf, Schneider, Stamm-Wilbrandt, & Dueck, 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 new best plan. We have found that delayed restarts allow better exploration of subplans from different parts of the current plan, and helps 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 3, broadly consists of four components: plan decomposition, LNS (i.e., the repeated destroy and repair steps), windowing, and on-line adaptation. The first step, decomposition, uses deordering to produce a partially ordered plan. Deordering enables the windowing strategies to find subplans that are easier to improve on, leading to much better anytime performance. We use block deordering (Siddiqui & Haslum, 2012), which simultaneously decomposes a given plan into "coherent" subplans, called blocks, and relaxes ordering constraints between blocks. Block deordering removes some of the inherent limitations of existing, step-wise deordering techniques, and is able to deorder sequential plans in some cases where no step-wise deordering is possible. The windowing component is a collection of strategies for extracting windows from the block deordered plan, and ranking policies which order the windows so that the system attempts to optimise more "promising" windows first.

BDPO2 extends our earlier system, BDPO (Siddiqui & Haslum, 2013b), 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 uses portfolios of window generation and ranking strategies and several subplanners. This improves the capability and robustness of the system, since no single alternative (windowing strategy, subplanner, etc.) dominates all others across all problems. Furthermore, we take advantage of the fact that the system solves many subproblems over the course of the local search to learn on-line which are the best alternatives for the current problem. In particular, we use the UCB1 multi-armed bandit learning policy (Auer, Cesa-Bianchi, & Fischer, 2002) for subplanner selection, and a sequential portfolio of window ranking policies.

The remainder of this article is structured as follows: Section 2 describes block deordering. The theory of block deordering presented here is slightly different from our earlier account (Siddiqui & Haslum, 2012), allowing for more deordering in some cases and better contrasting it with traditional partially ordered plan semantics. Section 3 presents an overview of the BDPO2 system and main empirical results, while Sections 4 and 5 give more details of the windowing and on-line adaptation components, respectively, including empirical analysis of their impact on the performance of the system as a whole. Section 6 reviews related work, and Section 7 presents conclusions and outlines ideas for future work.

2. Plan Decomposition

Our approach to continuing plan quality improvement is based on optimising a plan by parts, one at a time. Every subplan that we consider for local optimisation is a subsequence of some linearisation of a partially ordered plan. Therefore, a key step is removing unnecessary ordering constraints from the, typically sequential, input plan. This process is called *plan deordering*. The importance of deordering is demonstrated by one of our experiments (presented in Section 3.6, page 402), in which we apply BDPO2 to input plans that are already of high quality: The total plan quality improvement (measured by the increase in the average IPC plan quality score) achieved by BDPO2 without any deordering is 28.7% less than that achieved with 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 6 on page 381.) To remedy this, we have introduced *block deordering* (Siddiqui & 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 6 on page 381.) In this section, we present a new, and slightly different account of the theory and practice of block deordering. First, it relaxes a restriction on block deordered plans, thereby allowing more deordering of some plans. Second, it contrasts the semantics of block decomposed partially ordered plans with the traditional partially ordered plan semantics in a clearer way.

Sections 2.1–2.3 describe necessary background, while Sections 2.4–2.6 introduce block decomposed partially ordered plans and the block deordering algorithm.

2.1 The Planning Problem, Sequential Plan and its Validity

We consider the standard STRIPS representation of classical planning problems with action costs. A planning problem is 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 $\langle \mathrm{pre}(a), \mathrm{add}(a), \mathrm{del}(a) \rangle$, where $\mathrm{pre}(a)$, $\mathrm{add}(a)$, and $\mathrm{del}(a)$ are the preconditions, add and delete effects of a respectively. We also say that action a is a *consumer* of an atom m if $m \in \mathrm{pre}(a)$, a *producer* of m if $m \in \mathrm{add}(a)$, and a *deleter* of m if $m \in \mathrm{del}(a)$. An action a is applicable in a state S if $\mathrm{pre}(a) \subseteq S$, and if applied in S, results in the state $\mathrm{apply}(a, S) = (S \setminus \mathrm{del}(a)) \cup \mathrm{add}(a)$. A sequence of actions $\pi = \langle a_i, a_{i+1}, ..., a_j \rangle$ is applicable in a state S_i if (1) $\mathrm{pre}(a_k) \subseteq S_k$ for all $i \leq k \leq j$, and (2) $S_{i+1} = \mathrm{apply}(a_i, S_i)$, $S_{i+2} = \mathrm{apply}(a_{i+1}, S_{i+1})$, and so on; the resulting state is $\mathrm{apply}(\pi, S_i) = S_{i+j+1}$.

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

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 $\prec$. 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 $\text{PC}(m) \in \text{Re}(s_i \prec s_j)$ is usually called a *causal link* from s_i to s_j for m (McAllester & Rosenblitt, 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 $\pi_{\text{pop}} = \langle \mathcal{S}, \prec \rangle$ 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) $\exists s'_p : m \in \text{add}(s'_p) \wedge s_d \prec^+ s'_p \prec^+ s_c$ is true.*

As mentioned above, a p.o. plan, $\pi_{\text{pop}} = \langle \mathcal{S}, \prec \rangle$ 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 & 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's (1987) modal truth criterion, that

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

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 section. 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 $\pi_{\text{pop}} = \langle \mathcal{S}, \prec \rangle$ be a valid p.o. plan. A (step-wise) deordering of π_{pop} is a valid plan $\pi'_{\text{pop}} = \langle \mathcal{S}, \prec' \rangle$ 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 $\pi_{\text{seq}} = \langle a_1, ..., a_n \rangle$ can be represented as a p.o. plan with one step $s_i \in S$ for each action a_i in π_{seq} and an ordering $s_i \prec s_j$ whenever $i < j$, so that no two steps in S which are unordered. Thus, deordering a sequential plan is no different from (further) deordering a p.o. plan.

Computing a (step-wise) deordering with a minimum number of ordering constraints is NP-hard (Bäckström, 1998), but there are several non-optimal algorithms (e.g., Pednault, 1986; Veloso, Pérez, & Carbonell, 1990; Régnier & Fade, 1991). We have used a variant of the explanation-based generalisation algorithm by Kambhampati and Kedar (1994). 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 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. (The algorithm by Veloso, Pérez and Carbonell is similar, but selects the latest producer instead.) 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.

Kambhampati and Kedar's deordering algorithm, due to its greedy strategy, does not guarantee optimality. An example where it fails to transform a totally ordered plan to a least-constrained plan is shown in Figure 4. However, a recent study found that the algorithm did produce optimal step-wise plan deorderings of all plans on which it was tested (Muise, McIlraith, & Beck, 2012).

However, our motivation for plan deordering is to find a deordering that is adequate for generating useful candidate subplans for local optimisation. More important than achieving

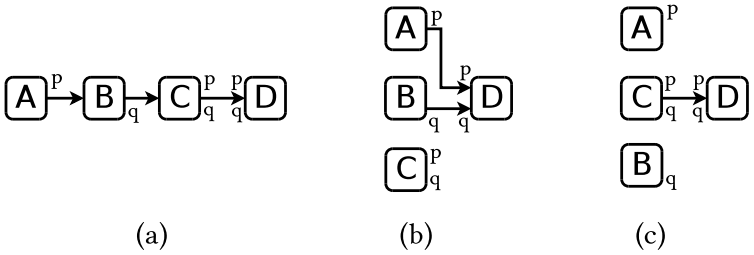

Figure 4: An example on which Kambhampati and Kedar's (1994) algorithm fails to find the least constrained plan. (Derived from Figure 14 in Bäckström's 1998 article on plan deordering.) Figure (a) is a sequential input plan, (b) is the plan produced by the algorithm after choosing the earliest producer (for the validation structure) of the preconditions p and q of D, and (c) is the minimally ordered version of (a). For simplicity, the goal atoms produced by the steps A, B, C and D are not shown in the figure.

an optimal step-wise 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.

2.4 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 5. 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 $\pi_{\text{bdp}} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$.*

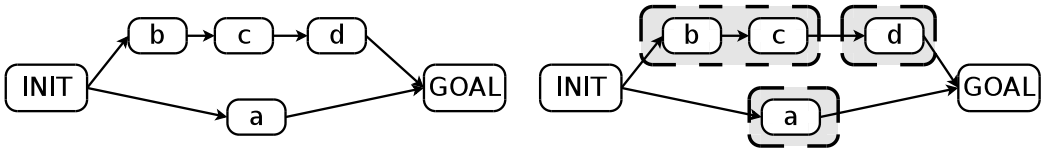

Figure 5: A normal p.o. plan (left) represents the set of all sequential plans that are linearisations of the plan steps, in this example $\langle a, b, c, d\rangle$, $\langle b, a, c, d\rangle$, $\langle b, c, a, d\rangle$, and $\langle b, c, d, a\rangle$. A block decomposed p.o. plan (shown on the 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.

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 $m \in \text{add}(\hat{s})$, 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 $m \in \text{pre}(\hat{s})$, 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 $m \in \text{del}(\hat{s})$, 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 $m \notin del(b)$.*

An example of how the block decomposition protects a causal link can be seen in Figure 7(i) on page 382.

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 the theorem cited above. 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 $s_p \prec_{\text{seq}} s_t \prec_{\text{seq}} s_c$ can not 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. □

2.5 Block Deordering

Block deordering (Siddiqui & 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 ordering constraints, but those are transitively implied by the other ordering constraints. Block deordering can often remove ordering constraints where step-wise deordering can not. 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 coherent, more "self-contained" subplans, and form the basis for the windowing strategies (described in detail in Section 4) that we use to generate candidate subplans for local optimisation.

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

As a simple example of block deordering, Figure 6(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. Block deordering, however,

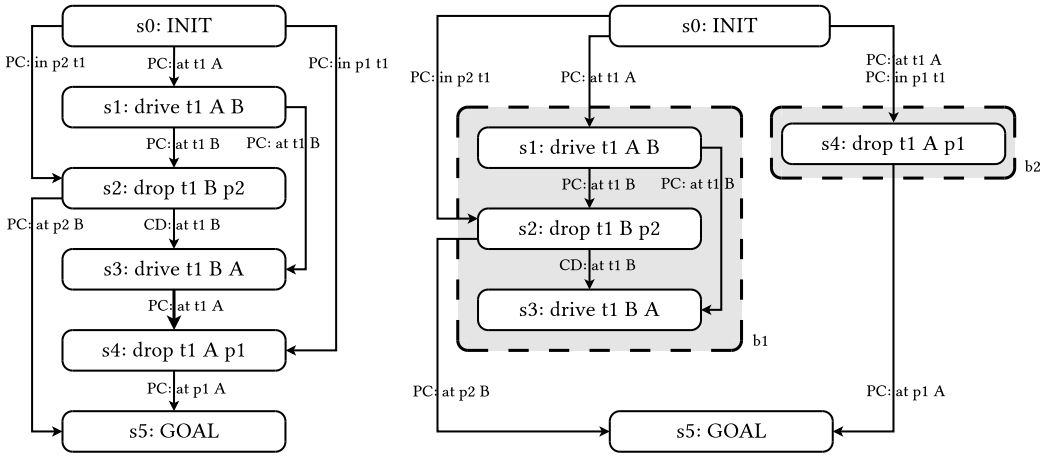

Figure 6: A sequential plan and a block deordering of this plan with two unordered blocks b1 and b2. Ordering constraints are labelled with their reasons: producer–consumer (PC), i.e., causal link, deleter–producer (DP), and consumer–deleter (CD). Note that no ordering constraint in the sequential plan can be removed without invalidating it. Thus, step-wise deordering of this plan is not possible.

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 6(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 $\langle \text{s1}, \text{s2}, \text{s3}, \text{s4} \rangle$ and $\langle \text{s4}, \text{s1}, \text{s2}, \text{s3} \rangle$. 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), 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, which it did not have with this ordering, and that delete effect causes a causal 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 can not be removed. This is demonstrated in Figure 7, 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 $b \in \mathcal{B}$; $s_t, s' \in b$; $s_p, s_c \notin b$; $m \in \text{add}(s')$; $m \notin \text{del}(b)$; and $s_t \prec^+ s'$. $s_t \prec^+ s'$ is a* threat protection ordering *if breaking this ordering causes $m \in \text{del}(b)$ and that causes $\langle s_p, m, s_c \rangle$ to become unprotected from s_t.*

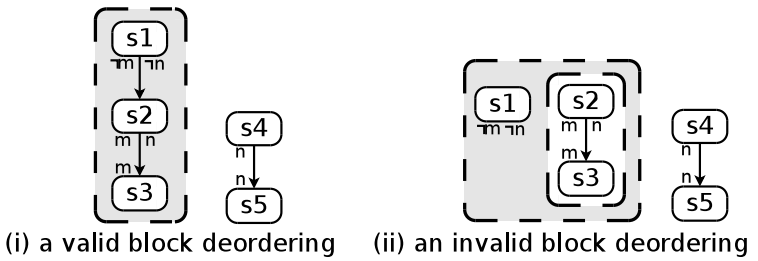

Figure 7: Two block decompositions of a plan containing five steps: s1, s2, s3, s4, and s5. In decomposition (i), there are three (transitively reduced) necessary orderings: s1 $\prec$ s2, s2 $\prec$ s3, and s4 $\prec$ s5, where Re(s1 $\prec$ s2) = {DP(m), DP(n)}, Re(s2 $\prec$ s3) = {PC(m)}, and Re(s4 $\prec$ 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 $del(b_x)$ = {m, 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 in (i) 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.

