University of Huddersfield Repository

Chrpa, Lukáš, McCluskey, T.L. and Osborne, Hugh

Determining Redundant Actions in Sequential Plans

Original Citation

Chrpa, Lukáš, McCluskey, T.L. and Osborne, Hugh (2012) Determining Redundant Actions in Sequential Plans. In: Tools with Artificial Intelligence (ICTAI), 2012 IEEE 24th International Conference on. IEEE, pp. 484-491. ISBN 9781479902279

This version is available at http://eprints.hud.ac.uk/id/eprint/16949/

The University Repository is a digital collection of the research output of the University, available on Open Access. Copyright and Moral Rights for the items on this site are retained by the individual author and/or other copyright owners. Users may access full items free of charge; copies of full text items generally can be reproduced, displayed or performed and given to third parties in any format or medium for personal research or study, educational or not-for-profit purposes without prior permission or charge, provided:

- The authors, title and full bibliographic details is credited in any copy;
- A hyperlink and/or URL is included for the original metadata page; and
- The content is not changed in any way.

For more information, including our policy and submission procedure, please contact the Repository Team at: E.mailbox@hud.ac.uk.

http://eprints.hud.ac.uk/

Determining Redundant Actions in Sequential Plans

Lukáš Chrpa, Thomas Leo McCluskey, Hugh Osborne
Knowledge Engineering and Intelligent Interfaces Research Group
School of Computing and Engineering
University of Huddersfield
Email: {l.chrpa, t.l.mccluskey, h.r.osborne}@hud.ac.uk

***Abstract*—Automated planning even in its simplest form, classical planning, is a computationally hard problem. With the increasing involvement of intelligent systems in everyday life there is a need for more and more advanced planning techniques able to solve planning problems in little (or real) time. However, planners designed to solve planning problems as fast as possible often provide solution plans of low quality. The quality of solution plans can be improved by their post-planning analysis by which redundant actions or optimizable subplans can be identified. In this paper, we present techniques for determining redundancy of actions in plans. Especially, we present techniques for efficient redundancy checking of pairs of inverse actions. These techniques are accompanied with necessary theoretical foundations and are also empirically evaluated using existing planning systems and standard planning benchmarks.**

***Keywords*-sequential plans; post-planning plan optimization; redundant actions; inverse actions**

I. Introduction

Automated planning [1] even in its simplest form, classical planning, is intractable (PSPACE-complete) [2]. Optimal planning, in classical planning finding the shortest plans, is generally harder than satisfying planning (i.e. finding any solution) [3], [4]. Therefore it is not surprising that post-planning plan optimization in generally NP-hard [5].

Nowadays intelligent systems are becoming ubiquitous which requires the development of more and more advanced planning systems which are able to operate in almost real-time. For instance, sometimes it is necessary to provide a solution very quickly to avoid imminent danger for a robot and prevent significant financial losses. There are planning engines which focus on speed of the planning process rather than the quality of solutions. A good example is LPG [6], a planner which performs a greedy local search in a Planning Graph [7]. FF [8] and LAMA [9] should also be mentioned which use weighted A$*$ with an inadmissible but well informed heuristic. In optimal planning, there is GAMER [10], a planner based on exploring Binary Decision Diagrams.

In this paper we focus on determining redundant actions in plans which can then be safely removed from the plans. This work is an extension of our previous work [11] where we presented some basic foundations related to determining redundant actions. Here, we present an efficient method for redundancy checking of pairs of inverse actions which significantly reduces the necessity for re-checking the pairs for redundancy once again. Moreover, we present a technique for determining 'grouped' pairs of inverse actions which might be redundant only all together (and not on their own). Despite the fact that dealing with pairs of 'grouped' pairs of inverse actions does not cover all possibly redundant actions we believe that our approach in the most cases is able to determine most of these redundant actions. The presented techniques for determining redundant actions are accompanied by necessary theoretical foundations and are also empirically evaluated using existing planning systems, which successfully competed at the International Planning Competition (IPC)[1], and standard planning benchmarks. Our approach can be understood as a supporting technique for the state-of-the-art plan optimization techniques (Neighborhood Graph [12], AIRS [13] — see Section II) rather than their competitor. This is because our approach can identify and remove redundant actions in a very short time (as discussed in Section IX) and therefore enable the state-of-the-art techniques to be more efficient.

II. Related Works

Approaches to plan optimization using genetic programming are promising, though the relation between plan generation time and optimization time is unclear [14]. The recent related work [12] proposes a Neighborhood Graph search technique for plan optimization. It expands a limited number of nodes around each state along the plan to a produce a Neighborhood Graph and then, by applying Dijsktra's algorithm, it finds a shortest path within the neighborhood. The most recent work [13] presents AIRS, an algorithm for plan optimization. AIRS heuristically investigates whether two states along the plan might be closer (i.e., a smaller number of actions is needed to move from one state to the other one). If such states are found then optimal or nearly-optimal planner is applied to re-plan. However, these methods are restricted to local optimality and do not exploit the information within the plan structure (e.g some actions might lie far from each other in a plan but can be adjacent in some permutation of the plan).

III. Preliminaries

Classical planning (in state space) deals with finding a sequence of actions transforming the static, deterministic and fully observable environment from some initial state to a desired goal state [1].

In the set-theoretic representation **atoms**, which describe the environment, are propositions. **States** are defined as sets of

[1] http://ipc.icaps-conference.org

propositions. **Actions** are specified via sets of atoms specifying their preconditions, negative and positive effects (i.e., $a = (pre(a), eff^-(a), eff^+(a))$). An action a is **applicable** in a state s iff $pre(a) \subseteq s$. Application of a in s results in a state $(s \setminus eff^-(a)) \cup eff^+(a)$ if a is applicable in s, otherwise the result of the application is undefined.

