Plan Quality Optimisation via Block Decomposition

Fazlul Hasan Siddiqui and **Patrik Haslum**
The Australian National University & NICTA Optimisation Research Group
Canberra, Australia
firstname.lastname@anu.edu.au

Abstract

AI planners have to compromise between the speed of the planning process and the quality of the generated plan. Anytime planners try to balance these objectives by finding plans of better quality over time, but current anytime planners often do not make effective use of increasing runtime beyond a certain limit. We present a new method of continuing plan improvement, that works by repeatedly decomposing a given plan into subplans and optimising each subplan locally. The decomposition exploits block-structured plan deordering to identify coherent subplans, which make sense to treat as units. This approach extends the "anytime capability" of current planners – to provide continuing plan quality improvement at any time scale.

1 Introduction

Producing high quality plans and producing them fast are two of the main aims of automated planning. There is, however, a tension between these two goals: plans found quickly are often of poor quality, and plans of good quality tend to be hard to find (except perchance if "quality" is taken to be just plan length). Although much progress has been made both on fast planning methods (without quality guarantees) and on optimal (and bounded suboptimal) planning methods, there is a gap between the capabilities of these two classes of planners: optimal planners are far from scaling up to the size of problems that fast, "any-solution" planners can solve, but the quality of plans found by such planners is often equally far from optimal. Few, if any, planners have the flexibility to be used at any point on the efficiency–quality trade-off scale.

Anytime planners promise to provide that flexibility, by finding an initial plan, possibly of poor quality, quickly and then continually finding better plans the more time they are given. The current best approach to anytime planning is based on restarting weighted A* search with a schedule of decreasing weights [Richter *et al.*, 2010], as implemented in the LAMA planner. This method, however, does not quite live up to the promise of continually improving plans over time. As the weight used in WA* decreases, it fairly quickly degenerates into a plain A* search (with an inadmissible heuristic).

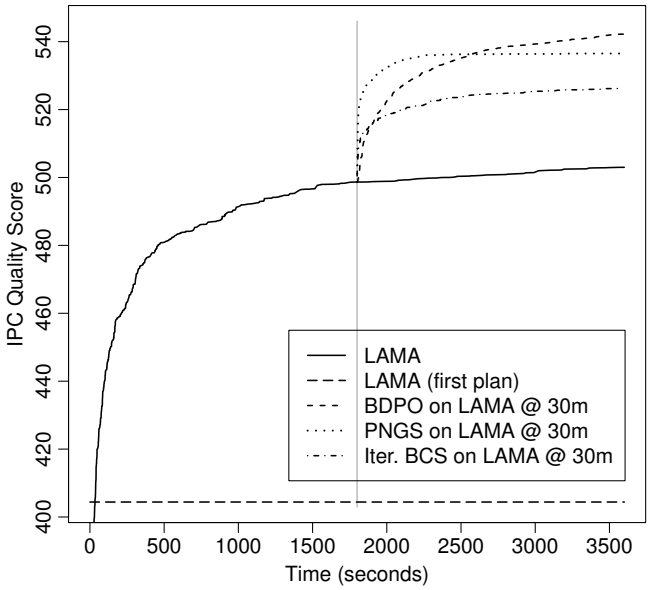

Figure 1: Total IPC quality score as a function of time for LAMA and three plan optimisation methods, on all problems used in our experiments (606 problems, of which LAMA solves 577). The score of a plan is calculated as c^{ref}/c, where c is the cost of the plan and c^{ref} the "reference cost" (best cost of all plans for the problem). Thus, a higher score reflects a lower-cost plan. *Note that the y-axis is truncated.*

This can be seen in the compound anytime profile of the planner, shown in Figure 1: it rises very sharply in the beginning, meaning that a small amount of additional time at this stage leads to a big improvement (mainly due to more problems solved), but then quickly flattens out. Over the second half of the 1 hour time limit, the improvement in plan quality (as measured by the summed IPC quality score) is less than 1%. Yet, as the figure also shows, it is possible to attain a considerably greater improvement.

The key to this continuing improvement of plan quality is focusing optimisation effort. We present a way to do this, by analysing the structure of the initial plan to identify coherent subplans, which are optimised independently using bounded-cost search. Improved subplans are substituted into the plan,

with multiple such substitutions made if possible. The process is then repeated, starting from the new best plan.

The main challenge is to find the right subplans for local optimisation. For this, we use plan deordering: A partially ordered plan exhibits some structure, in that dependencies and threats between the steps in the plan are explicit. In a partially ordered plan, we can identify independent "threads" of activity, and the points where these threads split and join. The sequential subplans between these points form the basic building blocks of our decomposition strategy. Standard plan deordering, however, requires unordered plan steps to be non-interfering. This limits its applicability, so that in many cases no deordering of a sequential plan is possible. To overcome this limitation, we use the idea of block decomposition of plans [Siddiqui and Haslum, 2012]. Intuitively, a "block" is a subset of plan steps that encapsulates some interactions. Decomposing a plan into blocks allows some blocks to be unordered even when their constituent steps cannot. This increased deordering of the plan makes the strategy for finding subplans more effective, and more widely applicable.