The notion of threat protection ordering was missing from our earlier block deordering procedure (Siddiqui & 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 subsection, 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.*

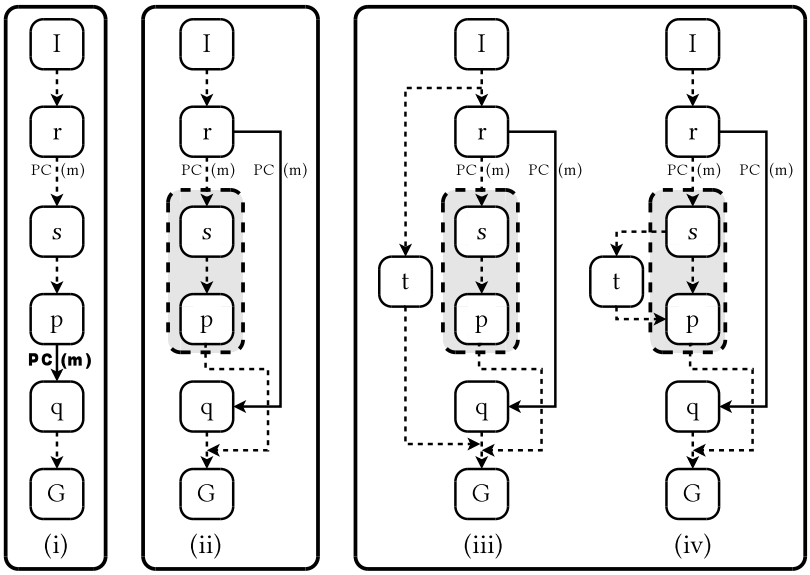

Figure 8: Formation of a block {s,p} and addition of a causal link ⟨r,m,q⟩ in (ii) in order to remove the reason PC(m) behind the basic ordering constraint p ≺ q from (i). Different situations, (iii and iv), where a threat, t, may be active to ⟨r,m,q⟩.

As an explanation of Rule 1, if $\mathrm{PC}(m) \in \mathrm{Re}(s_i \prec s_j)$, 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 $\mathrm{PC}(m)$ to $\mathrm{Re}(s_p \prec s_j)$ (i.e., adding $\langle s_p, s_j \rangle$ to $\prec$ if not already present) allows $\mathrm{PC}(m)$ to be removed from $\mathrm{Re}(s_i \prec s_j)$.

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

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

Assume PC(m) ∈ Re(p ≺ q) is removed, and the precondition of q is now supplied by the step r based on the newly established causal link ⟨r,m,q⟩ after deordering and formulating b_p = {s,...,p}, b_q = {q} in π'_{bdp}, as shown in Figure 8 (ii). We have to show that ⟨r,m,q⟩ has no active threat in π'_{bdp}, and therefore, π'_{bdp} is valid. Assume, there is an active threat, t, to ⟨r,m,q⟩ 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 ⟨r,m,q⟩.

Situation (1): assume s $\not\prec^+$ t, as shown in Figure 8 (iii). Since t is not an active threat to ⟨r,m,s⟩ in π_{bdp}, then according to Theorem 2, either t is contained by a block that does

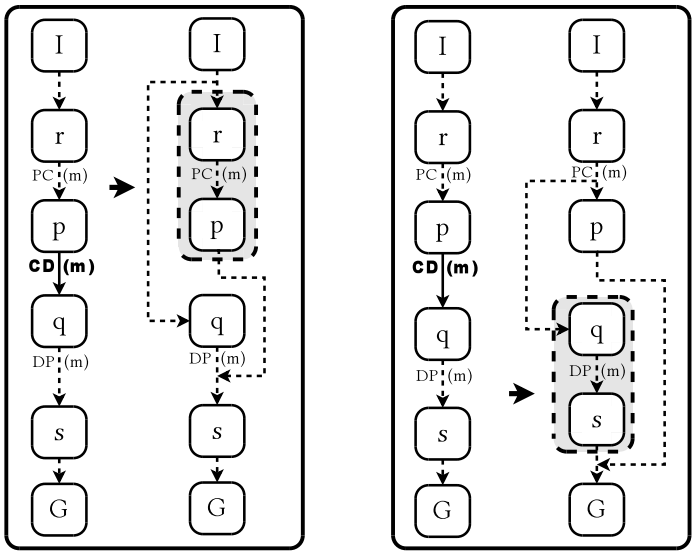

Figure 9: Formation of blocks for removing the reason CD(m) behind the basic ordering p $\prec$ q.

not delete the threatened atom and does not contain ⟨r,m,s⟩, or ⟨r,m,s⟩ 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 can not be an active threat to ⟨r,m,q⟩. For the second case, b' can not 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 can not 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' can not partially overlap with b_p = {s,...,p}. Since t is also not an active threat to ⟨p,m,q⟩ in π_{bdp}, ⟨p,m,q⟩ must be contained by some block $b'' = \{p, q, ...\}$ that does not contain t. Now, since b' and b'' can not partially overlap, b' or b'' (whichever is bigger) must contain at least r, s, p, and q, for which b' or b'' (whichever is bigger) protects ⟨r,m,q⟩ from t.

Situation (2): assume t $\not\prec^+$ p, also shown in Figure 8 (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 8 (iv). This is only possible if $t \in b_p$, since t can not 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) $\in$

$\mathrm{Re}(s_i \prec s_j)$. *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_i *does not consume* m.

Theorem 4. *Deordering according to* Rule 2 *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 $\mathrm{p}, \mathrm{q} \in \mathcal{S}$ with $\mathrm{CD}(m) \in \mathrm{Re}(\mathrm{p} \prec \mathrm{q})$. In order to meet the condition of Rule 2, let us assume b_p is a block that includes r and p such that $\langle$r,m,p$\rangle$ 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 9 (i)). Therefore it meets the condition that b_p must not consume m. Also, assume b_q is a block that contains {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 $\langle$r,m,p$\rangle$ in π'_{bdp}. However, $\langle$r,m,p$\rangle$ is contained by b_p that does not contain q, and therefore, according to Definition 6, $\langle$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 $\mathrm{p}, \mathrm{q} \in \mathcal{S}$ with $\mathrm{CD}(m) \in \mathrm{Re}(\mathrm{p} \prec \mathrm{q})$. In order to meet the condition of Rule 3, let us assume b_q is a block that includes q and s such that $\mathrm{DP}(m) \in \mathrm{Re}(\mathrm{q} \prec \mathrm{s})$ and every other deleter of m in b_q (if they exist) is ordered before s in π_{bdp} (as shown in Figure 9 (ii)). Therefore it meets the condition that b_q must not delete m. Also, assume b_p is a block that contains {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 $\langle$r,m,p$\rangle$ in π'_{bdp}. However, q is contained by b_q that does not contain $\langle$r,m,p$\rangle$, and does not delete m; therefore, according to Definition 6, $\langle$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.*

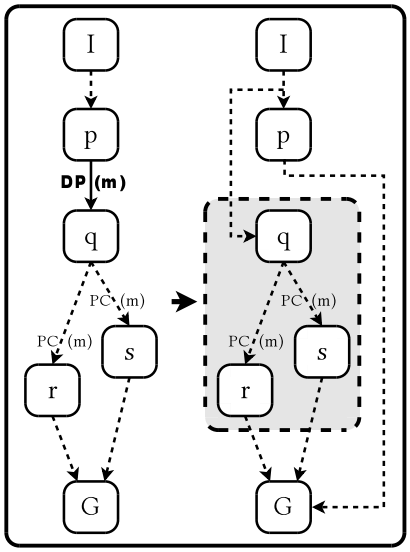

Figure 10: Formation of blocks for removing the reason DP(m) behind the basic ordering p $\prec$ q.

Proof. Let $\pi_{\text{bdp}} = \langle \mathcal{S}, \mathcal{B}, \prec \rangle$ be a valid block decomposed p.o. plan, and let p $\prec$ q be a basic ordering constraint (where $\text{p}, \text{q} \in \mathcal{S}$). Let b_q be a block that includes all the steps – r and s such that ⟨q,m,r⟩, ⟨q,m,s⟩ are the causal links in π_{bdp} (as shown in Figure 9 (ii)). Hence, it meets the condition of Rule 4. Also, assume b_p is a block that contains {p} and b_p, b_q are not ordered for any other 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.

2.6 Block Deordering Algorithm

The previous subsection described four conditions (Rules 1–4) under which adding blocks to a decomposition allows reasons for ordering constraints, and thus ultimately 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's (1994) 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: 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 $b_j \not\subset b_i$). 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 subsection. 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 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

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 $b'_i = b'_i \cup \{s'\} \cup$ 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 $b'_j = b'_j \cup \{s'\} \cup$ 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 $b'_j = b'_j \cup \{s'\} \cup$ 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 $b'_i = b'_i \cup \{s'\} \cup$ INTERMEDIATE(s', b'_i).
          else return NULL.
    Recompute $\mathrm{Re}(b'_i \prec b'_j)$.
  return (b'_i, b'_j).

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. Each iteration runs in polynomial time, but we do not know of an upper bound on the number of iterations. Note, however, that our procedure is anytime, in the sense that if interrupted before running to completion, the result at the end of the last completed iteration is still a block deordering of the plan. In BDPO2, we use a time-limit of 5 minutes for the whole deordering procedure. However, for almost every problem considered in our experiments (described in Section 3.1), block deordering finishes in a few seconds (except for a few problems in the Visitall domain, for which it takes a couple of minutes).

In 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 deordering plans to a greater extent, including in cases where no deordering is possible using the standard, step-wise, partial order plan notion. The impact of block decomposition on the anytime performance of our plan quality optimisation system is discussed in Section 3.6.

3. System Overview

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, it starts over from the new plan. This can be viewed as a local search, using the large neighborhood search (LNS) 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 the previous section, 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, Nau, & Traverso, 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 3.3. We return to the question of when and how multiple windows within the same plan can be simultaneously replaced in Section 3.5.

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 3.4.

Algorithm 2 The neighbourhood exploration procedure in BPDO2.

```
1:  procedure BDPO2(π_in, t_limit, banditPolicy, rankPolicy, optSubprob)
2:      Initialize: t_elapsed = 0, π_last = π_in, trialLimit[1...n] = 1, windowDB = ∅
3:      π_bdp = BlockDeorder(π_in)
4:      while t_elapse < t_limit and π_last is not locally optimal do
5:          if more windows needed then
6:              ExtractMoreWindows(π_bdp, windowDB, optSubprob)
7:          p = SelectPlanner(banditPolicy)
8:          w = SelectWindow(p, rankPolicy, trialLimit, 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 = OptimiseWindow(p, w)
12:         UpdateWindowDB(p, w, w_new, optSubprob, searchResult, windowDB)
13:         if C(w_new) < C(w) then
14:             π_new = Merge(π_bdp, windowDB)
15:             if C(π_new) < C(π_last) then π_last = π_new
16:         UpdateBanditPolicy(p, w, w_new, searchResult, banditPolicy)
17:         UpdateRankPolicy(p, searchResult, rankPolicy)
18:         if C(π_last) < C(π_in) and restart condition is true then
19:             return BDPO2 (π_last, t_limt − t_elapsed, banditPolicy, rankPolicy, optSubprob)
20:     return π_last
```

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 with each subplanner and the result), and structural summary information about the 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 Section 4. We also compute a lower bound on the cost of any replacement plan for the window, using the admissible LM-Cut heuristic (Helmert & 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 4.6. 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 3.5.

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.

The remainder of this section is organised as follows: The next two sections describe the settings that we have used for our experiments and an overview of main results, respectively. We then describe the subplanners used in BDPO2 (Section 3.3), the restart conditions (Section 3.4) and the Merge procedure (Section 3.5). Section 3.6 discusses the impact of block deordering on the performance of the system. The windowing strategies and ranking policies are described in Section 4, while more details of the on-line adaptation methods used are presented in Section 5.

3.1 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 & Grastien, 2011). The plans used as input to BDPO2 are the plan produced by IBaCoP2 (Cenamor, de la Rosa, & Fernández, 2014) in the 2014 IPC for the problems from that competition, and the best plan found by LAMA (Richter & 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. (In some domains less than 10 instances are solved by LAMA/IBaCoP2, which is why the total is 182 rather than 210.) 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.

