Heuristic Evaluation Based on Lifted Relaxed Planning Graphs

Bram Ridder and **Maria Fox**
King's College London
Department of Informatics

Abstract

In previous work we have shown that grounding, while used by most (if not all) modern state-of-the-art planners, is not necessary and is sometimes even undesirable. In this paper we extend this work and present a novel forward-chaining planner that does not require grounding and can solve problem instances that are too large for current planners to handle. We achieve this by exploiting equivalence relationships between objects whist constructing a lifted version of the *relaxed planning graph* (RPG) and extracting a relaxed plan. We compare our planner to FF and show that our approach consumes far less memory whist still being competitive. In addition we show that by not having to ground the domain we can solve much larger problem instances.

Introduction

In this work we present a new planning system that incorporates a novel heuristic and novel pruning techniques. The heuristic presented in this paper is *lifted*, this means that we do not need to enumerate all possible actions in order to find a solution to a planning problem. The heuristic presented in this paper is calculated by extracting a relaxed plan from a lifted version of the RPG. Whereas an RPG is constructed by instantiating operators on the object level, a lifted RPG is constructed using operators that are instantiated on the type level. We construct these types by introducing *equivalence relationships* between objects. This eliminates a limitation that is shared by many – if not all – state-of-the-art planners, *grounding*. Grounding involves the explicit enumeration of all possible ways to bind variables to objects.

Despite the benefits of grounding there is a serious drawback to this enumeration – the amount of memory required to store all the grounded actions. This drawback is not noticeable when we try to find plans for the benchmark domains of the *international planning competitions* because the sizes of these domains are relatively small compared to *real-life* problems. *Real-life* problems are not necessarily more difficult than the problems presented in the benchmark domains but the size of the domains can be much bigger (e.g. (Flórez et al. 2011)). This means that, despite their strengths, state-of-the-art planners cannot even start solving these problems. In addition, grounding large domains can take a considerable amount of time.

By eliminating this limitation we are able to solve larger problem instances that cannot be solved by planners that rely on grounding due to memory constraints. In addition we also present novel pruning techniques that further enhance the scalability of our planning system. We compare our planning system with FF (Hoffmann 2001).

The rest of the paper is structured as follows:

In section 2 we present the motivation for our work by analysing the FF heuristic and the construction of the RPG. In section 3 we present our lifted variant of the RPG, the lifted RPG, and describe how we extract a heuristic from this structure. In section 4 we present the empirical evaluation of our planning system compared to FF and demonstrate that is uses significantly less memory whilst still being competitive. In addition we show that our planning system scales much better than FF and can solve larger problem instances. In section 5 we present an overview of the literature that is related to our work. Finally, we present our conclusions in section 6.

Motivation

In this work we focus on *classical planning* problems.

Definition 1 — Typed planning task
A typed planning task is a tuple $\Pi = \langle T, O, P, A, s_0, s_g \rangle$, where O is a set of objects, each object $o \in O$ is associated with a type $t \in T$, written $Type(o)$. An atom is a tuple $\langle p, V \rangle$, where $p \in P$ is a predicate and V is a sequence of variables. A variable v is a pair $\langle t, D_v \rangle$, where $t \in T$ and $D_v \subseteq O$ is the domain. We refer to the ith variable with the notation V_i. A is a set of operators, where an operator $a \in A$ is a tuple $\langle name, parameters, precs, effects \rangle$, $parameters$ is a sequence of variables. $precs$ and $effects$ are sets of atoms, whose variables are elements of $parameters$. We use the notation $effects^+$ for the subset of atoms in $effects$ that are positive.

A solution to a typed planning task is a sequence of grounded actions that is applicable in the initial state, s_0 and the resulting state satisfies the goal state s_g.

The only domain independent planning system – to our knowledge – that tries to solve large problem instances is RealPLAN (Srivastava 2000). This planner was written to

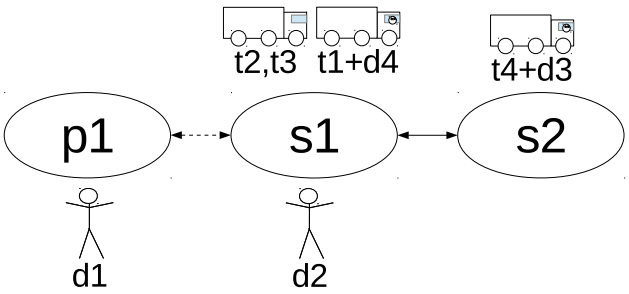

Figure 1: A simple Driverlog problem. The dashed lines can be traversed by drivers and the solid lines can be traversed by trucks.

solve the paradox that problem instances are harder to solve when more resources are added to the planning domain. E.g. they found that solving a planning problem to deliver a set of packages with one truck was easier than the same planning problem with a fleet of ten trucks. They decoupled the reasoning about resources from the planning task and found that they could solve larger problem instances when far more resources are supplied than are needed. Our approach, on the other hand, is more general and can solve large problem instances even if the resources are not over-supplied.