2 Related Work

The idea of optimising a solution one small part at a time has been used very successfully in constraint-based approaches to hard combinatorial optimisation problems like vehicle routing [Shaw, 1998] and scheduling [Godard *et al.*, 2005], where it is known as "large neighbourhood search" (LNS). In this setting, a local search step solves the constraint optimisation problem, using an expensive but highly optimising search, for a small set of variables, keeping remaining variables fixed at their current values. Our plan improvement approach can be viewed as applying LNS to planning. A key difficulty in LNS is defining the neighbourhood, i.e., identifying good subproblems to re-optimise. Routing and scheduling solvers using LNS rely on problem-specific heuristics, based on insights into the problem and its constraint formulation, for this. For planning, we need automatic and domain-independent methods: this is what our plan decomposition strategy provides. Our local search is a plain hill-climbing search, always moving to a better (but not necessarily best) plan in the neighbourhood. It might be improved through use of more sophisticated meta-heuristics, such as simulated annealing or restarts.

Ratner and Pohl [1986] use local optimisation to shorten solutions to sequential search problems. However, their approach to subproblem identification is a simple sliding window over consecutive segments of the current path. This is unlikely to find relevant subproblems for optimising plan cost in general planning problems, where the sequential plan is often an arbitrary interleaving of separate causal threads. In experiments, 75% of the subproblems for which we find an improved subplan correspond to non-consecutive parts of the original plan. Similarly, Balyo, Barták and Surynek [2012] use a sliding window to minimise parallel plan length.

Plan neighbourhood graph search (PNGS) [Nakhost and Müller, 2010] constructs a subgraph of the state space of the problem, limited to a fixed distance d from states traversed by the current plan, and searches for the cheapest plan in this subgraph. If it improves on the current plan, the process is repeated around the new best plan; otherwise, the distance is increased. ITSA* [Furcy, 2006] similarly explores an area of the state space near the current plan. Compared to our method (and LNS, generally) these can be seen as using a different neighbourhood, that includes only small deviations from the current plan, but anywhere along the plan. In contrast, we focus on a section of the plan at a time, but do not restrict how much the replacement subplan differs from the original plan section. As we will show, our method and PNGS have complementary strengths. Thus, a local search over both types of neighbourhoods might be even more effective.

The planning-by-rewriting approach [Ambite and Knoblock, 2001] also uses local modifications of partially ordered plans to improve their quality. Plan modifications are defined by (domain-specific) rewrite rules, which have to be provided by the domain designer or learned from many examples of both good and bad plans. Using instead a planner to solve the local improvement subproblem may be more time-consuming than applying pre-defined rules, but makes the process fully 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 local plan improvements we discover and using these where applicable to avoid calling the planner.

3 Plan Deordering and Block Decomposition

We use the standard STRIPS model of classical planning problems. We briefly recap some basic notions concerning partially ordered plans. For a detailed introduction, see, for example, the book by Ghallab et al. [2004].

A *partially ordered plan* (POP) is a tuple $(S, \prec)$, where S is the set of plan steps and $\prec$ is a strict partial order on S; $\prec^+$ denotes the transitive closure of $\prec$. A *linearisation* of a POP is a strict total order that contains $\prec$. Each step $s \in S$, except for the initial and goal steps, is labelled by an action, $\mathrm{act}(s)$, which, as usual, has precondition, added and deleted sets of propositions. With slight abuse of terminology, we talk about the preconditions and effects of a step, meaning those of its associated action. A *causal link*, (s_i, p, s_j), records a commitment that the precondition p of step s_j is supplied by an add effect of step s_i. The link is *threatened* if there is a step s_k that deletes p such that $s_i \prec s_k \prec s_j$ is consistent. The validity of a POP can be defined in two equivalent ways: (1) a POP is valid iff every linearisation of its actions is a valid sequential plan, under the usual STRIPS execution semantics; and (2) a POP is valid if every step precondition (including the goals) is supported by an unthreatened causal link. (That is, essentially, Chapman's [1987] modal truth condition.)