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 & 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 its results here, but will refer to it later when we discuss these system components in detail.

In experiment 2, we compare BDPO2 and eight other anytime planners and plan optimisation systems: LAMA (Richter & Westphal, 2010, IPC 2011 version); AEES (implemented in the Fast Downward code base; cf. Thayer et al., 2012b); IBCS (as described in Section 3.3); Beam-Stack Search (BSS) (Zhou & Hansen, 2005); PNGS, including "Action Elimination" (Nakhost & Müller, 2010); IBaCoP2 (Cenamor et al., 2014); LPG (Gerevini & Serina, 2002); and Arvand (Nakhost & Müller, 2009). BDPO2 uses PNGS and IBCS as subplanners, and is configured as described above. AEES uses LM-Cut (Helmert & Domshlak, 2009) as its admissible heuristic, and the FF heuristic, with and without action costs for its inadmissible estimates. BSS uses the LM-Cut heuristic. Our implementation of BSS does not use divide-and-conquer solution reconstruction, and was run with a beam width of 500. The other systems are described further in Section 6.

Each system was run for up to 7 hours 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 is much slower than the other planners used in the experiment. 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 experiment to the other systems, we add 2 hours to its runtime.

3.2 Overview of Results

Figure 11 shows a 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. The results in Figure 11 are from experiment 2 and 3, described in the previous section. It is the same as shown in Figure 2 (on page 371), 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 11 is the average over only those problems for which they have, at that time, found 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.) In other words, this metric is unaffected by the differences in coverage. Likewise, 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 14. For these systems, the average quality shown in Figure 11 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 some algorithms, notably PNGS, which exhausts the 8 GB available memory before reaching the 7 hour CPU time limit for 93.7% of problems, and LAMA, which does the same for 67% of problems. AEES runs out of memory on just over 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.

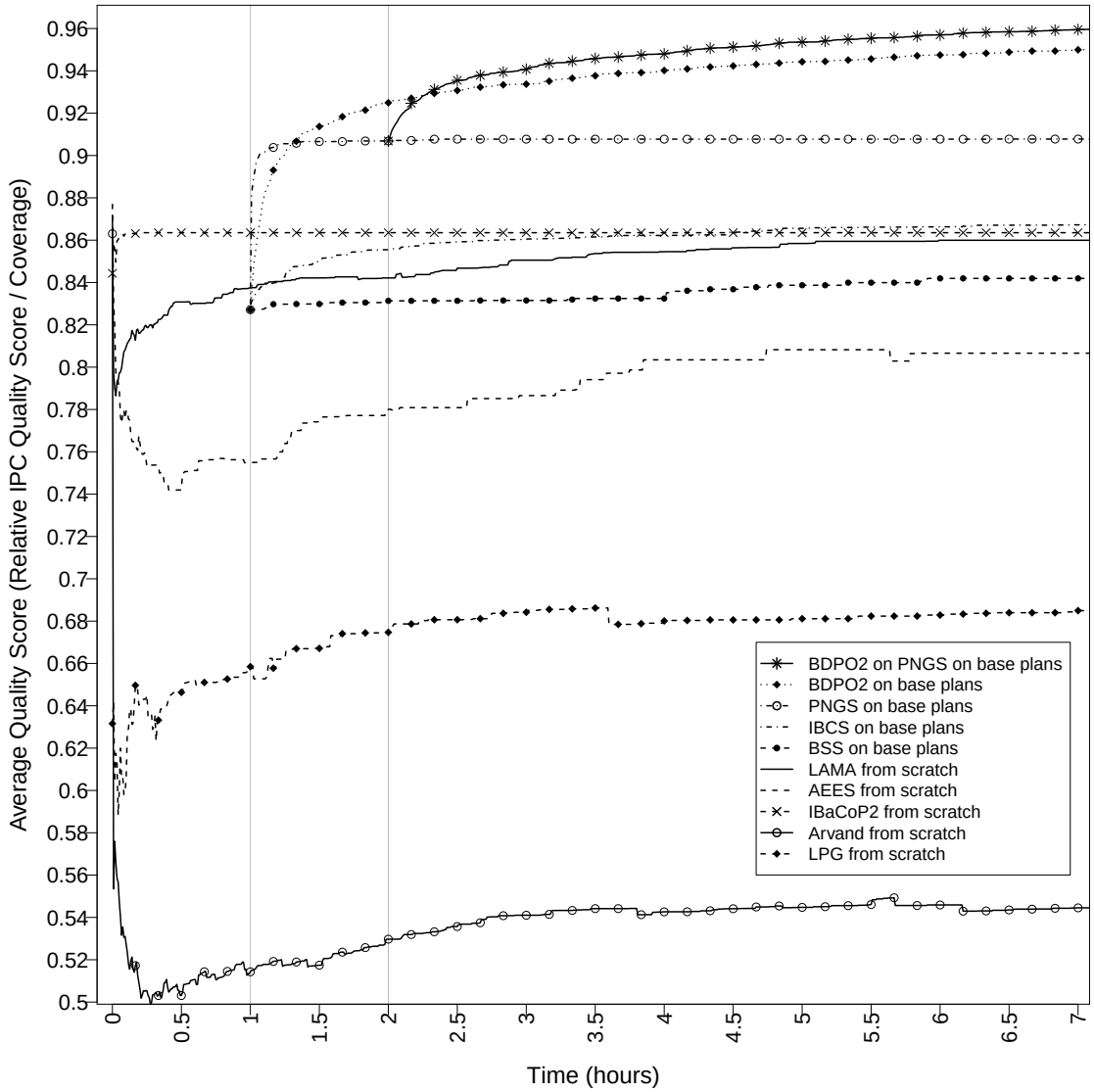

Figure 11: Average IPC quality score as a function of time per problem, on a set of 182 large-scale planning problems. The quality score of a plan is c^{ref}/c, where c is the cost of the plan and c^{ref} is the least cost of all plans for the problem; hence a higher score represents better plan quality. 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 3.1.

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 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 12. (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 3.1).

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 can be seen in Figure 12, which provides a more detailed view. 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 12: 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.)

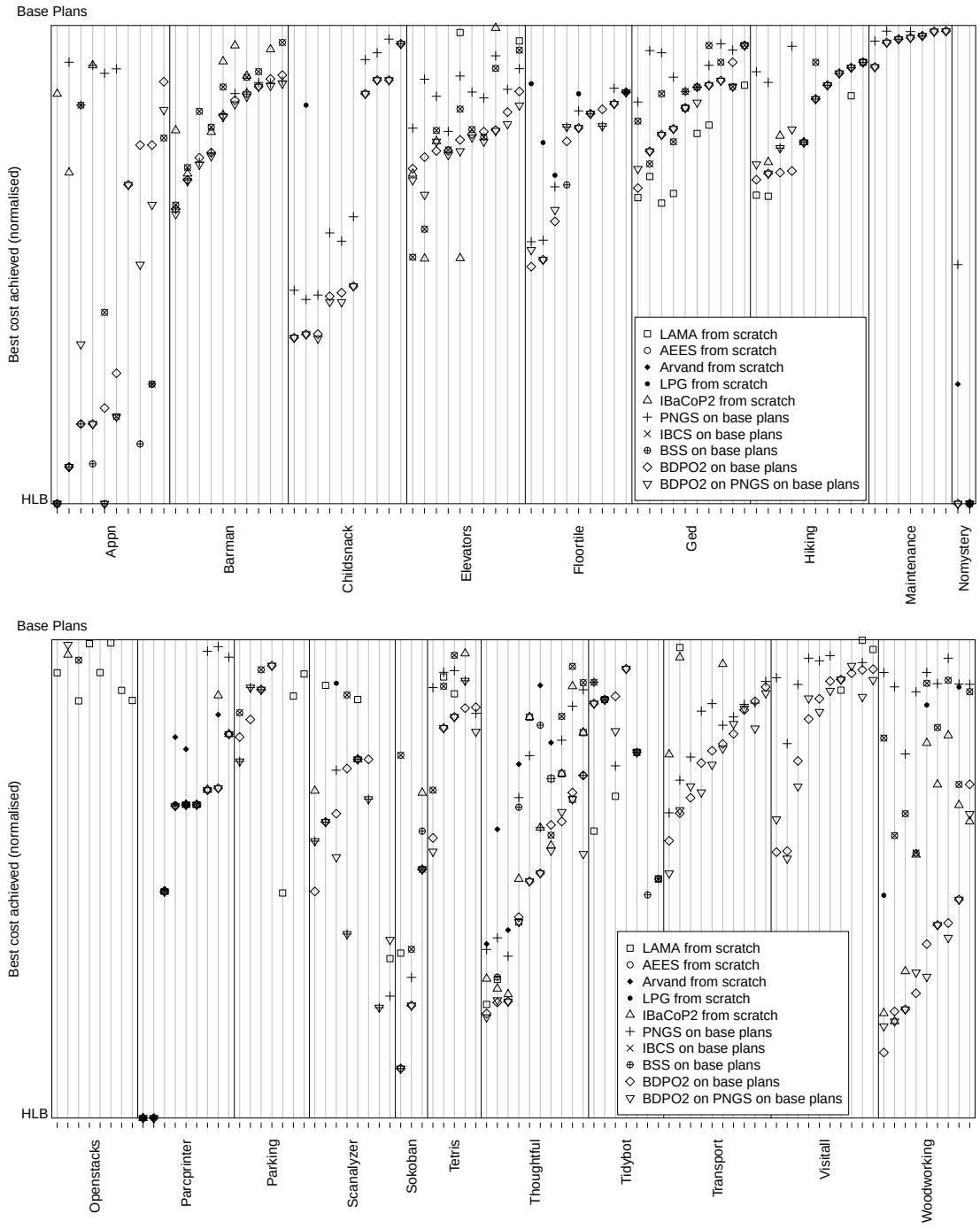

Figure 12: Best plan cost, normalised to the interval between the cost of the base plan and the corresponding highest known lower bound, achieved by different anytime plan optimisation methods in experiment 2, and by BDPO2 in experiments 2 & 3 (see Section 3.1).

Table 1 provides a different summary of the information in Figure 12, 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 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 well in these domains.

3.3 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 & 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 Section 5.2 on page 420.

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 & 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 & 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 & 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.

3.4 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 merging together several subplan improvements, as described in Section 3.5 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 (cf. description of Algorithm 2 at the beginning of this section). This means that after 39 attempts we have tried to optimise the 13 most promising 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 13 (page 403). In this experiment, both configurations were run using

Algorithm 3 Merge Improved Windows

```
1: procedure Merge(π̂_bdp, windowDB)
2:     Initialise π̂_bdp = π_bdp
3:     W = improved windows from windowDB sorted by cost reduction (C(w) − C(w_new))
4:     while W ≠ ∅ do
5:         (⟨p, w, q⟩, w_new) = pop window with highest C(w) − C(w_new) from W
6:         π̂_bdp = ReplaceIfPossible(π̂_bdp, ⟨p, w, q⟩, w_new)
7:         W = RemoveConflictingWindows(W, π̂_bdp)
8:     return π̂_bdp
```

the same setup as experiment 3, described in Section 3.1 on page 392. As can be seen, delayed restart yields better results overall. Compared to BDPO2 with immediate restart, it achieves a total improvement that is 12% higher. 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 13 shows that the additional time spent in each neighbourhood when using delayed restarts pays off.

3.5 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 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 378). 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. □

3.6 The Impact of Plan Decomposition

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 & Pohl, 1986; Estrem & Krebsbach, 2012; Balyo, Barták, & Surynek, 2012). In this section, we address the question of how much the block deordering contributes to the performance of BDPO2.

In the preliminary experiment (setup 1, as described in Section 3.1 on page 392) 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 3.1). In this experiment, we ran BDPO2 separately with different degrees of plan decomposition: (1) With block deordering (as in the default BDPO2 configuration, the one used in experiments 2 and 3 presented in Section 3.2 on page 394). (2) With standard, i.e., step-wise, plan deordering only. In this configuration, we used Kambhampati and Kedar's (1994) algorithm (described in Section 2.3) 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 3.4.