In the classical representation atoms are predicates. A **Planning operator** $o = (name(o), pre(o), eff^-(o), eff^+(o))$ is a generalized action (i.e. an action is a grounded instance of the operator), where $name(o) = op\_name(x_1, \ldots, x_k)$ (*op_name* is an unique operator name and $x_1, \ldots x_k$ are variable symbols (arguments) appearing in the operator) and $pre(o), eff^-(o)$ and $eff^+(o)$ are sets of (unground) predicates. The set-theoretic representation can be obtained from the classical representation by grounding.

A **planning domain** is specified via sets of predicates and planning operators (alternatively propositions and actions). A **planning problem** is specified via a planning domain, initial state and set of goal atoms. A **plan** is a sequence of actions. A plan is a **solution** of a planning problem if and only if a consecutive application of the actions in the plan (starting in the initial state) results in a state, where all the goal atoms are satisfied. A solution π of a given problem is **optimal** if for any solution π' of the given problem $|\pi| \leq |\pi'|$.

In sequential classical (STRIPS) planning it is not necessary for an atom to be present in both negative and positive effects of an action or operator because applying the action (operator) always results in a state where the atom is present. On the other hand, in parallel planning keeping an atom in both negative and positive effects might be useful because it prevents unwanted parallel execution of certain actions. Similarly, in sequential classical planning it is not necessary to have an atom in both the precondition and positive effects of an action because the atom must be already present before the action can be applied. Henceforth, we will assume that every action or planning operator a satisfies the following conditions:

$$eff^-(a) \cap eff^+(a) = \emptyset \quad (1)$$

$$pre(a) \cap eff^+(a) = \emptyset \quad (2)$$

In sequential classical planning we can easily show that every planning domain has its equivalent which follows constraints (1) and (2). If (1) is not satisfied for an action a, then a is modified by removing $eff^-(a) \cap eff^+(a)$ from $eff^-(a)$. Similarly, if (2) is not satisfied, then a is modified by removing $pre(a) \cap eff^+(a)$ from $eff^+(a)$. Note that if both conditions are violated then the action must be modified to satisfy first (1) and then (2).

IV. Redundant Actions

In non-optimal planning, solution plans may contain actions which are not necessary and can be omitted. We call these actions redundant actions when their removal from a solution plan still results in a solution plan. This is formalized below.

Definition 1. *Let Π be a planning problem and π its solution plan. We say that actions $a_{x_1}, \ldots, a_{x_k} \in \pi$ are **redundant** in π if and only if $\pi' = \pi \setminus \{a_{x_1}, \ldots, a_{x_k}\}$ is a solution plan of Π. Henceforth, $\{a_{x_1}, \ldots, a_{x_k}\}$ is denoted as a set of redundant actions in π.* ■

Remark 1. *Note that if a set of actions A_x is redundant in π, then a set of actions A_y such that $A_y \subset A_x$ might not be redundant in π. In literature [12], a plan π' obtained by removing redundant actions from π is called a **reduction** of π.*

Definition 2. *Let Π be a planning problem and π its solution plan. We say that a set of redundant actions A_x in π is **maximal** if and only if for every set of redundant actions A_z in π holds that $|A_x| \geq |A_z|$.* ■

Finding a maximal set of redundant actions is desirable for optimizing plans. However, it has been proven that the problem of determining the existence of π', a reduction of π, such that $|\pi'| \leq k$ for a given constant k is NP-hard [12]. Consequently, finding a minimal reduction, i.e., a shortest plan π' which is a reduction of π, is NP-hard as well. This is summarized in the following theorem.

Theorem 1. *Determining a maximal set of redundant actions A_x in a given plan π is NP-hard.*

Proof: This is analogous to the problem of finding a minimal reduction which is NP-hard [12]. ■

Despite the NP-hardness of the problem of determining a maximal set of redundant actions in many cases a lot of redundant actions can be determined in polynomial time. We will therefore focus on situations where redundant actions (e.g. pairs of inverse actions) can be identified easily (in polynomial time) which, we believe, reveals most of the redundant actions.

V. Action Dependencies

Actions ordered in plans influence each other. An action achieves atoms which are preconditions for some actions but on the other hand 'clobber' atoms required by other actions [15]. Recalling constraints (1), (2) (Section III) we can see that actions violating constraint (1) are 'false clobberers' while actions violating constraint (2) are 'false achievers'. Inspired by the meaning of causal links known in plan-space planning, we can identify dependencies between the actions in a given sequence (plan) in terms of which actions achieve atoms to other actions that need them as their precondition [16]. The formal definition follows.

Definition 3. *Let $\langle a_1, \ldots, a_n \rangle$ be an ordered sequence of actions. An action a_j is **directly dependent** on an action a_i (denoted as $a_i \rightarrow a_j$) if and only if $i < j$, $(eff^+(a_i) \cap pre(a_j)) \neq \emptyset$ and $(eff^+(a_i) \cap pre(a_j)) \not\subseteq \bigcup_{t=i+1}^{j-1} eff^+(a_t)$.*
*An action a_j is **dependent** on an action a_i if and only if $a_i \rightarrow^* a_j$ where $\rightarrow^*$ is the reflexive transitive closure of the relation $\rightarrow$.*
$\nrightarrow$ denotes that actions are not directly dependent and $\nrightarrow^*$ denotes that actions are not dependent. ■

To obtain a complete model of these relations in solution plans we have to use two special actions: $a_0 = (\emptyset, \emptyset, I)$ (I is an initial state for a given problem) and $a_g = (G, \emptyset, \emptyset)$ (G is a set of goal atoms for a given problem). Relations of action direct dependencies and dependencies can be found in $\mathcal{O}(n^2)$ steps (n is a length of a plan) [16].

Given the action dependence relation it is easy to identify which actions do not contribute to the goal (i.e. the special goal action a_g is not dependent on them). Such actions are redundant. This is formalized in the following proposition (for the proof, see [11]).