Block deordering [Siddiqui and Haslum, 2012] restricts the linearisations for which a POP must be valid. A *block* is a subset of plan steps that must not be interleaved with steps not in the block; steps within a block may still be partially ordered. A *block decomposition* divides plan steps into blocks that do not overlap. The decomposition may be recursive, so a block can be wholly contained in another, though. Blocks behave much like (non-sequential) macro actions, having preconditions, add and delete effects that can be a subset of the

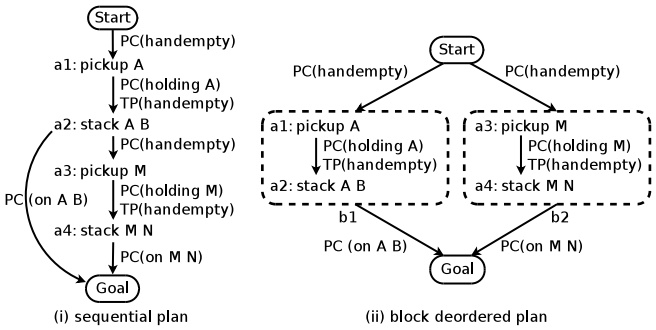

Figure 2: A sequential plan and its block deordering. Precedences are labelled with their reasons: producer–consumer (i.e., a causal link), denoted $\mathrm{PC}(p)$; threat–producer, denoted $\mathrm{TP}(p)$; and consumer–threat, denoted $\mathrm{CT}(p)$.

union of those of its constituent steps. This enables blocks to encapsulate some plan effects, reducing interference and thus allowing blocks to be unordered even when some steps in them may not. As a simple example, Figure 2(i) shows a sequential plan for a small Blocksworld problem. This plan can not be deordered into a convential POP, because each plan step has a reason to be ordered after the previous. In the block deordered plan (ii), neither of the two blocks delete or add proposition handempty (though it is a precondition of both). This removes the interference between them, and allows the two blocks to be unordered. Note that the possible linearisations of the block decomposed partially ordered plan are only a_1, a_2, a_3, a_4 and a_3, a_4, a_1, a_2.

We first apply standard deordering, using essentially the PRF algorithm [Bäckström, 1998]. Next, we apply a (non-optimal) heuristic procedure to automatically find a block decomposition that tries to maximise further deordering. For a detailed description of the block deordering procedure, we must refer to the earlier paper [Siddiqui and Haslum, 2012]. Our interest here is not in deordering per se, but using the structure it reveals to find subplans suitable for local optimisation. However, block deordering tends to produce blocks that localise interactions as much as possible, i.e., that are as "self-contained" as they can be, and this is useful also for local plan optimisation. Validity of a block decomposed partially ordered plan can be established in the same way as for POPs, by supporting each precondition of each block with an unthreated causal link. The block deordering algorithm returns not just the decomposed and deordered plan, but also a justification for its correctness, by labelling ordering constraints with their reasons (causal links or threats). We exploit this when merging improvements to separate parts of a plan.

4 Plan Optimisation Algorithm

The plan optimisation algorithm (Algorithm 1) consists of three main steps: First, block deordering is applied to the input plan (π_{in}), producing a block decomposition that minimises inter-block dependencies. Second, a set of candidate subplans for optimisation, termed "windows", are extracted from the block deordered plan. Window formation rules are divided into three groups, and the corresponding three sets of

Algorithm 1 Block Decomposition Plan Optimisation

procedure BDPO($\Gamma, \pi_{\text{in}}, t_{\text{limit}}, g$)
  Initialise $t_{\text{elapsed}} = 0$, $C_{\text{sp}} = \emptyset$, $\pi_{\text{last}} = \pi_{\text{in}}$.
  $\pi_{\text{bdp}} = \textsc{BlockDeorder}(\pi_{\text{in}})$.
  Set time bound $t_b =$ *initial time bound*.
  while $C_{\text{sp}} = \emptyset$ **do**
    if W_g not initialised **then**
      Set $W_g = \textsc{FormWindows}(\pi_{\text{bdp}}, g)$.
    for each window $(p_i, w_i, s_i) \in W_g$ **do**
      if $t_{\text{elapsed}} \geq t_{\text{limit}}$ **then return** π_{last}.
      $\Gamma^i_{\text{sub}} = \textsc{Subproblem}(p_i, w_i, s_i)$.
      if $h(\pi^i_{\text{sub}}) = \text{cost}(w_i)$ **then**
        Set $W_g = W_g \setminus \{(p_i, w_i, s_i)\}$.
        continue
      $\pi^i_{\text{sub}} = \textsc{BoundedCostPlanner}$
        $(\Gamma^i_{\text{sub}}, \text{cost}(w_i), t_b)$.
      if Γ^i_{sub} proven unsolvable **then**
        Set $W_g = W_g \setminus \{(p_i, w_i, s_i)\}$.
      else if $\pi^i_{\text{sub}} \neq$ *null* **then**
        /* $cost(\pi^i_{sub}) \leq cost(w_i)$ */
        $C_{\text{sp}} = C_{\text{sp}} \cup \{(w_i, \pi^i_{\text{sub}})\}$.
        $\hat{\pi}_{\text{bdp}} = \textsc{Merge}(C_{\text{sp}}, \pi_{\text{bdp}})$.
        if $\text{cost}(\hat{\pi}_{\text{bdp}}) < \text{cost}(\pi_{\text{last}})$ **then**
          $\pi_{\text{last}} = \textsc{SequencePlan}(\hat{\pi}_{\text{bdp}})$.
    if $\pi_{\text{last}} \neq \pi_{\text{in}}$ **then**
      return BDPO $(\Gamma, \pi_{\text{last}}, t_{\text{limit}} - t_{\text{elapsed}}, g)$.
    Set $g = (g + 1) \bmod 3$ /* *next group* */
    if all groups have been tried at t_b **then**
      if $W_1 = W_2 = W_3 = \emptyset$ **then return** *null*
      Set $t_b = 2 * t_b$.