Figure 13 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 block and step-wise deordering, delayed restarting further boosts the performance of LNS

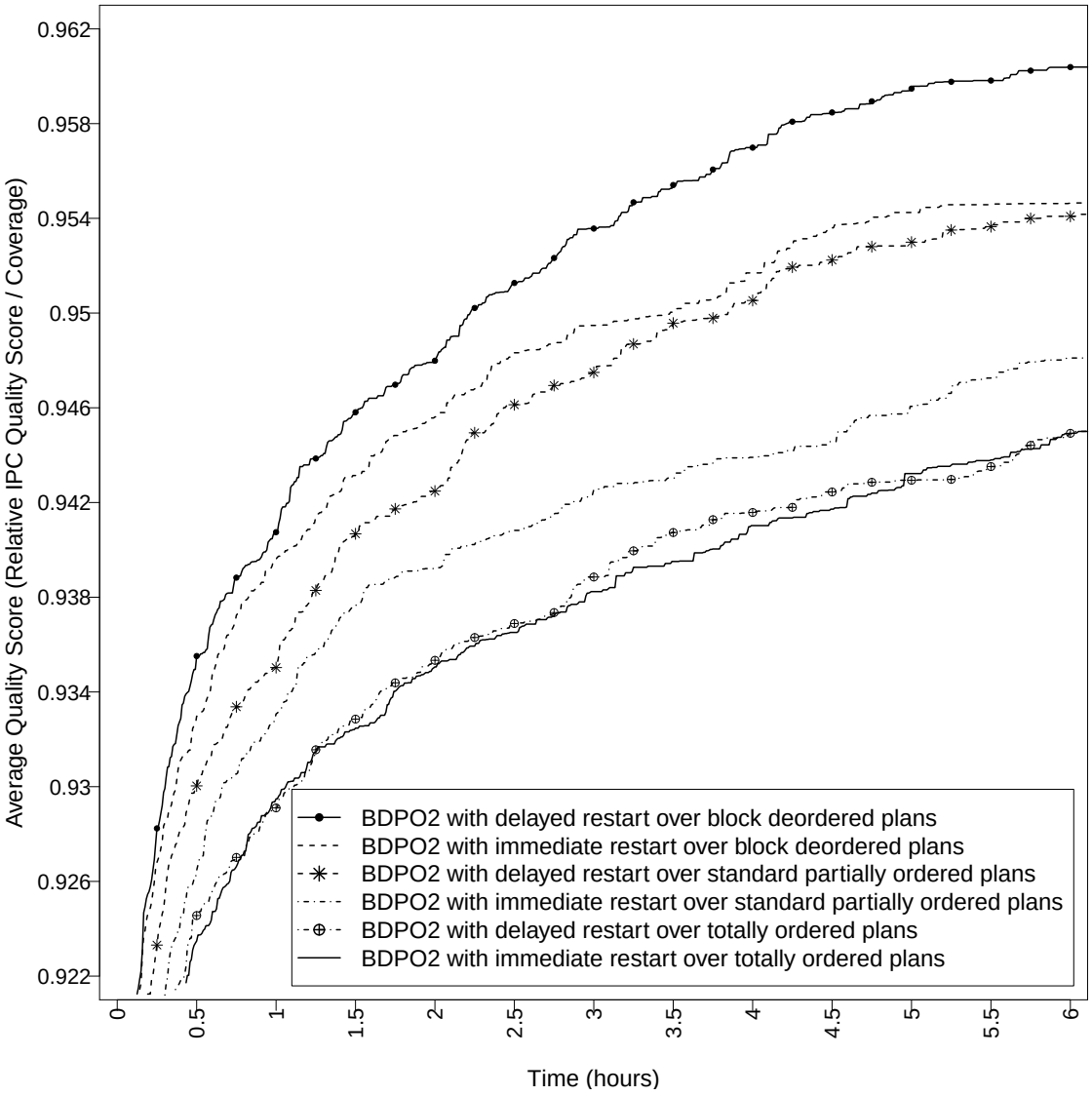

Figure 13: Average IPC quality score as a function of time per problem for BDPO2 applied to the totally ordered input plan; the standard (step-wise) deordering of the plan; and the block deordering of the plan. For each plan type, the system was run in two configurations: once with delayed restarting and once with immediate restarting (cf. Section 3.4 on page 399). This experiment was run with setup 3, as described in Section 3.1 on page 392. 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 11). Note also that the y-axis is truncated: All curves start from the average quality of the input plans, which is 0.907.

plan optimisation by 12% and 14.7%, respectively, 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 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 in a few cases more than 10. 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 3.2, in some domains subplanners need more runtime to find better plans for improvable windows, and the BDPO2 configuration without deordering does find a better plan than the default configuration for 26 of the 182 problems. 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 4.6) 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.

4. Windowing Strategies and Ranking of Windows

A window is a subplan of some linearisation of the block deordered plan, extracted in order to attempt local optimisation. This section describes the strategies we use to generate and rank windows, and an experimental evaluation of their impact on our system's performance.

Recall from Definition 8 (page 390) 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 many possible windows – typically far too many to attempt to optimise them all. A *windowing heuristic* is a procedure that extracts a reduced set of windows, hopefully including the most promising ones, in a systematic way. We

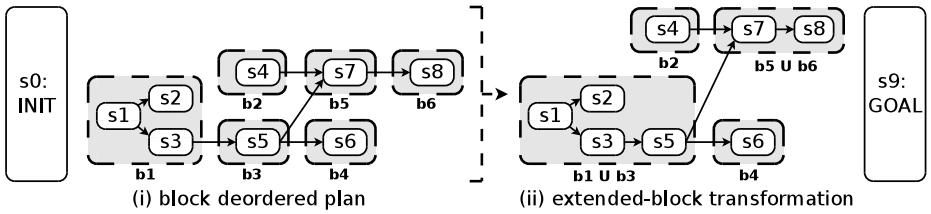

Figure 14: A block deordered plan and its transformation into extended blocks: blocks b1 and b3 are merged into a single block, as are blocks b5 and b6.

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 2: The total number (in thousands) of windows that were generated, not filtered out, and finally improved, using different windowing heuristics over different block types (basic and extended). The number of possible windows over the sequential input plans, before even considering deordering, is over 1.47 million. The rightmost pair of columns shows the rate of success, meaning the fraction of improved windows out of the generated windows. The numbers are from the results of experiment 1 (described in Section 3.1 on page 392).

present 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 heuristic is applied over two types of block – *basic* and *extended* – one at a time. Basic blocks 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 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 procedure for extended block formation. (IP(b) denotes the set b's immediate predecessors, while IS(b) is b's immediate successors.)

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: $\quad \mathrm{B_{ext}} \leftarrow \mathrm{B_{ext}} \cup \{b_i \cdot b_j\} \setminus \{b_i, b_j\}$

The process is further illustrated by an example in Figure 14. 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. Our experiment results show that windows of different sizes are more useful in different domains:

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)
```

For example, larger windows 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 4.4, and in particular Table 3 on page 411). Thus, all of them are, in this sense, useful. On 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 2 shows results from our first experiment, in which we systematically tried two subplanners (PNGS and IBCS) on every window generated (and not filtered out) by any windowing strategy over 219 input plans. The table shows the total number (in thousands) of windows that were generated, that remain after filtering, and that were finally improved by at least one of the two subplanners. In this experiment, windows were filtered out only if the window cost matched the lower bound given by the admissible LM-Cut heuristic (Helmert & Domshlak, 2009). The experiment setup is further described in Section 3.1 (on page 392). The first observation is that all strategies are very selective. The number of windows that could potentially be generated, even without considering deordering, i.e., only by taking all subsequences of the totally ordered input plans, is over 1.47 million. Thus, even the most prolific strategy generates less than a tenth of the possible windows. Second, we used the rate of success, meaning the fraction of windows generated that were improved by any of the subplanners used in the experiment, to order the strategies. 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 391) 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 cost matches the lower bound given by the admissible LM-Cut heuristic (Helmert & 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. Section 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 $|W| = \text{nWindowsLimit}$ or time is up. If no windows are generated, and some strategies are still not exhausted, the time limit is increased.

4.1 Rule-Based Windowing Heuristic

Our first version of BDPO (Siddiqui & Haslum, 2013b) used a single windowing strategy, based on applying a fixed set of rules over extended blocks. Because this strategy complements the new windowing heuristics well, we have kept it in BDPO2.

Each rule when applied to a block b in a block deordered plan π_{bdp} selects a set of blocks from to go into the replaced part (w) based on their relation to b. To ensure that the window is consistent with the block deordering (i.e., has a consistent linearisation, as stated in Definition 8 on page 390), any blocks that are constrained to be ordered between blocks in the window must also be included. We call these the *intermediate blocks*, 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. $w' \leftarrow \{b\} \cup \text{Un}(b) \cup \text{IP}(b)$.
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. $w' \leftarrow \{b\} \cup \text{Un}(b) \cup \text{IS}(\{b\} \cup \text{Un}(b))$.

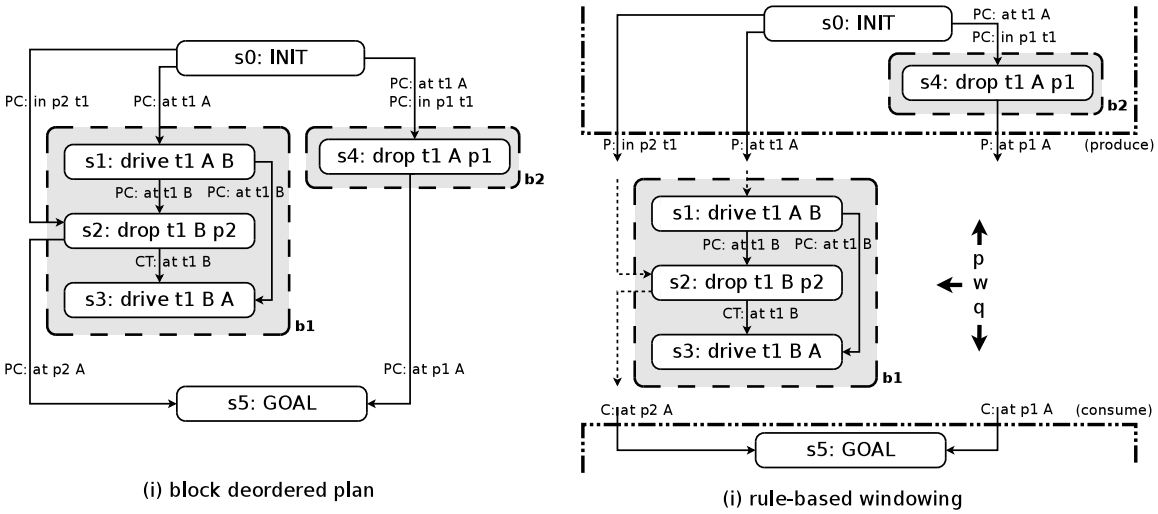

Figure 15: Window formation by applying the 1st rule of the rule-based windowing heuristic over block b1, i.e., $w \leftarrow \{\text{b1}\}, p \leftarrow \text{Un(b1)}$. The unordered block b2 is placed in its predecessor set. Note that this window can be optimised by removing s3 because this step has no causal link to its successors.

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

Given the blocks selected by one of the rules above, the partitioning of blocks into $\langle p, w, q\rangle$ is made 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 15 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 first rules, which include fewer blocks, generally produce smaller windows, while the later rules tend to produce larger window (though there is no exact relation, since the number of actions in a block varies). The heuristic applies all rules to each block in the block deordered plan π_{bdp} in turn. Rules are applied in the order 1,10,2,9,3,8,4,7,5,6, i.e., starting with the first, the last, the second, the second last, and so on. The blocks are ordered by size (descending), with ties broken by their order in the input plan (in opposite direction for extended blocks).

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.

4.2 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 im-

proved. We call this *cyclic behavior*. In one experiment, we found 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 $m \in I$. However, since a window never contains the initial step s_{I}, candidate windows are formed from extended producers instead. A step $s \notin \{s_{\mathrm{I}}, s_{\mathrm{G}}\}$ is an extended producer of atom m iff s produces m, or s consumes m and there is no $s' \neq s_{\mathrm{I}}$ that produces m and ordered before s in the block deordered plan. The formal definition 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 (cf. Definition 14) is then formed by taking a linearisation of those blocks, consistent with the input plan.

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 $s_a \prec_{\mathrm{in}} s_b \prec_{\mathrm{in}} s_c \prec_{\mathrm{in}} s_d$. The linearisation starts with the block that contains the first element of $\prec_{\mathrm{in}}$, i.e., b_x in this case (since it contains s_a); $\prec_{\mathrm{in}}$ is then updated to $\prec_{\mathrm{in}} \setminus b_x$, and the linearisation continues in the same fashion until $\prec_{\mathrm{in}}$ 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 $\prec_{\mathrm{in}}$, 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.*

For example, in the plan shown in Figure 15(i), atom (at t1 A) has cyclic behaviour, since it holds in the initial state and is added by step s3. Its extended producers are s1, s3 and s4, so the cyclic thread is $T_{(\text{at t1 A})} = \langle \text{b1}, \text{b2} \rangle$.

Finally, candidate windows are formed by taking a consecutive subsequence of blocks (and intermediate blocks, as necessary) 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.

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. 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.

4.3 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 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 15(i) appears in two causal links, both with the same producer: $\langle \mathrm{s1}, (\text{at t1 B}), \mathrm{s2} \rangle$ and $\langle \mathrm{s1}, (\text{at t1 B}), \mathrm{s3} \rangle$. Thus, the causal followers are $\mathrm{CF}_{(\text{at t1 B})} = \langle \{\mathrm{s1}, \mathrm{s2}, \mathrm{s3}\} \rangle$.

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 sequence of causal follower blocks of $\mathrm{CF}_{(\text{at t1 B})}$ in the plan in Figure 15(i) is $\mathrm{CFB}_{(\text{at t1 B})} = \langle \{\mathrm{b1}\} \rangle$, since all steps in $\mathrm{CF}_{(\text{at t1 B})}$ are contained in block b1.

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*

	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 3: 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 3.1.

(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.

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. 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.

4.4 The Impact of Windowing Heuristics

No one single windowing heuristic or block type, nor any combination of them, is guaranteed to find all improvable windows. The first row of Table 3 shows the percentage of improvable windows found using one block type but not the other (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. (The results are from the first experiment, described in Section 3.1). It 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). On the other hand, 36.78% of all improvable windows are not found by this heuristic. Each of the windowing heuristics has its strengths and limitations. The rule-based heuristic, for example, can only generate windows that contain sequences of extended blocks up to a fixed length, while the cyclic thread and causal followers heuristics only make windows from blocks connected by a single atom.

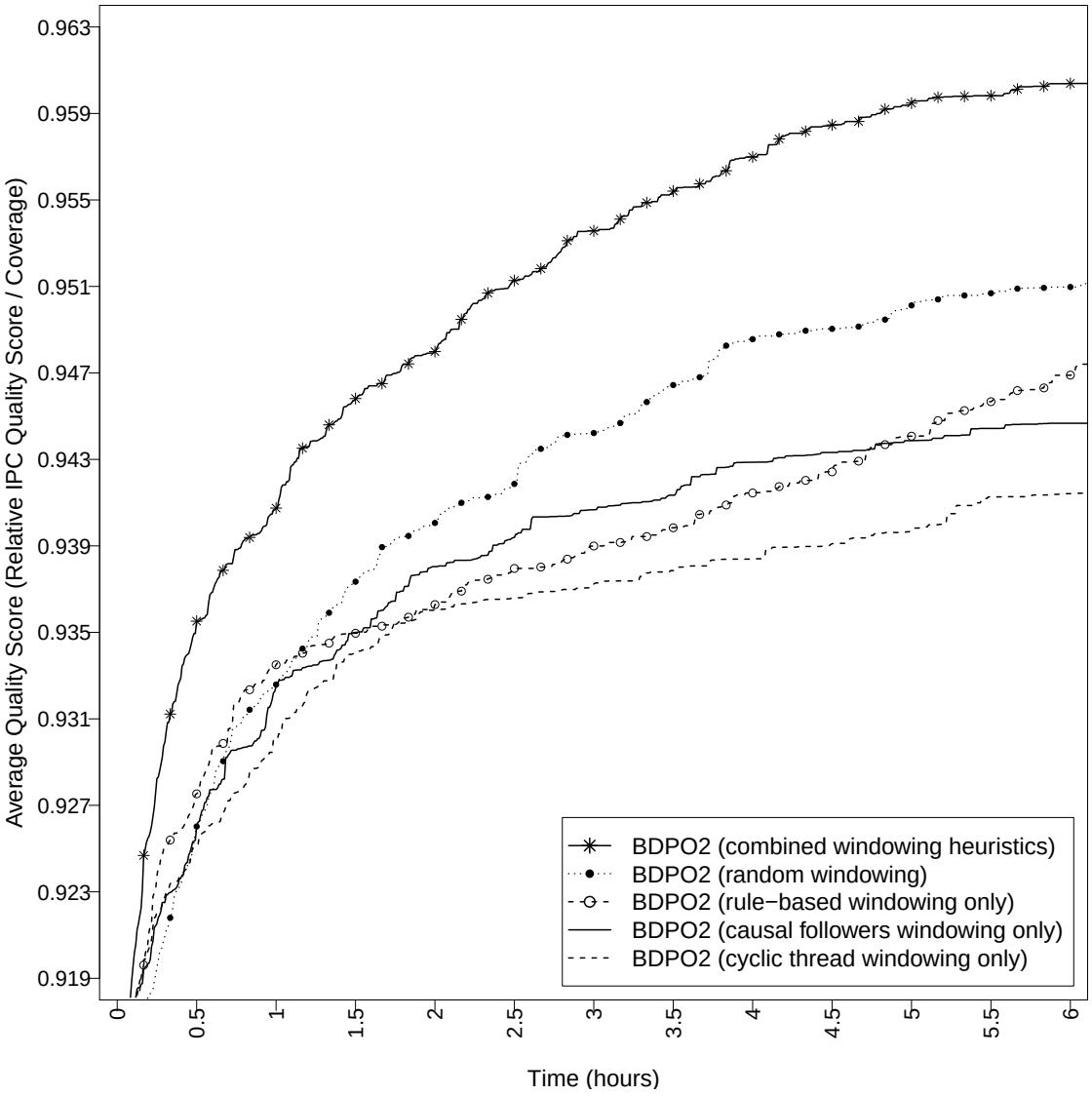

Figure 16: Average IPC quality score as a function of time for separate runs of BDPO2 using each of the three windowing heuristics alone, all three heuristics combined, and random window generation. Each run is done using the same setup as experiment 3, described in Section 3.1 (on page 392). The x-axis here shows only the runtime of BDPO2 (i.e., without the 2 hour delay for generating the input plans, as shown in Figure 11). Note also that the y-axis is truncated: The average quality of the input plans is 0.907.

Figure 16 shows the impact of different windowing heuristics on the anytime performance of BDPO2, as measured by the average IPC plan quality score achieved as a function of time-per-problem. In this experiment, we ran BDPO2 with each of the three windowing heuristics alone, and with all three combined in a sequential portfolio, as described at the beginning of this section. (The combined portfolio of windowing heuristic is the same configuration of BDPO2 that is presented in the experimental results in Section 3.2, page 394.) We also compare these with a non-heuristic, random windowing strategy, in which each window is formed by taking a random subsequence of blocks from a random linearisation of the block deordered plan. Subsequences are chosen so that the distribution of window sizes (measured by the number of actions in the window) is roughly the same as that produced by the combined heuristics. The experiment uses setup 3 (described in Section 3.1 on page 392), i.e., the input plans to BDPO2 are already of high quality. (Their average IPC plan quality score is 0.907.)

As predicted by the data in Table 3, using any of the three windowing heuristics on its own results in a much worse system performance, since each fails to find a substantial fraction of improvable windows. In fact, random window generation is better than any of the heuristics on their own. However, the combined portfolio of heuristics outperforms random windowing by a good margin: the total quality improvement achieved with the random windowing strategy is 17.1% less than that of the best BDPO2 configuration. This demonstrates that the heuristics capture information that is useful to guide the selection of windows.

4.5 Possible Extensions to the Windowing Strategies

Since a window is formed by partitioning plan steps into three disjoint sets of blocks, the number of possible windows is exponential. The challenge for a good windowing heuristic is to extract a reduced set that contains windows more likely to be improved. Every windowing strategy has some limitations. 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 combination of strategies we use may miss some improvable windows. For example, a long sequence of blocks that do not form part of a cyclic thread or causal followers sequence with respect to a single atom will not be captured by these heuristics. An example of this is shown in Figure 17, where three candidate windows, W1, W2 and W3, found by the causal followers windowing heuristic are not improvable separately. In this situation, forming a window as the union of separate windows, found by one or several strategies, can overcome the limitations of those strategies. In the example, the union of W1 and W2 is improvable. This type of composite windows could be formed in the later stages of the plan improvement process, after all the individual windowing heuristics have been exhausted. However, the number of composite windows that can be created from a large set of candidate windows is combinatorial and thus optimising all of them will take a long time.