Proposition 1. *Let $\pi = \langle a_1, \ldots, a_n \rangle$ be a solution plan of a planning problem Π and $a_g = \{G, \emptyset, \emptyset\}$ (G is a set of goal atoms in Π) be an action. Let $A^- = \{a_i \mid a_i \in \pi, a_i \nrightarrow^* a_g\}$ be a set of actions on which the goal is not dependent. Then all actions in A^- are redundant in π.*

VI. Inverse Actions

In planning, action effects are often reversible. For example, picking a block up from the table can be reversed by putting the block down on the table. Informally, if an application of an action a in a state s results in a state s' and an application of some action a' in the state s' results back in the state s or its subset, then a' reverts the effects of the action a. In other words, the action a' is inverse to the action a. The formal definition follows.

Definition 4. *We say that action a and a' are **inverse** if and only if a consecutive application of a and a' in any state s where a is applicable results in a state s' such that $s' \subseteq s$.* ■

The above definition might look too general. Basically, actions with interchanged positive and negative effect are inverse if also their preconditions contain all atoms presented in their negative effects. This is formalized in the following lemma.

Lemma 1. *Let a, a' be actions. If $\mathit{eff}^-(a) \subseteq \mathit{pre}(a)$, $\mathit{eff}^-(a') \subseteq \mathit{pre}(a')$, $\mathit{eff}^+(a) = \mathit{eff}^-(a')$ and $\mathit{eff}^-(a) = \mathit{eff}^+(a')$ then the actions a and a' are inverse.*

Proof: Without loss of generality we assume that actions a and a' can be consecutively applied in a state s. Then the result of such an application is: $(((s \setminus \mathit{eff}^-(a)) \cup \mathit{eff}^+(a)) \setminus \mathit{eff}^-(a')) \cup \mathit{eff}^+(a')$. From the assumption we get: $(((s \setminus \mathit{eff}^-(a)) \cup \mathit{eff}^+(a)) \setminus \mathit{eff}^+(a)) \cup \mathit{eff}^-(a) \subseteq s$ (from the assumption can be easily obtained that $\mathit{eff}^-(a) \subseteq s$). The proof is done analogously for a consecutive application of a' and a in some state s. ■

Pairs of inverse actions, which are potentially redundant, might influence each other in plans. Therefore, it is useful to analyze positions of pairs of inverse actions in plans because then we can identify potential interferences between particular pairs of inverse actions. Informally, by interferences we mean situations when some pair of inverse actions cannot be removed from a plan before some other pair of inverse actions is removed. This is discussed more thoroughly in Section VII. We can formally define four ways in which pairs of inverse actions can be placed within a given plan (for illustration, see Figure 1).

Definition 5. *Let x, x', y and y' be indices such that $x < x'$, $y < y'$ and $x \leq y$. Let π be some plan and $a_x, a_{x'}, a_y, a_{y'} \in \pi$ be actions such that $(a_x, a_{x'})$ and $(a_y, a_{y'})$ are pairs of inverse actions. For indices i, j such that actions $a_i, a_j \in \pi$ it holds that $i < j$ if and only if a_i is applied before a_j in π. We say that:*

1) *the pairs $(a_x, a_{x'})$ and $(a_y, a_{y'})$ are **independent** if $x' < y$,*
2) *the pairs $(a_x, a_{x'})$ and $(a_y, a_{y'})$ are **nested** if $y' < x'$,*
3) *the pairs $(a_x, a_{x'})$ and $(a_y, a_{y'})$ are **interleaving** if $y < x'$ and $x' < y'$,*
4) *the pairs $(a_x, a_{x'})$ and $(a_y, a_{y'})$ are **shared** if $x = y$ or $x' = y$ or $x' = y'$.*

■

VII. Identifying Redundant Inverse Actions in Plans

Inverse actions are obviously redundant if they are executed successively. However, inverse actions might not be necessarily adjacent in plans but still be redundant. For example, if some plan contains a sequence `pickup(a)` (pick up a block a from the table), `move(r,l1,l2)` (move a robot r from a location $l1$ to $l2$), `putdown(a)` (put the block a on the table), then the inverse actions `pickup(a)` and `putdown(a)` are redundant because the action `move(r,l1,l2)` is not influenced by any of them. On the other hand, if some plan consists a sequence `pickup(a)`, `paint(a,red)` (paint a block a by red paint), `putdown(a)`, then the inverse actions `pickup(a)` and `putdown(a)` are not redundant because the action `paint(a,red)` requires the block a to be held by the robotic hand. Distinguishing between these cases can be done by analyzing action dependencies in plans. In the following proposition we show that if no action placed between inverse actions a and a' in a given plan is directly dependent on a or 'clobbers' an atom given back by a', then actions a and a' are redundant in the given plan.

Proposition 2. *Let $\pi = \langle a_1, \ldots, a_n \rangle$ be a plan. Let $a_i, a_j \in \pi, i < j$ be inverse actions. If there is no action a_k $(i < k < j)$ such that $a_i \rightarrow a_k$ or $\mathit{eff}^-(a_k) \cap \mathit{eff}^+(a_j) \neq \emptyset$, then a_i and a_j are redundant in π.*

Proof: See [11]. ■

The previous proposition gives an insight into how we can detect redundant inverse actions. For a pair of inverse actions we need (at worst) $\mathcal{O}(l)$ steps (l is the number of actions placed between the inverse ones) to decide whether they are redundant or not. A naive approach for deciding redundant actions and eliminating them from a plan [11] works in the following way:

1) Construct action (direct) dependencies and identify pairs of inverse actions

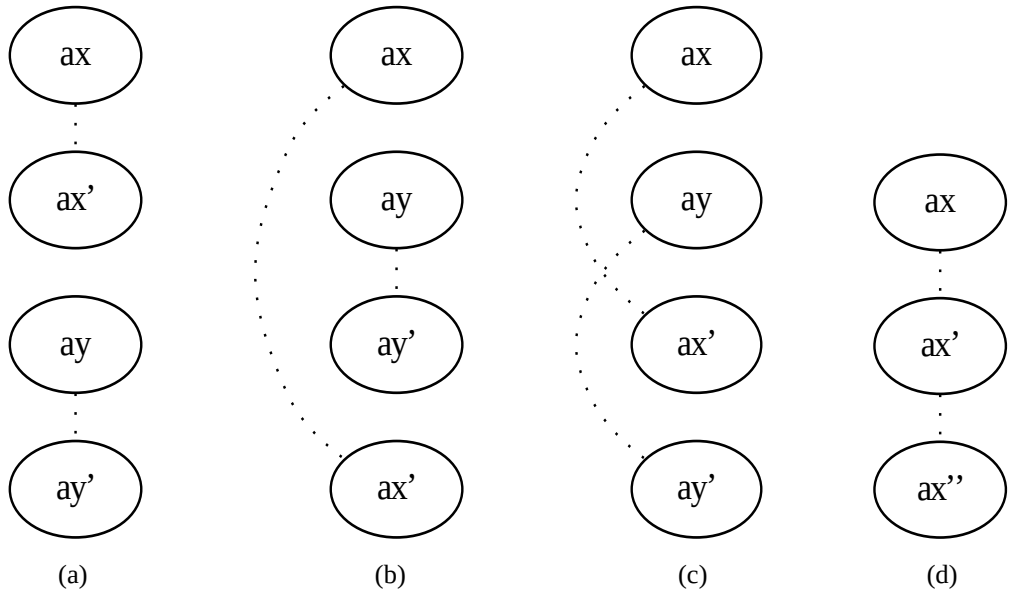

Fig. 1. Ways of placement of inverse actions in plans: (a) independent, (b) nested, (c) interleaving, (d) shared

2) For each pair of inverse actions decide whether they are redundant (according to Proposition 2).
3) Remove redundant actions from the plan. If no action has been removed then terminate otherwise go to step 1).

As mentioned before step 1) can be done in $\mathcal{O}(n^2)$ steps (n is the length of the plan). Step 2) can be done in at worst $\mathcal{O}(kl)$ steps where k is the number of pairs of inverse actions and l is the highest number of actions placed between any pair of inverse actions. In the worst case we may repeat the whole process k times, hence the (worst case) complexity is $\mathcal{O}(k(n^2 + kl))$.

For example, if some plan contains a sequence `unstack(a,b)`, `putdown(a)`, `pickup(a)`, `stack(a,b)` we can see that pairs of inverse actions (`unstack(a,b)`, `stack(a,b)`) and (`putdown(a)`, `pickup(a)`) are nested. Using common sense we have to remove the inner pair (`putdown(a)`, `pickup(a)`) before trying to remove the outer pair (`unstack(a,b)`, `stack(a,b)`). However, using the above approach we may try to remove the outer pair at first which is not possible since `putdown(a)` is directly dependent on `unstack(a,b)`. We have to therefore try again to remove the outer pair in the following iterations which will succeed if and only if the inner pair is removed.

The above example gives an insight into in which order we should check pairs of inverse actions for redundancy. Straightforwardly, the order in which independent or shared pairs of inverse actions are checked for redundancy is not crucial, i.e., if no action from one pair of inverse actions lies between an independent pair of inverse actions (situation (a) in Figure 1) the first pair cannot influence the results of the redundancy check of the second pair. For shared pairs of inverse actions it is obvious that at most one of the pairs can be removed because after that only one action remains (situation (d) in Figure 1). Nested pairs of inverse actions (situation (b) in Figure 1) should be checked for redundancy, as indicated in the example above, in such a way that the inner pair of inverse actions is checked before the outer pair. Interleaving pairs of inverse actions (situation (c) in Figure 1) are the most 'tricky' case because we can find two contradictory situations where one of the orderings is wrong while the other is correct and vice versa. Let $(a_x, a_{x'})$ and $(a_y, a_{y'})$ be interleaving pairs of inverse actions such that $x < y < x' < y'$. If $a_x \rightarrow a_y$ and another action placed between a_x and $a_{x'}$ violates the conditions in Proposition 2, then $(a_x, a_{x'})$ can be removed only after $(a_y, a_{y'})$ is removed. On the other hand, if $a_y \rightarrow a_{x'}$ and another action placed between a_y and $a_{y'}$ violates the conditions in Proposition 2, then $(a_y, a_{y'})$ can be removed only after $(a_x, a_{x'})$ is removed. This shows that in general we cannot find an ordering in which we check pairs of inverse actions for redundancy.

Despite the above findings the pairs of inverse actions can be efficiently ordered prior to their redundancy check. If $(a_x, a_{x'})$ and $(a_y, a_{y'})$ are pairs of inverse actions then $(a_x, a_{x'})$ will be checked for redundancy before $(a_y, a_{y'})$ if $x > y$ (i.e. a_x is applied after a_y in a given plan). The formal definition follows.

Definition 6. *Let π be a plan such that if $a_i, a_j \in \pi$ and $i < j$ then and only then a_i is applied before a_j in π. We define a relation $\prec$ between pairs of inverse actions such that $(a_y, a_{y'})$ $(a_x, a_{x'}) \prec (a_y, a_{y'})$ if and only if $x \geq y$.* ■

Using $\prec$ for ordering pairs of inverse actions, however, does not guarantee that some pairs of inverse actions do not have to be re-checked for redundancy. On the other hand, we can identify under which conditions we do not have to perform re-checking for redundancy, i.e., every pair of inverse actions is checked for redundancy at most once. These conditions draw

from the above example where we showed that when dealing with interleaving pairs of inverse actions there is generally no given order in which we can check the pairs for redundancy. This is formalized in the following theorem.