windows are tried in turn. Each window generates a bounded-cost subproblem, which is the problem of finding a cheaper replacement for that subplan. The algorithm calls a planner on each of these subproblems, with a cost bound equal to the cost of the current subplan, and a time-out. In principle, the subplanner can be any planning method that accepts a bound on plan cost; we use an iterative bounded-cost search. Whenever a better replacement subplan is found, all replacement subplans found so far (C_{sp}) are fed into a merge procedure, which tries to combine several of them to achieve a greater overall improvement. (At least one replacement subplan can always be merged, so if C_{sp} is non-empty, π_{last} is better than π_{in}.) If any improvement is found in the current set of windows, the procedure starts over with the new best plan (π_{last}); it restarts with the group of rules that generated the set of windows where the improvement was found. If no improvement has been found in all window sets, the time-out is doubled and the subplanner tried again on each subproblem, except those known to be solved optimally already. This is detected by comparing the subplan cost with a lower bound (obtained from an admissible heuristic, h), or by the subplanner proving the bounded-cost subproblem unsolvable. The initial time bound is 15 seconds, or $t_{\text{limit}}/|W_1 \cup W_2 \cup W_3|$, whichever is smaller.

4.1 Window Formation

Before extracting windows, we extend blocks to cover complete non-branching subsequences of the plan. That is, if a 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 block. (Note that when we talk about blocks here and in the following, these can also consist of single actions.)

A window is a triple (p, w, s), where w is the set of blocks in the subplan (to be replaced), and p and s are sets of blocks to be placed before and after w, respectively. Any block that is ordered before (resp. after) a block in w must be assigned to p (resp. s), but for blocks that are not ordered w.r.t. any block in w we have a choice of placing them in p or s. Let $\mathrm{Un}(b)$ be the set of blocks not ordered w.r.t. b, $\mathrm{IP}(b)$ its immediate predecessors of b, and $\mathrm{IS}(b)$ its immediate successors. The three groups of window-forming rules are:

1. $w \leftarrow \{b\}, p \leftarrow \mathrm{Un}(b)$; $w \leftarrow \{b\}, s \leftarrow \mathrm{Un}(b)$; $w \leftarrow \{b\} \cup \mathrm{Un}(b)$; $w \leftarrow \{b\} \cup \mathrm{IP}(b), s \leftarrow \mathrm{Un}(b)$; and $w \leftarrow \{b\} \cup \mathrm{IS}(b), p \leftarrow \mathrm{Un}(b)$.
2. $w \leftarrow \{b\} \cup \mathrm{Un}(b) \cup \mathrm{IP}(b)$; $w \leftarrow \{b\} \cup \mathrm{Un}(b) \cup \mathrm{IS}(b)$; and $w \leftarrow \{b\} \cup \mathrm{Un}(b) \cup \mathrm{IP}(b) \cup \mathrm{IS}(b)$.
3. $w \leftarrow \{b\} \cup \mathrm{Un}(b) \cup \mathrm{IP}(\{b\} \cup \mathrm{Un}(b))$; $w \leftarrow \{b\} \cup \mathrm{Un}(b) \cup \mathrm{IS}(\{b\} \cup \mathrm{Un}(b))$; and $w \leftarrow \{b\} \cup \mathrm{Un}(b) \cup \mathrm{IP}(\{b\} \cup \mathrm{Un}(b)) \cup \mathrm{IS}(\{b\} \cup \mathrm{Un}(b))$.

Each rule is applied to each block b, which may produce duplicates; of course, only unique windows are kept. This grouping aims to reduce overlap between windows in each set, and group them roughly by size. BDPO processes windows in each set in order of increasing size, measured by the number of actions in w.

4.2 Subproblem Construction and Subplanner

Each window (p, w, s) gives rise to a bounded-cost subproblem. This is the problem of finding a replacement for the part w of the plan, of cost less than $\mathrm{cost}(w)$, that can be substituted between plan parts p and s. The subproblem differs from the original problem only in its initial state and goal.