4.6 Window Ranking

Although the windowing strategies generate only a fraction of all possible windows, the number of candidate windows is still often large (cf. Table 2). In order to speed up the

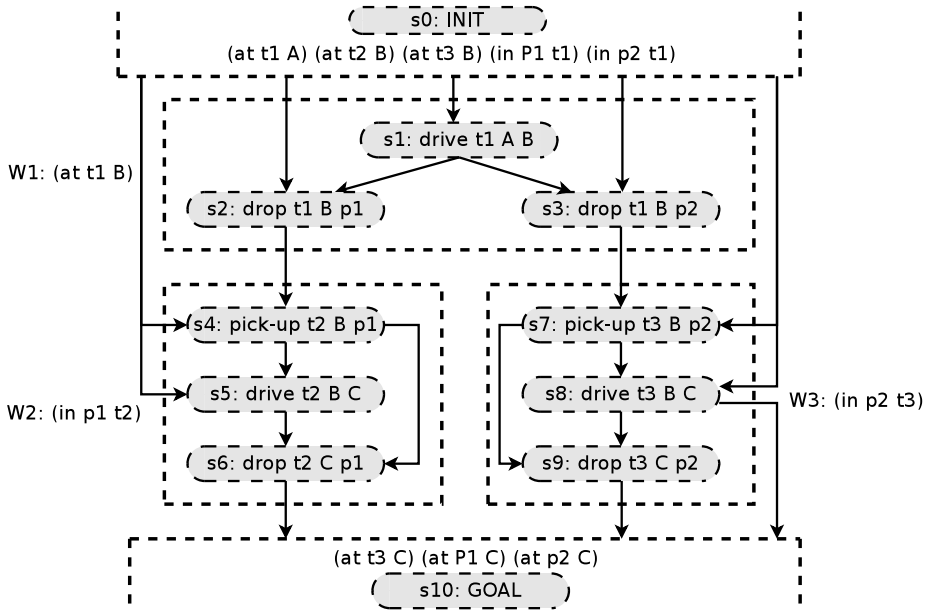

Figure 17: Three candidate windows, W1, W2, and W3, found by the causal followers windowing heuristic for atoms (at t1 B), (in p1 t2), and (in p2 t3) respectively. None of them are improvable. However, the composite window formed by merging W1 and W2 is improvable by substituting the delivery of package 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).

plan improvement process, it is helpful to order windows so that those more likely to be improved are optimised first. This is the role of window ranking.

Ranking windows is made difficult by the fact 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 section, 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. In the Sokoban domain, on the other hand, medium-sized windows are better. Moreover, an improvable window may not be improved by a particular subplanner within the given time bound. We have noted that in some domains, e.g., Pegsol or Scanalyzer, subplanners require, on average, more time to find a lower-cost plan.

We have developed a set of window ranking policies by examining structural properties of the generated candidate windows generated and the results of our first experiment (cf. Section 3.1) in which we ran 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 & Domshlak, 2009). Investigating the properties of improved and unimproved windows, we identified four metrics that work relatively well across domains:

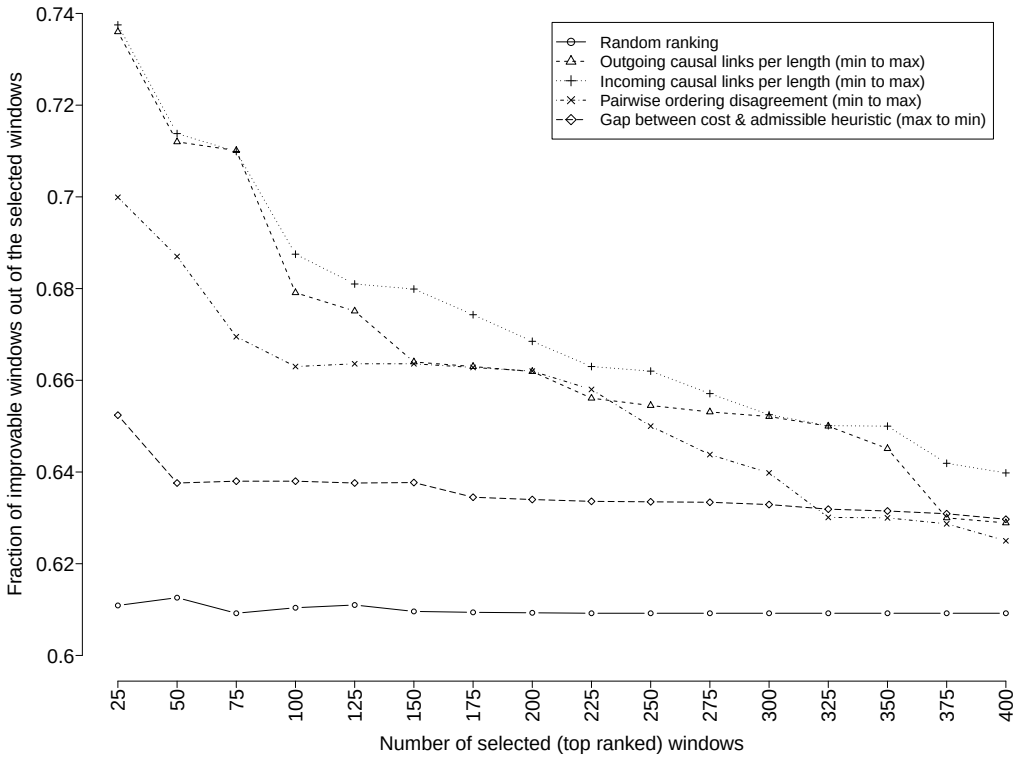

Figure 18: Fraction of improvable windows, across all domains, out of the selected top windows from the ranked orders generated by each of the ranking policies (see text).

(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 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.

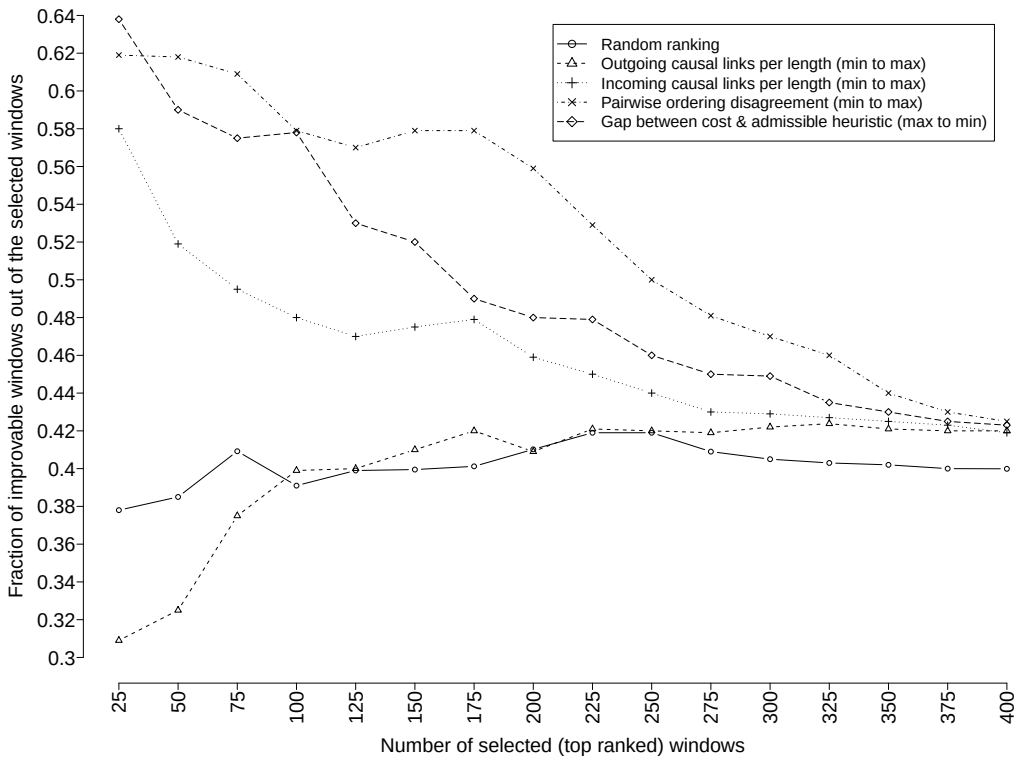

Figure 19: Fraction of improvable windows in the Parking domain, out of the selected top windows from the ranked orders generated by each of the ranking policies (see text).

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 18 compares these ranking policies with the performance of a random ordering of the windows. On average across all domains, all four ranking policies are good at picking out improvable windows. For example, if we take 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 contain only 61% improvable windows. The random ranking in Figure 18 is the best result out of three separate random rankings for each of the values on the x-axis. As expected, it exhibits roughly 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 of individual ranking policies varies by domain, and for each policy we find some domain in which it is not good. For example, Figure 19 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.

BDPO2 uses the first three ranking policies in a sequential portfolio (as explained in Section 3). For each subplanner, BDPO2 uses a current ranking policy to select the next

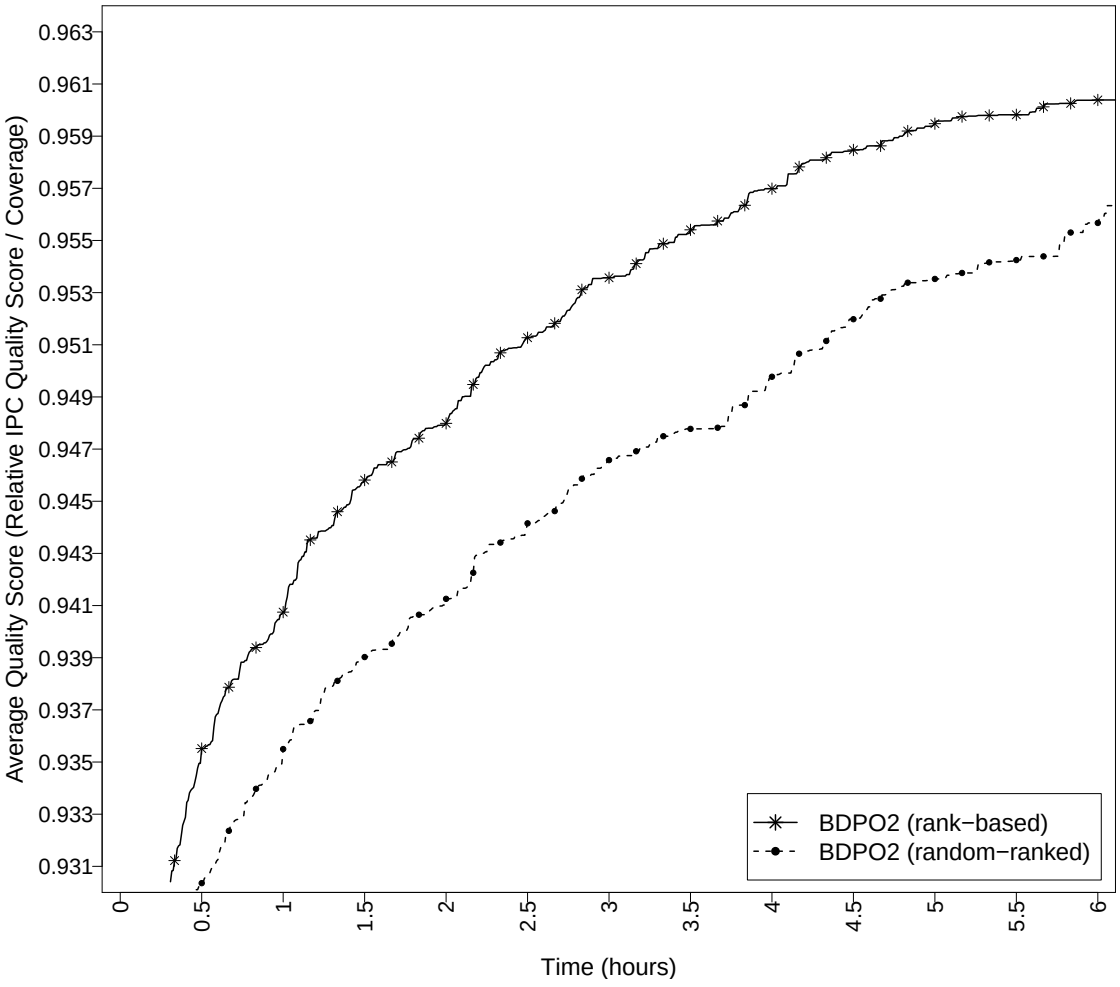

Figure 20: Average IPC plan quality score as a function of time in two separate runs: with and without window ranking. In the second case, the order of the candidate windows is randomised. Each run is done using experimental setup 3, as described in Section 3.1 on page 392. The time shown here is the runtime of BDPO2 only (excluding the 2 hour delay for generating the input plans, as shown in Figure 11). Also, the y-axis is truncated: All curves start from the average quality score of the input plans, which is 0.907.

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 the candidate windows for that subplanner.

The use of window ranking has a beneficial effect on the anytime performance of the plan improvement process, as shown in Figure 20. We achieve higher quality scores, and in particular, achieve them faster, when using window ranking compared to random ranking. In this experiment, we ran BDPO2 once with the portfolio of ranking policies, as described

above, and once with windows chosen for optimisation in a random order. The experiment used the same setup as experiment 3 (described in Section 3.1 on page 392).

We tried many alternative 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, Kumar, Naor, & Sivakumar, 2001), and multivariate Spearman's rho (Bedo & 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 our choice of using the ranking policies in a cyclic portfolio instead.

5. On-line Adaptation

The LNS approach to 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 21 shows the fraction of local improvements found by each of 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, window generation strategies and ranking policies may also be adapted to the current problem, so that the system is more likely to select subplans for optimisation that are improvable.

We use an on-line machine learning technique – the multi-armed bandit (MAB) model, to be specific – to select the subplanner for each local optimisation attempt. This technique, and its impact on the anytime performance of BPO2 is described in the following subsections.

For window selection, 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 4.6). 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. The number of windows in each neighbourhood that are optimised is typically small compared to the number of candidate windows generated. On average across all problems in experiment 3 (cf. Section 3.1 on page 392) optimisation by at least one subplanner is tried on 24.8% of generated windows. Because of this, adapting the ranking policy has more influence over

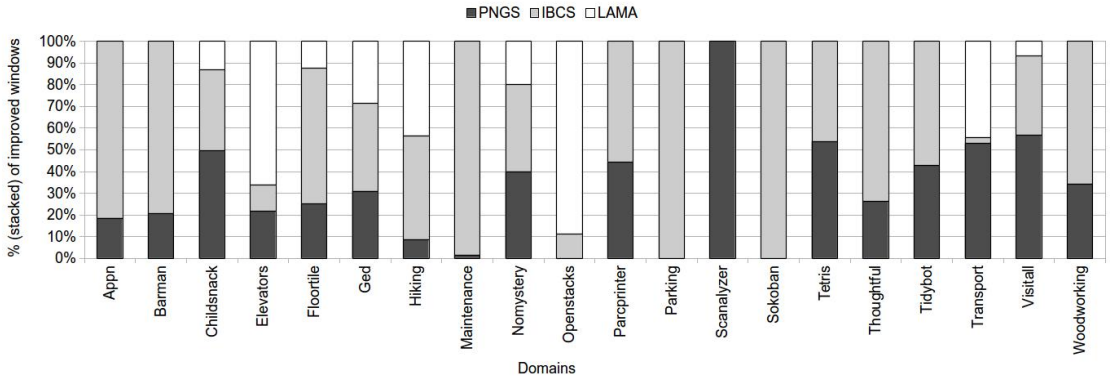

Figure 21: The percentage of improved windows found by each of the subplanners (PNGS, IBCS, and LAMA), out of the total number of improved windows found by all the subplanners. In this experiment, BDPO2 was run three times, each time with one subplanner. The setup was the same as for experiment 3 (described in Section 3.1 on page 392).

which windows are tried than adapting the windowing strategies. The effect of adaptive window ranking on the anytime performance of BDPO2 is shown in Figure 20 (page 417).

5.1 Bandit Learning

The *multi-armed bandit* (MAB) model is a popular machine learning formulation for dealing with the exploration versus exploitation dilemma. In a MAB problem, an algorithm is presented with a sequence of trials. In each round, the algorithm chooses one 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. 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.

MAB has found numerous applications in diverse fields (e.g., control, economics, statistics, and learning theory) after the influential paper by Robbins (1952). Many policies have been proposed for the MAB problem under different assumptions, for example, with independent (Auer et al., 2002) or dependent arms (Pandey, Chakrabarti, & Agarwal, 2007), exponentially or infinitely many arms (Wang, Audibert, & Munos, 2008), finite or infinite time horizon (Jones & Gittins, 1974), with or without contextual information (Slivkins, 2014), and so on.

We cast the problem of selecting the subplanner for each local optimisation attempt as a multi-armed bandit problem. The goal of this is to maximise the total number of improved windows over time. We use a learning algorithm based on the optimistic exploration strategy, which chooses 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 each trial t, and for each arm k, the strategy is to use past observations and a probabilistic argument to define high-probability confidence intervals for the expected reward μ_k. The most favorable environment for arm k is thus the upper

confidence bound (UCB) on μ_k. A simple policy based on this strategy is to play the arm having the highest 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, Munos and Szepesvári (2009), and KL-UCB by Garivier and Cappé (2011). We use the UCB1 algorithm for planner selection. The UCB1 algorithm selects at each trial t 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 trial t, and n_k is the number of times arm k has been tried so far. The second term, $\sqrt{\frac{2 \ln t}{n_k}}$, 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. The UCB1 algorithm can achieve logarithmic regret uniformly over the number of trials and without any preliminary knowledge about the reward distributions (Auer et al., 2002).

Applied to subplanner selection in BDPO2, the algorithm works as follows: First, we select each subplanner p once, to initialise the average reward $\widehat{\mu}_p$. After each optimisation attempt, we give a reward of 1 to the chosen subplanner if it found an improvement and a reward of 0 otherwise. We could use some other scheme for assigning rewards rather than simply 0 and 1, for example, making the reward proportional to the amount of improvement (or time taken to find it). However, we have observed that assigning varying rewards to subplanners makes the bandit learning system more complicated, and does not help in achieving better overall result. Next, we select for each attempt a subplanner p that maximises the upper confidence bound of p, $B_{p,t} = \widehat{\mu}_p + \sqrt{\frac{2 \ln t}{n_p}}$, 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 $B_{p,t}$ grows with t but shrinks when t and n_p increase uniformly. This ensures that 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 μ_{p^*}, where p^* is the planner with the best average reward.

5.2 The Impact of Bandit Learning

The response of the bandit policy for subplanner selection is shown in Figure 22. 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 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

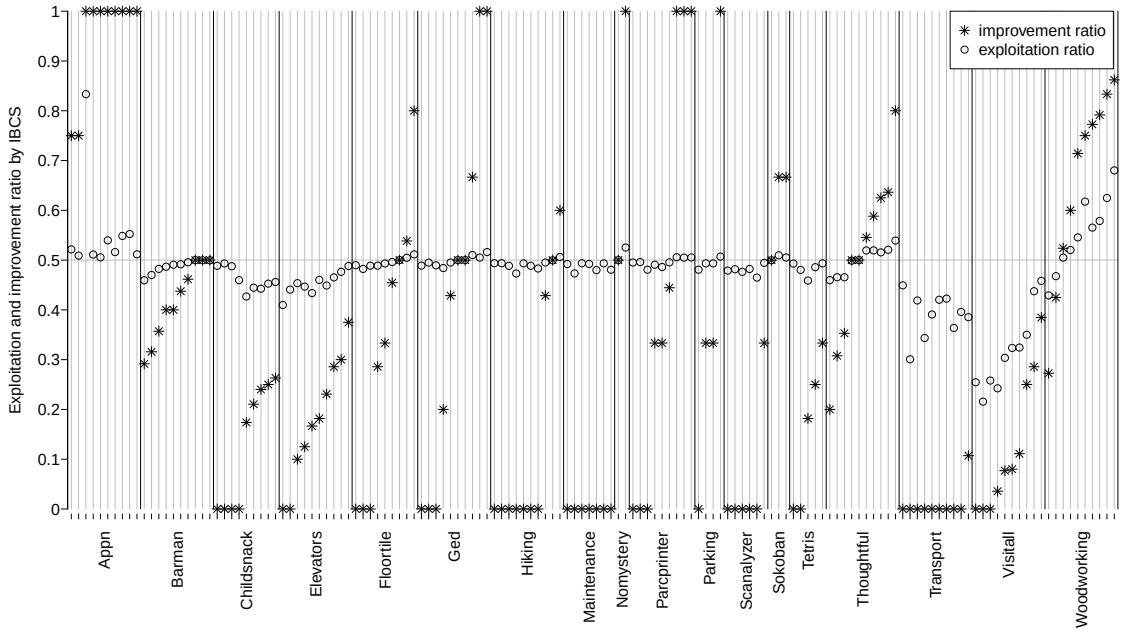

Figure 22: The response of the bandit policy to subplanner success rates. The exploitation ratio is the fraction of the total number of optimisation attempts for which the IBCS subplanner was chosen, out of the total number of attempts by both subplanners. The improvement ratio is the fraction of the 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 3.1 on page 392.

becomes 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 23 shows the impact of combining the subplanners using the UCB1 bandit policy, compared to simply alternating between subplanners or using each subplanner alone, on the anytime performance of BDPO2. In this experiment we ran BDPO2 once with each of IBCS, PNGS and LAMA as the only subplanner, once combining two of them (IBCS and PNGS) using a simple alternation policy, which selects each of the two in turn, and once combining the two using the bandit policy. Each run was done with experiment setup 3 (as described in Section 3.1 on page 392), i.e., with input plans of a high quality. (The IPC plan quality score of each plan is calculated as before; see page 394). The average score of the input plans is 0.907.) As expected, combining the IBCS and PNGS subplanners in some fashion leads to more quality improvement across the entire time scale than achieved by running BDPO2 with any individual subplanner. The figure also shows that combining multiple subplanners using the bandit policy is a better strategy than simply alternating between

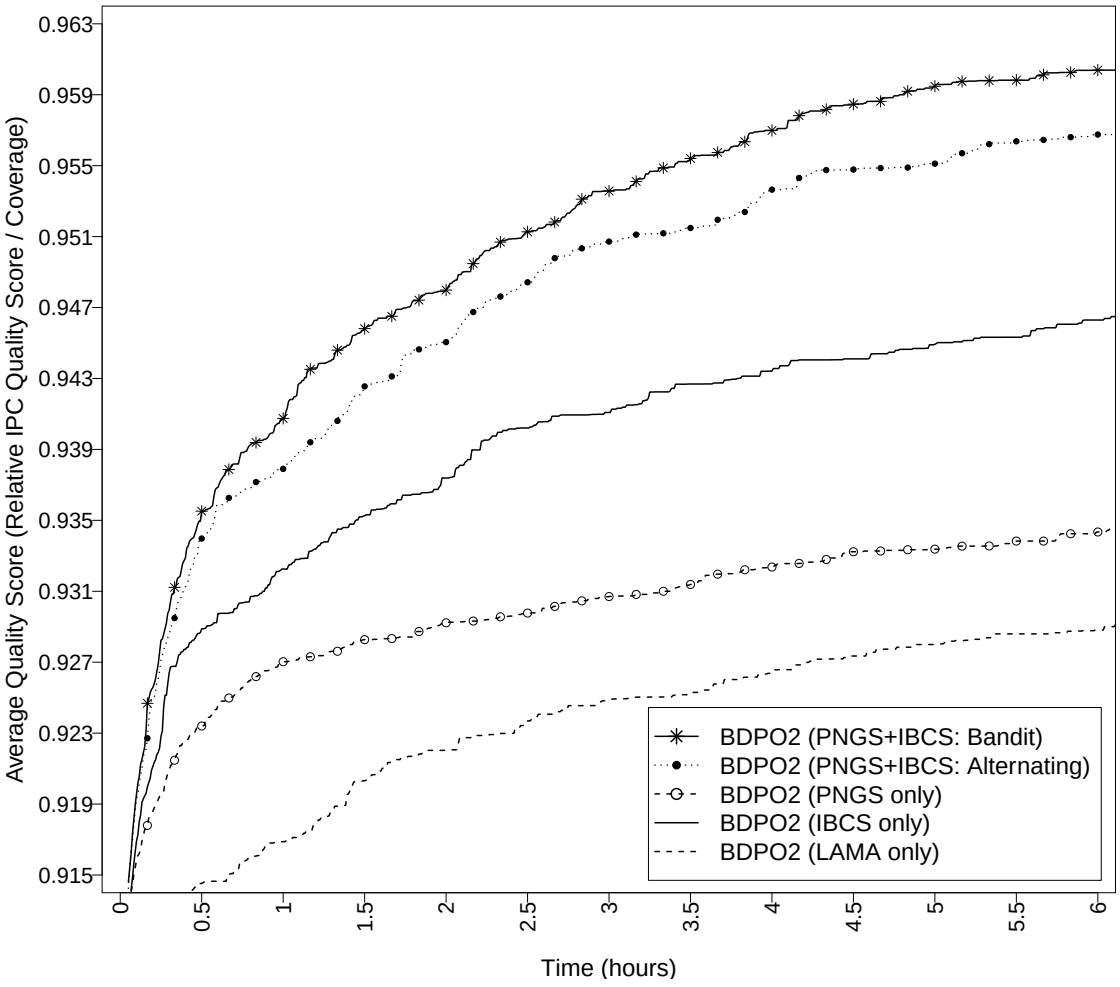