Theorem 2. *Let $\langle(a_{x_1}, a_{x'_1}), \ldots, (a_{x_m}, a_{x'_m})\rangle$ be an ordered sequence of pairs of inverse actions (all actions are present in a given plan π) such that $\forall i, j : i < j, (a_{x_i}, a_{x'_i}) \prec (a_{x_j}, a_{x'_j})$. If pairs of inverse actions are checked for redundancy in sequence, then re-checking these pairs will only reveal new information (i.e. mark a pair of inverse actions as redundant) if and only if there are interleaving pairs of inverse actions $(a_{x_i}, a_{x'_i})$, $(a_{x_j}, a_{x'_j})$ such that $(a_{x_i}, a_{x'_i}) \prec (a_{x_j}, a_{x'_j})$ and there is just one k such that $x_i < k < x'_i, a_k \rightarrow a_{x_i} \vee$ $\mathit{eff}^-(a_k) \cap \mathit{eff}^+(a_{x'_i}) \neq \emptyset$ and $k = x'_j$.*

Proof: Without loss of generality let $(a_{x_p}, a_{x'_p})$, $(a_{x_q}, a_{x'_q})$ $(1 \leq p < q \leq m)$ be pairs of inverse actions. From the assumption we get that $(a_{x_p}, a_{x'_p}) \prec (a_{x_q}, a_{x'_q})$ and therefore $(a_{x_p}, a_{x'_p})$ is checked for redundancy before $(a_{x_q}, a_{x'_q})$. There are four situations which can occur:

1) $(a_{x_p}, a_{x'_p})$ and $(a_{x_q}, a_{x'_q})$ are *independent*. Proposition 2 says that only actions placed between a_{x_p} and $a_{x'_p}$ or a_{x_q} and $a_{x'_q}$ respectively can influence the decision as to whether a_{x_p} and $a_{x'_p}$ or a_{x_q} and $a_{x'_q}$ respectively are redundant. According to Definition 5 none of the actions a_{x_q}, $a_{x'_q}$ is placed between a_{x_p}, $a_{x'_p}$ or vice versa. Hence the result of the redundancy check for $(a_{x_p}, a_{x'_p})$ cannot be influenced by the result of the redundancy check for $(a_{x_q}, a_{x'_q})$.
2) $(a_{x_p}, a_{x'_p})$ and $(a_{x_q}, a_{x'_q})$ are *shared*. According to Definition 5 if either of the pair is redundant and removed then the other one is no longer a pair of inverse actions (it consists of only one action), e.g if $x'_q = x_p$ and $(a_{x_p}, a_{x'_p})$ is redundant and going to be removed, then only the action a_{x_q} remains in the other 'pair'. Hence, if $(a_{x_q}, a_{x'_q})$ is redundant and going to be removed, then $(a_{x_p}, a_{x'_p})$ is not redundant (after $(a_{x_q}, a_{x'_q})$ is removed).
3) $(a_{x_p}, a_{x'_p})$ and $(a_{x_q}, a_{x'_q})$ are *nested*. Given Definition 5 and the relation $\prec$ we can see that the pair $(a_{x_p}, a_{x'_p})$ is placed in between the pair $(a_{x_q}, a_{x'_q})$. From this and Proposition 2 we can see that the result of the redundancy check for $(a_{x_p}, a_{x'_p})$ cannot be influenced by the result of the redundancy check for $(a_{x_q}, a_{x'_q})$. Note that the result of the redundancy check for $(a_{x_q}, a_{x'_q})$ may be influenced by the result of the redundancy check for $(a_{x_p}, a_{x'_p})$, therefore it is necessary to check for redundancy in the order given by $\prec$.
4) $(a_{x_p}, a_{x'_p})$ and $(a_{x_q}, a_{x'_q})$ are *interleaving*. Given Definition 5 and the relation $\prec$ we can see that a_{x_p} is placed in between the pair $(a_{x_q}, a_{x'_q})$ and $a_{x'_q}$ in between the pair $(a_{x_p}, a_{x'_p})$. The result of the redundancy check for $(a_{x_p}, a_{x'_p})$ may be influenced by the result of redundancy check for $(a_{x_q}, a_{x'_q})$ if and only if $a_{x'_q}$ is the only action which prevents the redundancy check for $(a_{x_p}, a_{x'_p})$ to be successful (see Proposition 2). However, such a specific case is reflected in the assumption of the theorem and in other cases the result of the redundancy check for $(a_{x_p}, a_{x'_p})$ cannot be influenced by the result of the redundancy check for $(a_{x_q}, a_{x'_q})$.

In summary, we do not have to re-check the pair $(a_{x_p}, a_{x'_p})$ after the pair $(a_{x_q}, a_{x'_q})$ is found to be redundant. ■

Taking into account ordering pairs of inverse actions given by the relation $\prec$, then the anticipated complexity of determining redundancy of these pair of actions is $\mathcal{O}(n^2 + kl)$ (n is the length of the plan, k is the number of pairs of inverse actions and l is the highest number of actions placed between any pair of inverse actions). This reflects the nonnecessity for re-checking some pairs of inverse actions for redundancy, however, in an unlikely case where some interleaving pairs of inverse actions violate the assumption in Theorem 2 we have to re-check remaining pairs of inverse actions for redundancy.

A. Grouping Nested Inverse Actions

Consider an example where `pickup(a)`, `stack(a,b)`, `pickup(c)`, `stack(c,d)`, `unstack(a,b)`, `putdown(a)` is a subsequence of some plan. We can identify nested pairs of inverse actions `stack(a,b)`, `unstack(a,b)` and `pickup(a)`, `putdown(a)`. However, If `pickup(c)` is essential in the plan, then the pair `stack(a,b)`, `unstack(a,b)` cannot be removed according to Proposition 2 because `pickup(c)` is directly dependent on `stack(a,b)` (`stack(a,b)` frees the robotic hand for `pickup(c)`). On closer inspection, we can find out that considered together the actions `stack(a,b)`, `unstack(a,b)` and `pickup(a)`, `putdown(a)` are redundant in the plan but when considered on their own (as a pair) the actions `stack(a,b)`, `unstack(a,b)` are not redundant in the plan. Therefore it seems to be useful to extend Proposition 2 for nested pairs of inverse actions.