To find the initial state for the subproblem, we generate a linearisation, $a_{p_1}, \ldots, a_{p_k}$, of the actions in p, and progress the original initial state through this sequence, i.e., apply the actions in this sequence. To find the goal, we pick a linearisation, $a_{s_1}, \ldots, a_{s_m}$, of the actions in s, and regress the original goal backwards through this sequence. This ensures that for any plan $a'_1, \ldots, a'_n$ returned by the subplanner, the concatenation of the three sequences, i.e., $a_{p_1}, \ldots, a_{p_k}$, $a'_1, \ldots, a'_n, a_{s_1}, \ldots, a_{s_m}$, is a plan for the original problem.

The subplanner must return a plan of cost less than the given bound, $\mathrm{cost}(w)$. We use a simple bounded-cost greedy search, guided by the (unit-cost) FF heuristic and using an f-value based on the admissible LM-Cut heuristic [Helmert and Domshlak, 2009] to prune states that cannot lead to a plan within the cost bound. 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. Because bounded-cost search can return any plan that is within the cost bound, 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. This ensures we get not just an improved subplan, but the best improved subplan that the search can find within the given time limit.

As an alternative to constructing each subproblem from an arbitrary linearisation, we could take a "least committment" approach, taking as initially true only those facts that hold after any linearisation of p, and as goal all facts that must hold for every linearisation of s to succeed. (This is the same principle used to compute the preconditions and effects of a block, since steps in the block may be partially ordered.) This, however, severely restricts plan choices for the subproblem, reducing the chances of finding an improvement.

Algorithm 2 Merging Candidate Subplans

```
1: procedure MERGE(C_sp, π_bdp)
2:     Initialise π̂_bdp = π_bdp.
3:     Sort C_sp by decreasing (cost(w_i) − cost(π^i_sub)).
4:     for each (w_i, π^i_sub) ∈ C_sp in order do
5:         π_temp = REPLACEIFPOSS(w_i, π^i_sub, π̂_bdp).
6:         if π_temp ≠ null then
7:             π̂_bdp = π_temp
8:             C_sp = C_sp \ {(w_j, π^j_sub) ∈ C_sp |
                          w_j overlaps with w_i}.
9:     return π̂_bdp
```

4.3 Merging Improved Subplans

When an improved subplan is found, the window (i.e., the w part) in the original plan can be replaced with the new subplan, by construction of the subproblem. Obviously, we gain a greater improvement if we can make several replacements simultaneously. Merging all candidates (windows with improved subplans) is usually not possible, since windows may overlap. It is further complicated because each subproblem is constructed from a potentially different linearisation: the replacement subplan may have additional preconditions or delete effects that the replaced window did not, or lack some of its add effects. For a single candidate, these flaws can be resolved by adding more ordering constraints on the plan, but different candidates may require contradictory orderings.

Merging is done by a greedy procedure (Algorithm 2). It takes a current, block deordered, plan (π_{bdp}) and a set of candidates (C_{sp}), and sorts the candidates in decreasing order of their contribution to decreasing plan cost, i.e., the cost of the replaced plan part ($\mathrm{cost}(w_i)$) minus the cost of the new subplan ($\mathrm{cost}(\pi^i_{\text{sub}})$). In this order, each candidate is tried in turn: If the replacement is still possible, it is made, and any remaining candidates that overlap with the window are removed from further consideration. The first replacement is always possible, so the new plan returned by merging has lower cost than the input plan. Another, possibly better, merging strategy would be to try sets of non-overlaping candidates, i.e., independent sets, (approximately) in order of their summed plan cost decrease.

MERGE maintains at all times a valid block deordered plan ($\hat{\pi}_{\text{bdp}}$), meaning that each precondition of each block (and each goal) is supported by an unthreatened causal link. Initially, this is the input plan, for which causal links (and

additional ordering constraints) are computed by block deordering. The REPLACEIFPOSS subroutine takes the current plan, and returns an updated plan (which becomes the current plan), or failure if the replacement is not possible. Recall that preconditions and effects of a block are computed using least-committment semantics. This is done for both the replaced window (w_i) and the replacement subplan (π^i_{sub}), where the subplan is treated as a single block whose actions are totally ordered. For any atom in $\text{pre}(\pi^i_{\text{sub}})$ that is also in $\text{pre}(w_i)$, the existing causal link is kept; likewise, causal links from an effect in $\text{add}(w_i)$ that are are also in $\text{add}(\pi^i_{\text{sub}})$ 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 ($p \in (\text{pre}(\pi^i_{\text{sub}}) \setminus \text{pre}(w_i))$), a new causal link must be found, and likewise for each precondition of a later block (or the goal) that was supplied by w_i but is missing from $\text{add}(\pi^i_{\text{sub}})$. Finally, π^i_{sub} may threaten some existing causal links that w_i did not; for each of these preconditions, we also try to find a new link.

The subroutine FINDCAUSALLINK takes the current block deordered plan ($\hat{\pi}_{\text{bdp}}$), the consumer block (b), which can also be the goal, and the atom (p) that the consumer requires, and performs a limited search for an unthreatened causal link to supply it. Specifically, it tries the following two options:

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

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

If these two steps cannot find one of the required causal links, the replacement fails, and the candidate is skipped by merge.