Figure 23: Average IPC quality score as a function of time per problem in five different runs of BDPO2: using only each one of the three subplanners, using two of them (IBCS and PNGS) combined with the UCB1 bandit policy, and without (using simple alternation instead). This experiment was run with setup 3 as described in Section 3.1 (on page 392). Note that the y-axis is truncated: All curves start at the average quality of the input plans, which 0.907. The time shown here is the runtime of BDPO2 only, excluding the 2 hour delay for generating the input plans shown in Figure 11).

them. The total quality improvement achieved by BDPO2 using the alternation policy is 6.8% less than that by BDPO2 using the bandit policy.

6. 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.

6.1 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 exhaust memory before finding a solution. 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 & 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 the LAMA planner (Richter & Westphal, 2010) LAMA finds the first plan using a greedy best-first search (Bonet & Geffner, 2001). It also uses several search enhancements, like preferred operators and deferred evaluation (Richter & Helmert, 2009). 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. It uses both an admissible heuristic for plan cost and more informative but inadmissible estimates to guide the search. AEES (Thayer et al., 2012b) is an anytime version of EES. To achieve an anytime behavior, AEES lowers the value of w whenever a new best solution is found.

The bounded-cost search (Stern, Puzis, & Felner, 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 C. The aim of a bounded-cost search algorithm is to find such a solution as quickly as possible. 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 BEES and BEEPS algorithms (Thayer, Stern, Felner, & Ruml, 2012a) adapt EES to the setting of bounded cost search. These algorithms expand

the best open node among those whose inadmissible cost estimate is at most C, falling back to to expanding the node with the best admissible estimate only if the set is empty.

Branch-and-bound algorithms explore the search space in some systematic fashion, using an admissible heuristic (lower bound on cost) to prune nodes that cannot lead to a solution better than the best found so far. Branch-and-bound can be implemented with a linear-memory, depth-first search strategy as well as on top of other strategies. In the experiment reported in Section 3.2 (page 394) we used Beam-Stack Search (BSS) (Zhou & Hansen, 2005) as a bounded-cost search algorithm by providing as an initial upper bound the cost of the base plan for each problem. BSS combines backtracking branch-and-bound with beam search, which behaves like a breadth-first search but limits the size of the open list in each layer by a user-specified parameter, known as the beam width. When forced to backtrack, BSS reconstructs nodes pruned from the open list so that the search is complete. The beam width parameter can be used to control the memory consumption of BSS so that it never exceeds available memory. For planning problems, however, whose state spaces are often dense in transpositions and where accurate admissible heuristics are expensive to compute, repeatedly reconstructing paths to unexplored nodes becomes time-consuming.

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 11 on page 395 and Section 3.2 on page 394.) For example, in our experiment LAMA and AEES found better plans for only 8.7% and 6.1%, respectively, of the total number of problems between 3 hours and 6 hours CPU time, while BDPO2 found better plans for 30.4% of the problems during the same time interval. Memory is one limiting factor, but not the only one. For almost half the problems, AEES ran for a full 7 hours CPU time without running out of memory, yet found very few improved plans. BSS found plans with a cost less than the initial upper bound (the cost of the base plans) for only 14 out of 182 problems even after 24 hours CPU time per problem.

6.2 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. 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 & 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, Nakhost, & Müller, 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 on hard problems in many domains. The execution of Arvand consists of a series of search episodes. Each episode starts with a set of random walks from the initial state. The endpoint of each random walk is 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 repeats until either the goal is reached, or enough transitions are made without heuristic progress, in which case the process is restarted. The IPC 2011 and 2014 versions of Arvand apply post-processing to improve the quality of each generated plan. The post-processing techniques are Action Elimination and Plan Neighborhood Graph Search (Nakhost & Müller, 2010); they are discussed in the next subsection. Because Arvand's search is randomised, the system can continue generating alternative plans, which are then optmised, indefinitely, storing at all times the best plan generated so far. This provides a certain anytime capability. It is in this manner that it was used in the experiment reported in Section 3.2 on page 394.

The LPG planner (Gerevini & 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.

There is a close relationship between local search approaches to planning and plan repair or adaptation methods (Garrido, Guzman, & Onaindia, 2010). The LPG planner originated as a method of plan repair (Gerevini & Serina, 2000), and iterative repair methods can also be used for plan generation (Chien, Knight, Stechert, Sherwood, & Rabideau, 2000).

A key difference between our use of local search and its previous uses in planning is that we carry out a local 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 in LPG, requires neighbourhood evaluation to consider how close an element is to becoming a valid plan, and balancing that with quality.

The large neighbourhood search (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, Laborie, & Nuijten, 2005). Theoretical

and experimental studies have shown that the increased neighborhood size may improve the effectiveness (quality of solutions) of local search algorithms (Ahuja, Goodstein, Mukherjee, Orlin, & Sharma, 2007). 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 a "destroy" heuristic and 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 often includes an element of randomness, enabling the search to explore modifications to different parts of the current solution. 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 algorithms (e.g., Ropke & Pisinger, 2006; Schrimpf et al., 2000) allow the local search to move to a neighbouring solution with a lower quality (e.g., using simulated annealing). We consider only strictly improving moves. However, in difference to previous 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 3.4 on page 399).

6.3 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. This is also related to plan repair and adaptation (Chien et al., 2000; Fox, Gerevini, Long, & Serina, 2006; Garrido et al., 2010), but with the key difference that plan repair or adaptation starts from a plan that is not valid for the current situation and focuses on making it work; the discrepancy between the current state or goals and those the plan was originally built for provide guidance to where repairs are needed. In contrast, post-processing for plan optimisation may require modifications anywhere in the current plan.

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. Furcy's (2006) Iterative Tunneling Search with A* (ITSA*) 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 plan improvements quickly, but running it for an additional 6 hours improves its average IPC plan quality score, over that of the best plans it finds in the first hour, only by 0.01%. Running instead BDPO2, using PNGS as the only subplanner and taking the best plans found by PNGS in 1 hour as input, improves the average plan quality score by 3% in 6 hours.

Ratner and Pohl (1986) used local optimisation for shortening solutions to sequential search problems. To select the subpath to optimise, they used a sliding window of a pre-defined size $d_{\max}$ over consecutive segments of the current path. Estrem and Krebsbach (2012) instead used a form of windowing heuristic: They select for local optimisation pairs of states on the current path that maximise an estimate of redundancy, based on the ratio between the estimated distances between the two states, given by a state space heuristic, and the cost of the current path. Balyo, Barták and Surynek (2012) used a sliding window approach to minimise parallel plan length (that is, "makespan", assuming all actions have unit duration). Rather than take segments of a single path in the state space, we use block deordering of the input plan to create candidate windows for local optimisation. As shown by the experimental results, this is very important for the success of BDPO2: The total improvement of average plan quality achieved without deordering was 28.7% less than that achieved by BDPO2 using block deordering of input plans (cf. Section 3.6 on page 402).

The planning-by-rewriting approach (Ambite & 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.

6.4 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 it at all. Therefore, many of todays most successful planners run a sequential portfolio of planners (Coles, Coles, Olaya, Celorrio, Linares López, Sanner, & Yoon, 2012).

Gerevini, Saetti and Vallati (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, Röger, Seipp, Karpas, Hoffmann, Keyder,

Nissim, Richter, & Westphal, 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, Hoos, Leyton-Brown, & Stützle, 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, Fawcett, Gerevini, Hoos, & Saetti, 2011) and the Fast Downward planner (Fawcett, Helmert, Hoos, Karpas, Röger, & Seipp, 2011). The PbP2 portfolio planner (Gerevini, Saetti, & Vallati, 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. Section 5). Much more important, however, is the focus on subproblems that our approach brings: comparing Figures 11 (page 395) and 23 (page 422), 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 4.6) 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, our results 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 or IBaCoP2, as in the results of experiment 3, shown in Figure 2 on page 371), often achieves better quality final plans than investing all available time into any single method.

7. Conclusions and Future Work

Plan quality optimisation, particularly for large problems, is a central concern in automated planning. 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". We have presented an approach to anytime plan improvement, and its realisation in the BDPO2 system. This approach is based on the large neighbourhood local search strategy (Shaw, 1998), using windowing heuristics to select candidate windows from a block deordering of the current plan, for local optimisation using off-the-shelf bounded-cost planning techniques.