The idea of 'grouping' nested pairs of inverse actions is based upon an observation (indicated in the example above) that sometimes the whole group of nested pairs of inverse actions is redundant but a single pair of inverse actions is not redundant. Let $(a_x, a_{x'})$ and $(a_y, a_{y'})$ be nested pairs of inverse actions where $y > x$ and $a_x \rightarrow a_y$. If some action a_z $(y < z < y')$ is directly dependent on a_y then we cannot remove either $(a_y, a_{y'})$ or $(a_x, a_{x'})$. It might describe a situation where a_x removes some atoms which a_y puts back for a_z. Removing both a_x and a_y therefore might not cause inapplicability of a_z. We formalize this in the following proposition.

Proposition 3. *Let $(a_x, a_{x'})$ and $(a_y, a_{y'})$ be nested pairs of inverse actions in some plan π such that $x < y < y' < x'$. If all the following hold:*

1) *for all k such that $x < k < x', k \neq y, k \neq y'$ we have $a_x \nrightarrow a_k$ and $\mathit{eff}^-(a_k) \cap \mathit{eff}^+(a_{x'}) = \emptyset$*
2) *for all k such that $x < k < y$ we have $\mathit{eff}^-(a_k) \cap \mathit{pre}(a_x) = \emptyset$*
3) *for all k such that $y < k < y'$ we have $\mathit{eff}^-(a_k) \cap \mathit{eff}^+(a_{y'}) = \emptyset$*

4) *for all* k *such that* $y < k < y'$ *and* $a_y \rightarrow a_k$ *we have* $\mathit{pre}(a_k) \cap \mathit{eff}^+(a_y) \subseteq \mathit{pre}(a_x)$
5) *for all* k *such that* $y' < k < x'$ *and* $a_{y'} \rightarrow a_k$ *we have* $\mathit{eff}^+(a_x) \cap \mathit{eff}^+(a_{y'}) \cap \mathit{pre}(a_k) = \emptyset$

then the actions $a_x, a_{x'}, a_y$ *and* $a_{y'}$ *are redundant in* π.

Proof: Assume that the actions $a_x, a_{x'}, a_y$ and $a_{y'}$ are removed from π, a solution of some problem. Then we have to show that $\pi \setminus \{a_x, a_{x'}, a_y, a_{y'}\}$ is still a solution of the problem. We will analyze all situations with respect to the position of some action a_k.

- $k < x$ — Straightforwardly, applicability or outcome of a_k is not affected by removing actions positioned after it.
- $x < k < y$ — Given condition 1), then according to Proposition 2 a_k is not affected by removing a_x and $a_{x'}$. Removing a_y and $a_{y'}$ does not affect a_k because a_k is placed before them.
- $y < k < y'$ — Given condition 1), then according to Proposition 2 a_k is not affected by removing a_x and $a_{x'}$. Condition 3) ensures that atoms present in the positive effects of $a_{y'}$ are not removed because from Definition 4 it can be seen that these atoms must be present before a_y is executed. If no action placed in between a_y and $a_{y'}$ removes some of these atoms, then they will remain valid for actions placed after $a_{y'}$. If $a_y \nrightarrow a_k$, then according to Proposition 2 a_k is not affected by removing a_y and $a_{y'}$. If $a_y \rightarrow a_k$, then condition 4) says that atoms achieved by a_y to a_k are already present before a_x is executed (the atoms are in its precondition). Condition 2) says that none of these atoms can be removed by actions positioned in between a_x and a_y. Hence, a_k is not affected by removing the actions $a_x, a_{x'}, a_y$ and $a_{y'}$.
- $y' < k < x'$ — According to Proposition 2 a_k is not affected by removing a_y and $a_{y'}$. Condition 5) says that a_k cannot become directly dependent on a_x after $a_{y'}$ is removed. This together with condition 1) results in the fact that a_k is also not affected by removing a_x and $a_{x'}$.
- $k > x'$ — Conditions 1) and 3) ensures that atoms present before application of a_x or a_y remain valid even if the actions $a_x, a_{x'}, a_y$ and $a_{y'}$ are removed. Hence, a_k cannot be affected.

In summary, we have shown that the remaining actions in the plan are still applicable and by taking into account also a special goal action (having all goal atoms in its precondition) we can find out that $\pi \setminus \{a_x, a_{x'}, a_y, a_{y'}\}$ is a solution of the given problem. ■

Even though the above proposition deals only with two nested pairs of inverse actions, we believe that the proposition can be generalized for more pairs. On the other hand, in the most of planning domains it is not necessary to take into account more than two such pairs.

VIII. Implementation Details

A high-level design of our post-planning plan optimization algorithm is depicted in Algorithm 1. The optimization techniques discussed in this paper are applied from the easiest one to the most difficult one. This is because actions marked for removal by easier techniques do not have to be considered by more difficult techniques. This is obviously more efficient. One technical detail which might not be obvious from the theory given in the previous sections is in handling marked (redundant) actions which are going to be removed. Marked actions should be treated as actions which are no longer in the plan. However, this might cause changes in direct dependency relations. To avoid recomputation of action (direct) dependencies every time some actions have been marked we can use the following observation. Let a_x and $a_{x'}$ be a pair of inverse actions and a_k an action placed in between them. If a_k has been marked, then an action a_l placed in between a_k and $a_{x'}$ may become directly dependent on a_x if $\mathit{eff}^+(a_x) \cap \mathit{eff}^+(a_k) \cap \mathit{pre}(a_l) \neq \emptyset$. This follows directly from Definition 3.