Note that some of the ordering constraints between w_i and the rest of the plan may become unnecessary when w_i is replaced with π^i_{sub}, because π^i_{sub} may not delete every atom that w_i deletes and may not have all preconditions of w_i. Even if an ordering $b \prec w_i$, for some block b, is not required after replacing w_i with π^i_{sub}, removing it may make π^i_{sub} unordered w.r.t. blocks $b' \prec b$. Each of these must be checked for potential threats, either due to atoms deleted by π^i_{sub} or by b', and new ordering constraints $b' \prec \pi^i_{\text{sub}}$ added where needed. In the same way, if an ordering $\pi^i_{\text{sub}} \prec b$ is removed, potential threats with blocks $b' \succ b$ must be checked.

Theorem 1. *If the input plan, π_{bdp} is valid, then so is the plan returned by* MERGE.

Proof sketch. This is shown by induction on the sequence of accepted replacements. Initially, the current plan is the valid input plan. It changes only when a replacement is made. A successful replacement does not invalidate the plan: All necessary causal links to and from the replacement subplan are established by REPLACEIFPOSS. Likewise, any causal links threatened by the new subplan are re-established. The remaining possibility, that the new subplan becomes unordered w.r.t. a block b' that w_i was not, and therefore π^i_{sub} threatens a causal link that w_i did not, or a causal link to or from π^i_{sub} is threatened by b', is explicitly checked for before removing any ordering constraint. □

5 Results

The starting point for plan improvement is the best plan found by LAMA [Richter and Westphal, 2010, IPC 2011 version] in 30 minutes (called "LAMA@30" in the following). The compared methods are block decomposition plan optimisation (BDPO), plan neighbourhood graph search (PNGS), iterated bounded-cost search (IBCS) and letting LAMA continue (LAMA@60). Each is given 30 minutes and 3Gb memory per problem. IBCS uses the same bounded-cost search as local plan optimisation, applied to the whole problem.

We use problems from the satisficing track of the 2008 and 2011 IPC[1], data set 2-nd of the Genome Edit Distance (GED) domain [Haslum, 2011], and the Alarm Processing for Power Networks (APPN) domain [Haslum and Grastien, 2011].

Figure 1 shows the cumulative IPC quality score over time. Although this is a convenient summary, it does not convey the complete picture: it is strongly weighted by coverage (the score taking only the first plan found by LAMA for each problem is 404.77), and, more importantly, does not tell us anything about how much plan quality can actually be improved. To this end, we compare plan costs with the highest known lower bound for each problem (obtained by a variety of methods, including several optimal planners; cf. [Haslum, 2012]). For 163 out of 576 problems solved by LAMA@30, the best plan cost already matches the lower bound, which means no improvement is possible. Figure 3 shows for the remaining problems (excluding one in the PegSol domain) the cost of the best plan found by the plan optimisation methods, normalised to the gap between the initial plan cost and the highest lower bound. Table 1 provides a different summary of the same data. For the Genome Edit Distance problems, a non-optimal problem-specific algorithm (GRIMM) finds better solutions than all planners in most cases. However, BDPO and PNGS both find a better plan than GRIMM for 12 out of 156 problems, while LAMA@60 manages 6.

The results clearly show strong complementarity between the methods, both across domains and in time. Only IBCS does not outperform all other methods in any domain. In the PegSol domain, all plans but one found by LAMA@30 are optimal and no method improves on the cost of the last one. In OpenStacks and VisitAll, PNGS and BDPO find very few improvements, while LAMA finds more but smaller.

Most of the plan improvement achieved by LAMA is done early: During the first 30 minutes, it reduces plan cost, from that of the first plan it finds, by an average 17.9%, but the reduction over the next 30 minutes is only 0.8%. In contrast, BDPO, starting from the plan by LAMA@30, achieves an average plan cost reduction of 8%, and PNGS 7.3%. PNGS is also fast (99.6% of its improved plans are found in the first 5 minutes), and limited mainly by memory (it runs out of

[1] We exclude the CyberSec domain, which our current implementation is unable to deal with. For domains that appeared in both the 2008 and 2011 IPC, we use only the instances from 2011 that were new in that year.