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 4.5, extending the windowing heuristics and improving the on-line learning of effective window rankings is one way to improve the approach. Also, complementing the window ranking, which estimates how "promising" a window is, with 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 or IBaCoP2). This result cannot be achieved by any of the previous anytime planning approaches alone. Thus, another area of future work is to examine in greater depth what is the best way to combine different plan improvement methods, and how this can be learned on-line while optimising a plan. For example, we have conducted a study of the optimal time to switch from base plan generation, using LAMA, to post-processing using PNGS or BDPO, as a function of the total runtime (Siddiqui & Haslum, 2013a).

As we have demonstrated experimentally, the block deordering step is essential for the good performance of BDPO2 (cf. Section 3.6 on page 402). Block deordering creates a decomposition of the plan into non-interleaving blocks while removing ordering constraints between blocks. This lifts a limitation of conventional, "step-wise", deordering, which requires all unordered steps in the plan to be non-interfering. 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 on page 379). Therefore, block deordering can yield less order-constrained plans, including in some cases where no conventional deordering is possible.

The plan structure uncovered by block decomposition can also have other uses. Recently it was used in the planner independent macro generation system BloMa (Chrpa & Siddiqui, 2015) to find longer macros that capture compound activities in order to improve planners' coverage and efficiency. In some domains (e.g., Barman, ChildSnack, Scanalyzer, Parcprinter, Gripper, Woodworking, etc.), block deordering often identifies structurally similar subplans, which also have symmetric improvement patterns. This could potentially be exploited in learning plan rewrite rules (Ambite, Knoblock, & Minton, 2000). The structure of block deordered plans, which often comprises a nested, hierarchical decomposition into meaningful subplans, is reminiscent of Hierarchical Task Network (HTN) representations. Hence, block deordering technique could potentially be applied to generating (or helping to generate) HTN structures in a domain independent way, reducing the knowledge-engineering effort. Recent work by Scala and Torasso (2015) extends deordering to plans for planning domains with numeric state variables, identifying numeric dependencies that capture the additional reasons for necessary orderings. Defining the conditions on blocks sufficient to encapsulate these dependencies would allow block deordering also of numeric plans. There may be a synergy between block deordering and numeric planning, since numeric dependencies often involve groups of plan steps, rather than a single producer–consumer pair.

Acknowledgment

This work was partially supported by the Australian Research Council discovery project DP140104219 "Robust AI Planning for Hybrid Systems". NICTA is funded by the Aus-

tralian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

Ahuja, R. K., Goodstein, J., Mukherjee, A., Orlin, J. B., & Sharma, D. (2007). A very large-scale neighborhood search algorithm for the combined through-fleet-assignment model. *INFORMS Journal on Computing*, *19*(3), 416–428.

Ambite, J. L., & Knoblock, C. A. (2001). Planning by rewriting. *Journal of Artificial Intelligence Research (JAIR)*, *15*(1), 207–261.

Ambite, J. L., Knoblock, C. A., & Minton, S. (2000). Learning plan rewriting rules. In *Proc. of the 5th International Conference on Artificial Intelligence Planning Systems, AIPS 2000, Breckenridge, CO, USA, April 14-17, 2000*, pp. 3–12. AAAI Press.

Audibert, J.-Y., Munos, R., & Szepesvári, C. (2009). Exploration–exploitation tradeoff using variance estimates in multi-armed bandits. *Theoretical Computer Science*, *410*(19), 1876–1902.

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, *47*(2-3), 235–256.

Bäckström, C. (1998). Computational aspects of reordering plans. *Journal of Artificial Intelligence Research (JAIR)*, *9*, 99–137.

Balyo, T., Barták, R., & Surynek, P. (2012). 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.

Bedo, J., & Ong, C. S. (2014). Multivariate Spearman's rho for aggregating ranks using copulas. *CoRR*, *abs/1410.4391*.

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

Cenamor, I., de la Rosa, T., & Fernández, F. (2014). IBaCoP and IBaCoP2 planners. In *Proc. of the 8th International Planning Competition, IPC 2014, Deterministic Part*, pp. 35–38.

Chapman, D. (1987). Planning for conjunctive goals. *Artificial Intelligence*, *32*(3), 333–377.

Chien, S., Knight, R., Stechert, A., Sherwood, R., & Rabideau, G. (2000). Using iterative repair to improve the responsiveness of planning and scheduling. In *Proc. of the 5th International Conference on Artificial Intelligence Planning Systems, AIPS 2000, Breckenridge, CO, USA, April 14-17, 2000*, pp. 300–307. AAAI Press.

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*, pp. 1537–1543. AAAI Press.

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

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

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

Dwork, C., Kumar, R., Naor, M., & 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*, pp. 613–622, New York, NY, USA. ACM.

Estrem, S. J., & 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.*

Fawcett, C., Helmert, M., Hoos, H., Karpas, E., Röger, G., & 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*, pp. 13–20. AAAI Press.

Fox, M., Gerevini, A., Long, D., & Serina, I. (2006). Plan stability: Replanning versus plan repair. In *Proc. of the 16th International Conference on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK, June 6-10, 2006.*, pp. 212–221. AAAI Press.

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 1620, 2006, Boston, Massachusetts*, pp. 21–26. AAAI Press.

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

Garrido, A., Guzman, C., & Onaindia, E. (2010). Anytime plan-adaptation for continuous planning. In *Proc. of the joint 28th Workshop of the UK Special Interest Group on Planning and Scheduling and 4th Italian Workshop on Planning and Scheduling*, pp. 47–54.

Gerevini, A., Saetti, A., & 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*, pp. 350–353. AAAI Press.

Gerevini, A., Saetti, A., & 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.

Gerevini, A., & 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*, pp. 281–290. AAAI Press.

Gerevini, A. E., & Serina, I. (2000). Fast plan adaptation through planning graphs: Local and systematic search techniques. In *Proc. of the 5th International Conference on Artificial Intelligence Planning Systems, AIPS 2000, Breckenridge, CO, USA, April 14-17, 2000*, pp. 112–121. AAAI Press.

Ghallab, M., Nau, D. S., & Traverso, P. (2004). *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Godard, D., Laborie, P., & 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*, pp. 81–89. AAAI Press.

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.

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*, pp. 74–82. AAAI Press.

Haslum, P., & 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.

Haslum, P., & 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*, pp. 150–158. AAAI Press.

Helmert, M., Röger, G., Seipp, J., Karpas, E., Hoffmann, J., Keyder, E., Nissim, R., Richter, S., & 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.

Helmert, M., & 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*, pp. 162–169. AAAI Press.

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*, pp. 453–458, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Hoffmann, J., & Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, *14*, 253–302.

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

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

Kambhampati, S., & Kedar, S. (1994). A unified framework for explanation-based generalization of partially ordered and partially instantiated plans. *Artificial Intelligence*, *67*(1), 29–70.

McAllester, D., & Rosenblitt, D. (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.*, pp. 634–639. AAAI Press / The MIT Press.

Muise, C. J., McIlraith, S. A., & 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*, pp. 358–362. AAAI Press.

Nakhost, H., & 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, pp. 1766–1771.

Nakhost, H., & 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*, pp. 121–128. AAAI Press.

Nebel, B., & Bäckström, C. (1994). On the computational complexity of temporal projection, planning, and plan validation. *Artificial Intelligence*, *66*(1), 125–160.

Pandey, S., Chakrabarti, D., & 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, pp. 721–728. ACM.

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

Pohl, I. (1970). Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, *1*(3), 193–204.

Ratner, D., & 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.*, pp. 173–177. Morgan Kaufmann.

Régnier, P., & 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*, pp. 100–111. Springer.

Richter, S., & 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*, pp. 273–280. AAAI Press.

Richter, S., Thayer, J. T., & 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*, pp. 137–144. AAAI Press.

Richter, S., & Westphal, M. (2010). The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research (JAIR)*, *39*, 127–177.

Robbins, H. (1952). Some aspects of the sequential design of experiments. In *Herbert Robbins Selected Papers*, Vol. 58, pp. 527–535. Springer.

Ropke, S., & Pisinger, D. (2006). An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, *40*(4), 455–472.

Scala, E., & 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*, pp. 1673–1681. AAAI Press.

Schrimpf, G., Schneider, J., Stamm-Wilbrandt, H., & Dueck, G. (2000). Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, *159*(2), 139–171.

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*, pp. 417–431. Springer.

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*, pp. 803–814, Berlin, Heidelberg. Springer.

Siddiqui, F. H., & 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*, pp. 22–31. http://icaps13.icaps-conference.org/wp-content/uploads/2013/05/evops13-proceedings.pdf.

Siddiqui, F. H., & 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*, pp. 2387–2393. AAAI Press.

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

Stern, R. T., Puzis, R., & 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*, pp. 234–241. AAAI Press.

Thayer, J., Stern, R., Felner, A., & 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*, pp. 270–278. AAAI Press.

Thayer, J. T., Benton, J., & 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*, pp. 120–128. AAAI Press.

Thayer, J. T., & 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*, pp. 674–679. AAAI Press.

Vallati, M., Fawcett, C., Gerevini, A., Hoos, H., & 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.

Veloso, M. M., Pérez, A., & 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*, pp. 207–212. Morgan Kaufmann.

Wang, Y., Audibert, J., & 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*, pp. 1729–1736. Curran Associates, Inc.

Xie, F., Nakhost, H., & 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*, pp. 315–322. AAAI Press.

Xie, F., Valenzano, R. A., & Müller, M. (2010). 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*, pp. 269–277. AAAI Press.

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

Zhou, R., & 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*, pp. 90–98. AAAI Press.