Algorithm 1 High-level design of our plan optimization algorithm

1: Determine action direct dependencies and dependencies
2: Determine pairs of inverse actions and sort them with respect to $\prec$ (see Definition 6)
3: Mark such actions on which the goal is not dependent
4: **repeat**
5: Check pairs of inverse actions for redundancy and mark actions if redundant
6: **until** No action has been marked or none of the interleaving pairs of inverse actions violates the conditions of Theorem 2
7: Check grouped nested pairs of inverse actions for redundancy and mark actions if redundant
8: Remove marked actions from the plan

For illustration, the algorithm for checking redundancy of pairs of inverse actions is depicted in Algorithm 2. Clearly, we cannot remove the pair if one of its action has already been marked (Line 2) since it refers to shared pairs of inverse actions where one of them has been marked for removal. Following the observation mentioned above, $atoms$ (Line 9) stands for atoms which are created by a_x (the first action in the pair) and at least one of the marked actions. In other words, a_x might become an achiever for some other action and therefore the other action might become directly dependent on a_x. This is verified in Line 11, where besides verifying the conditions of Proposition 2 we have to check whether a precondition of a given action contains an atom (or atoms) from $atoms$. If so, then removing some actions in between a_x and the given action would result in the given action becoming directly dependent on a_x.

The same philosophy can be use when implementing the other algorithms (Lines 6 and 7 in Algorithm 1) if we do not want to recompute the direct dependency relation each time we mark some action(s) for removal.

Domain	no. of problems	original	optimized	factor	time	goal not dep.	inverse	grouped inverse	re-checks
LPG									
Depots	22	1099	1029	6.4%	0.23s	0	68	8	0
Driverlog	20	1477	1251	15.3%	0.46s	0	226	0	0
Gold-miner	30	1370	1149	16.1%	0.43s	1	220	0	0
Matching-BW	11	909	805	11.4%	0.52s	0	40	64	0
Storage	27	5818	1676	71.2%	2.43s	0	3986	156	1
Zeno	20	958	946	1.3%	0.17s	0	12	0	0
Metric-FF									
Depots	20	968	884	8.7%	0.28s	4	60	20	1
Driverlog	17	617	599	2.9%	0.23s	0	18	0	0
Gold-miner	28	738	738	0.0%	0.49s	0	0	0	0
Matching-BW	13	948	880	7.2%	0.40s	0	4	64	0
Storage	18	281	281	0.0%	0.16s	0	0	0	0
Zeno	20	632	631	0.2%	0.13s	1	0	0	0
LAMA									
Depots	22	1310	1153	12.0%	0.48s	5	116	36	0
Driverlog	20	1315	1183	10.0%	0.51s	0	132	0	0
Gold-miner	30	2798	2798	0.0%	1.64s	0	0	0	0
Matching-BW	16	1512	1204	20.4%	0.59s	0	112	196	0
Storage	19	496	450	9.3%	0.28s	0	38	8	0
Zeno	20	692	686	0.9%	0.44s	6	0	0	0

TABLE I
EXPERIMENTAL RESULTS SHOW THE PERFORMANCE OF OUR PLAN OPTIMIZATION APPROACH.

Algorithm 2 Algorithm for checking pairs of inverse actions for redundancy

for all $(a_x, a_{x'})$ in the sequence of pairs of inverse actions ordered by $\prec$ **do**
  if a_x or $a_{x'}$ is marked **then**
    continue
  end if
  $viol := false$
  $atoms := \{\}$
  for $k := x + 1$ to $x' - 1$ **do**
    if a_k is marked **then**
      $atoms := atoms \cup (\mathit{eff}^+(a_x) \cap \mathit{eff}^+(a_k))$
    else
      $viol := a_k \rightarrow a_x \vee \mathit{eff}^-(a_k) \cap \mathit{eff}^+(a_{x'}) \vee atoms \cap \mathit{pre}(a_k) \neq \emptyset$
    end if
    if $viol$ **then**
      break
    end if
  end for
  if $\neg viol$ **then**
    mark both a_x and $a_{x'}$
  end if
end for

IX. EXPERIMENTAL EVALUATION

For evaluation purposes we chose several IPC benchmarks (typed strips), namely Depots, Zeno, DriverLog, Matching-BlockWorld, Gold-Miner and Storage. As benchmarking planners we chose Metric-FF [17], LAMA 2011 [9] and LPG-td [6]. All the planners successfully competed in the IPC. LPG was optimized for speed and ran with a random seed set to 12345. LAMA was set to use lazy greedy best first search accommodated by Landmark and FF heuristics. Metric-FF ran in default settings. Only problems solved by the planners within 1000s were considered.

Our method for plan optimization through looking for redundant actions is implemented in C++. The method support typed STRIPS representation in PDDL [18]. The experiments were performed on Intel i5 2.8 GHz, 8GB RAM, where Ubuntu Linux was used for running planners and Windows 7 for running our method.

Cumulative results (aggregated results of all problems considered in a particular domain) are presented in Table I. "Factor" is the percentage by which the plans were shortened (optimized) by our approach. "Time" is the time our method needed for the optimization of all problems considered in a given domain. "Re-checks" is the number of times the conditions of Theorem 2 were violated, i.e., how many times we had to re-check pairs of inverse actions for redundancy. The best overall results were achieved for LPG, especially in the Storage domain the plans were shortened by more than 70%! LPG is a planner based on greedy local search techniques and it is therefore to be expected that the solutions are often obtained in a little time but their quality tends to be low. In many cases (except the Zeno domain), these solutions can be significantly improved by our method focused on eliminating redundant inverse actions in a very little time (at most tens of milliseconds per problem). Metric-FF, a successor of the well known FF planner [8], uses best-first search techniques accompanied by a heuristic which is inadmissible but quite well informative. The solutions are not optimal but usually are of higher quality. In this case, fair results were achieved only in the Depots and Matching-BW domains. The state-of-the-art planner LAMA uses greedy search accompanied with Landmark and FF heuristic. The solutions are generally obtained more quickly but their quality is lower. Our method

gained promising results (shortening the solutions by more than $\sim$10%) in four domains. However, in the Gold-miner domain our method was not able to identify any redundant actions even though the solutions (of the problems in the Gold-miner domain) are far from being optimal. The Gold-miner domain is basically about finding a way through the maze to find and collect gold. There are obstacles in the maze which can be removed either by bomb or by laser. The specific issue in this domain is that if we use a bomb, the bomb is 'consumed' and we have to collect another one. On the other hand, if we use a laser, the laser remains in the hand and can be used again. Preferring bombs to lasers for removing obstacles causes a significant growth of the solution length. However, this strategy does not produce plans with redundant inverse actions which makes our method inefficient.