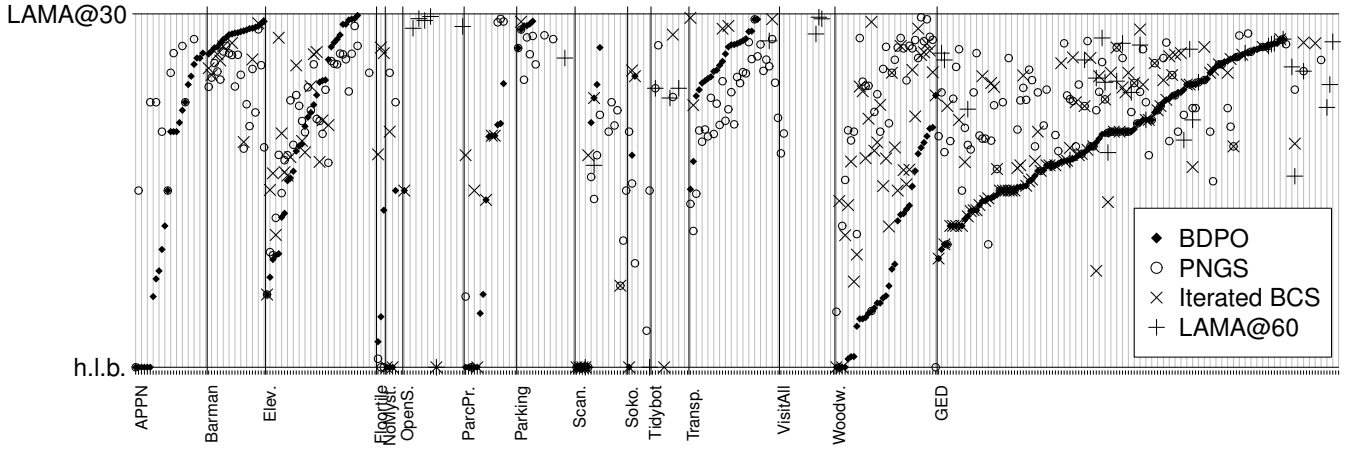

Figure 3: Cost of plans found by the four improvement methods applied to LAMA's best plan at 30 minutes. Costs are normalised to the interval between the initial plan and the highest known lower bound; problems where the initial plan is known to be optimal are excluded. Within each domain, problems are sorted by the cost of the best BDPO plan.

	BDPO			PNGS			IBCS			LAMA @60		
	$=$	$<$	$\star$	$=$	$<$	$\star$	$=$	$<$	$\star$	$=$	$<$	$\star$
APPN	100	76	24	24	0	4	8	0	0	8	0	0
Barman	0	0	0	100	100	0	0	0	0	0	0	0
Elevators	42	26	0	61	45	0	29	13	0	13	0	0
Floortile	0	0	0	100	100	67	0	0	0	0	0	0
NoMystery	83	33	50	33	17	0	50	0	33	17	0	0
OpenStacks	67	0	0	67	0	0	76	5	5	95	24	5
ParcPrinter	100	50	28	28	0	6	44	0	6	22	0	0
Parking	60	0	0	95	35	0	50	0	0	55	5	0
Scanalyzer	33	0	22	100	61	28	39	0	22	17	0	6
Sokoban	50	0	12	75	38	0	50	0	12	50	12	12
Tidybot	54	0	0	77	15	0	62	8	8	77	15	0
Transport	3	0	0	100	97	0	0	0	0	0	0	0
VisitAll	21	0	0	84	63	0	21	0	0	37	16	0
Woodworking	97	86	11	9	3	6	6	0	6	0	0	0
GED	83	30	0	38	8	0	43	4	0	18	4	0

Table 1: For each plan improvement method, the percentage of instances where it matches the best plan ($=$); finds a plan strictly better than any other method ($<$); and finds a plan that is known to be optimal, i.e., matched by the highest lower bound ($\star$). The percentage is of instances in each domain that are solved, but not solved optimally, by LAMA@30.

memory on 86% of problems, and out of time on 8.5%), while BDPO is limited mainly by time and continues to find better plans (though at a decreasing rate) even beyond 30 minutes.

6 Conclusions

Different planning and plan optimisation approaches have their strength at different time scales. This suggests that truly anytime planning – providing continuing improvement of plan quality at any time scale – is best achieved by a combination of methods.

The deordering step is critical to the success of local plan optimisation: 75% of subproblems for which we find an improved subplan do not correspond to consecutive parts of the original plan (61% looking only at the w part). The importance of block deordering, in addition to standard deordering, varies between domains: in some (e.g., Sokoban, GED) no deordering is possible without block decomposition, while in some (e.g., ParcPrinter, Woodworking) it adds very little.

Block decomposition plan optimisation can be viewed as a large neighbourhood local search, moving at each step from one valid plan to a new, better, valid plan. Results show strong complementarity, across problems and time scales, with anytime search and plan neighbourhood graph search, another local plan improvement method. Hence, a better strategy may be to combine them, either in a sequential portfolio (i.e., running BDPO on the result of PNGS, or vice versa), similar to portfolios of planners [Helmert *et al.*, 2011; Gerevini *et al.*, 2009], or, more interestingly, by extending the neighbourhood of the local search to include multiple types of plan improvement operations. Adaptive LNS [Ropke and Pisinger, 2006] uses a set of heuristic local improvement methods, and tries to learn, on-line, the strategy for selecting which to use. Our heuristic of restarting each iteration with the group of windowing rules that led to the last improvement can be seen as a simple adaptive strategy. Other aspects of the method, such as the merging strategy, grouping of windowing rules, subplanner, etc., can probably also be improved.

Local plan optimisation also highlights a promising use case for bounded-cost planning. This problem, i.e., quickly finding any plan with cost within a given, absolute, bound, has recently received interest in planning and search [Stern *et al.*, 2011; Thayer *et al.*, 2012]. Through block decomposition plan optimisation, any improvements made to bounded-cost planning algorithms translate directly into more efficient anytime planning.

Acknowledgment NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council via the ICT Centre of Excellence program.

References

[Ambite and Knoblock, 2001] J.L. Ambite and C.A. Knoblock. Planning by rewriting. *Journal of AI Research*, 15(1):207–261, 2001.

[Bäckström, 1998] C. Bäckström. Computational aspects of reordering plans. *Journal of AI Research*, 9:99–137, 1998.

[Balyo *et al.*, 2012] T. Balyo, R. Barták, and P. Surynek. On improving plan quality via local enhancements. In *Proc. 5th Symposium on Combinatorial Search (SoCS'12)*, pages 154–156, 2012.

[Chapman, 1987] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[Furcy, 2006] D. Furcy. ITSA*: Iterative tunneling search with A*. In *AAAI Workshop on Heuristic Search, Memory-Based Heuristics and their Applications*, pages 21–26, 2006.

[Gerevini *et al.*, 2009] A. Gerevini, A. Saetti, and M. Vallati. An automatically configurable portfolio-based planner with macro-actions: PbP. In *Proc. 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*, pages 350–353, 2009.

[Ghallab *et al.*, 2004] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers, 2004. ISBN: 1-55860-856-7.

[Godard *et al.*, 2005] D. Godard, P. Laborie, and W. Nuijten. Randomized large neighborhood search for cumulative scheduling. In *Proc. 15th International Conference on Automated Planning & Scheduling (ICAPS'05)*, pages 81–89, 2005.

[Haslum and Grastien, 2011] P. Haslum and A. Grastien. Diagnosis as planning: Two case studies. In *ICAPS'11 Scheduling and Planning Applications Workshop*, 2011.

[Haslum, 2011] P. Haslum. Computing genome edit distances using domain-independent planning. In *ICAPS'11 Scheduling and Planning Applications Workshop*, 2011.

[Haslum, 2012] P. Haslum. Incremental lower bounds for additive cost planning problems. In *Proc. 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*, pages 74–82, 2012.

[Helmert and Domshlak, 2009] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proc. 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*, 2009.

[Helmert *et al.*, 2011] M. Helmert, G. Röger, J. Seipp, E. Karpas, J. Hoffmann, E. Keyder, R. Nissim, S. Richter, and M. Westphal. Fast downward stone soup (planner abstract). In *7th International Planning Competition (IPC 2011), Deterministic Part*. `http://www.plg.inf.uc3m.es/ipc2011-deterministic`, 2011.

[Nakhost and Müller, 2010] H. Nakhost and M. Müller. Action elimination and plan neighborhood graph search: Two algorithms for plan improvement. In *Proc. 20th International Conference on Automated Planning and Scheduling (ICAPS'10)*, pages 121–128, 2010.

[Ratner and Pohl, 1986] D. Ratner and I. Pohl. Joint and LPA*: Combination of approximation and search. In *Proc. National Conference on AI (AAAI'86)*, pages 173–177, 1986.

[Richter and Westphal, 2010] S. Richter and M. Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of AI Research*, 39:127–177, 2010.

[Richter *et al.*, 2010] S. Richter, J. Thayer, and W. Ruml. The joy of forgetting: Faster anytime search via restarting. In *Proc. of the 20th International Conference on Automated Planning and Scheduling (ICAPS'10)*, pages 137–144, 2010.

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

[Shaw, 1998] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Proc. 4th International Conference on Principles and Practice of Constraint Programming (CP'98)*, pages 417–431, 1998.

[Siddiqui and Haslum, 2012] F.H. Siddiqui and P. Haslum. Block-structured plan deordering. In *AI 2012: Advances in Artificial Intelligence (Proc. 25th Australasian Joint Conference*, volume 7691 of *LNAI*, pages 803–814, 2012.

[Stern *et al.*, 2011] R. Stern, R. Puzis, and A. Felner. Potential-search: A bounded-cost search algorithm. In *Proc. 21st International Conference on Automated Planning and Scheduling (ICAPS'11)*, pages 234–241, 2011.

[Thayer *et al.*, 2012] J. Thayer, R. Stern, A. Felner, and W. Ruml. Faster bounded-cost search using inadmissible heuristics. In *Proc. 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*, pages 270–278, 2012.