It is not surprising that we were able to identify only a few redundant actions by a simple analysis of the action dependency relation (i.e., actions are redundant if the goal is not dependent on them). Identifying redundant pairs of inverse actions revealed most of the redundant actions. The ordering in which pairs of inverse actions were checked for redundancy (see Section VII) showed its efficiency since we had to re-check the pairs for redundancy only in two cases ($\sim$0.5% of all the checks). Grouping nested pairs of inverse actions (see Section VII-A) was beneficial especially in the Depots and Matching-BW domains. This is because a single hoist (or robotic hand) operates over more pallets (or spots) on which objects (e.g crates) can be stacked, and we need two actions to move an object from one stack to another using the hoist (or robotic hand). As indicated in the example discussed in Section VII-A, it might easily happen that in some plan we move an object a between some stacks, then an object b between two stacks on which a has not been stacked, and then move a back. Moving a somewhere and then back is obviously redundant but we have to remove all four actions responsible for this at once.

The presented techniques for determining redundant actions in plans are focused on the most common situations but cannot reveal all redundant actions. Our aim is to provide a computationally easy method for determining redundant actions. Due to the NP-hardness of the problem of determining the maximal set of redundant actions in a given plan, we cannot guarantee to find all of the redundant actions. Plan optimization besides determining redundant actions is also about determining whether some subsequence of actions in a plan can be replaced by a shorter (or optimal) subsequence of actions. Current techniques that have been mentioned such as Neighborhood Graph search [12] and AIRS [13] are addressing this issue. However, our method is complementary to these techniques rather than a competitor. We believe that our method can be used to 'pre-optimize' plans before more sophisticated techniques (such as one of these) are applied.

X. Conclusions

In this paper we have presented techniques for determining redundant actions in plans, especially pairs or grouped nested pairs of inverse actions. This can be used for post-planning plan optimization since redundant actions can safely be removed from plans. The efficiency of the process of checking pairs of inverse actions for redundancy has also been considered and we suggested in which order these pairs should be checked. We have presented relevant theoretical foundations and provided an empirical evaluation of the proposed techniques for determining redundant actions. The empirical evaluation then showed that plans can be fairly optimized (shortened) in a very short time (tens of milliseconds).

In future we are going to investigate how we can efficiently find non-optimal subsequences of actions (not necessarily adjacent) in plans. This should deal with issues such as LAMA's non-optimal strategy in solving the Gold-miner problems (discussed in the previous section). Also we will study how to extend our approach for non-classical planing (e.g. temporal or probabilistic planning).

Acknowledgements

The research was funded by the UK EPSRC Autonomous and Intelligent Systems Programme (grant no. EP/J011991/1).

References

[1] M. Ghallab, D. Nau, and P. Traverso, *Automated planning, theory and practice*. Morgan Kaufmann Publishers, 2004.
[2] T. Bylander, "The computational complexity of propositional strips planning," *Artificial Intelligence*, vol. 69, pp. 165–204, 1994.
[3] M. Helmert, "Complexity results for standard benchmark domains in planning," *Artificial Intelligence*, vol. 143, no. 2, pp. 219–262, 2003.
[4] M. Helmert, "New complexity results for classical planning benchmarks," in *Proceedings of ICAPS 2006*, 2006, pp. 52–62.
[5] E. Fink and Q. Yang, "Formalizing plan justifications," in *In Proceedings of the Ninth Conference of the Canadian Society for Computational Studies of Intelligence*, 1992, pp. 9–14.
[6] A. Gerevini, A. Saetti, and I. Serina, "Planning in pddl2.2 domains with lpg-td," in *Proceedings of the fourth IPC*, 2004.
[7] A. Blum and M. Furst, "Fast planning through planning graph analysis," *Artificial Intelligence*, vol. 90, no. 1-2, pp. 281–300, 1997.
[8] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, 2001.
[9] S. Richter and M. Westphal, "The lama planner: guiding cost-based anytime planning with landmarks," *Journal Artificial Intelligence Research (JAIR)*, vol. 39, pp. 127–177, 2010.
[10] S. Edelkamp and P. Kissmann, "Gamer: Bridging planning and general game playing with symbolic search," in *Proceedings of the sixth IPC*, 2008.
[11] L. Chrpa, T. L. McCluskey, and H. Osborne, "Optimizing plans through analysis of action dependencies and independencies," in *Proceedings of ICAPS*, 2012, 338–342.
[12] H. Nakhost and M. Müller, "Action elimination and plan neighborhood graph search: Two algorithms for plan improvement," in *Proceedings of ICAPS*, 2010, pp. 121–128.
[13] S. J. Estrem and K. D. Krebsbach, "Airs: Anytime iterative refinement of a solution," in *Proceedings of FLAIRS*, 2012, pp. 26–31.
[14] C. H. Westerberg and J. Levine, "Optimising plans using genetic programming," in *Proceedings of ECP*, 2001, pp. 423–428.
[15] D. Chapman, "Planning for conjunctive goals," *Artificial Intelligence*, vol. 32, no. 3, pp. 333–377, 1987.
[16] L. Chrpa, "Generation of macro-operators via investigation of action dependencies in plans," *Knowledge Engineering Review*, vol. 25, no. 3, pp. 281–297, 2010.
[17] J. Hoffmann, "The metric-ff planning system: Translating "ignoring delete lists" to numeric state variables," *Journal Artificial Intelligence Research (JAIR)*, vol. 20, pp. 291–341, 2003.
[18] M. Ghallab, C. K. Isi, S. Penberthy, D. E. Smith, Y. Sun, and D. Weld, "Pddl - the planning domain definition language," Tech. Rep., 1998.