In this work we generalise the construction of the RPG such that it does not require every action to be grounded. The process of constructing an RPG involves a lot of redundancy.

Example 1 *Consider the trucks* t2 *and* t3 *in the Driverlog problem depicted in Figure 1. These trucks start at the same location, are empty, and are not being driven by a driver. It is clear that the same set of actions will be applied to all these trucks and that any fact that is reachable for one truck (e.g.* (at t2 s3)*) will also be reachable for any of the other trucks (e.g.* (at t3 s3)*). These trucks are* equivalent*, so we can substitute these instances with a new object* T. *This reduces the size of the RPG considerably because instead of dealing with these two trucks separately we now only need to consider one.*

Just as the trucks in Example 1 can be made equivalent, it is clear that although the instances of type *driver – d1*, *d2*, *d3*, and *d4* – are not at the same location in the initial state they can become equivalent too. This is because all these drivers can *reach* the same locations and board the same trucks, any fact that is reachable for one driver is also reachable for the other drivers. In the next section we will formalise how we find the *equivalence* relationships between objects and how we subsequently exploit these relationships during the construction of the lifted RPG and the extraction of the heuristic.

In this work we find and exploit these *equivalence relationships* between objects and show the reduction in the number of actions we need to consider in order to construct the lifted version of the RPG. The concept of *equivalence* is tightly related to symmetry breaking, we refer to section 5 for an overview of the literature and the relation to this work.

The Lifted Relaxed Planning Graph Heuristic

As we have demonstrated in Example 1, we can significantly reduce the size and number of actions in an RPG if we combine objects into equivalence classes.

Definition 2 — Object Equivalent Classes
Given a typed planning problem $\langle T, O, P, A, s_0, s_g \rangle$ and a state s that contains all facts that are reachable from s_0 using the set of relaxed actions $\{a \in A \mid a_{effects} = a_{effects^+}\}$, two objects $o \in O$ and $o' \in O$ are part of the same equivalent class iff $Type(o) = Type(o')$ and $\kappa_{o,o'}(s) = s$, where

$$k_{o,o'} : state \rightarrow state,$$

such that $k_{o,o'}$ transposes all occurrences of o and o'.

If we apply Definition 2 to Example 1 then we can construct two *object equivalent classes*, one for all the drivers and one for all the trucks.

Definition 3 — Equivalent Objects
Given a typed planning problem $\langle T, O, P, A, s_0, s_g \rangle$ and a state s, two objects $o \in O$ and $o' \in O$ are equivalent iff they are part of the same equivalent class and $\kappa_{o,o'}(s') \subseteq s_0$, where s' is the set of all facts in s that contain o or o'.

In Example 1 the trucks *t2* and *t3* are equivalent because they are at the same location and empty; Both can reach the same locations and can be driven by the same set of drivers by applying a similar sequence of actions. For example, consider any sequence of actions from the initial state – for example: { *(board d1 t2 s1)*, *(driver d1 t2 s1 s2)* } – if we alter the sequence of actions by replacing all instances of *t2* with *t3* then the sequence of actions is still valid and the resulting state is equivalent except that driver *d1* is driving the truck *t3* and *t3* is at location *s2*.

The drivers *d1* and *d2*, on the other hand, are not equivalent because they are both at different locations. However, if we execute the sequence of actions (*(walk d1 s1 p1)*, *(walk d2 p1 s1)*) then both drivers become equivalent as well.

Object equivalence classes can be constructed by constructing an RPG till the level-off point and comparing the facts in the last fact layer using Definition 3. However, in order to construct the RPG we need to ground the domain.

In order to infer equivalence classes without having to ground the entire domain, we use TIM (Fox and Long 1998). TIM performs domain analysis on a typed planning task without having to ground the domain and infers a new type structure. Objects that are part of the same set of preconditions and effects and are part of the same variable domains are assigned the same *type*. In Example 1, the objects *t1*, *t2*, *t3* and *t4* are assigned the same *type* because given the operator *drive* they appear in the first variable domain of the precondition and effect *(at { t1, t2, t3, t4} { s1, s2, p1 })* and they appear in the same indexes of all the other preconditions and effects of the other operators.

For each inferred type and the set of objects that are part of that type, TIM performs a reachability analysis from the initial state to determine the set of indices of predicates the type can be part of. An index i of a predicate, with the name n, is called a *property* and is denoted as n_i. For example, consider the inferred type t that contains the set of objects

{ *t1*, *t2*, *t3*, *t4* }, from the initial state we can extract the *properties* at_1, $empty_1$ and $driving_2$. Next TIM performs a reachability analysis by checking which operators can add or remove properties that are part of t. In this case the operators *board*, *disembark*, and *drive* affect the properties of t.

Given the sets of objects, properties, and operators that affect those properties for an inferred type t, TIM creates *transitions rules* that map the *exchange* of one set of properties for another given an operator. We use the following notation: $\langle$ set of properties that are deleted by $o\rangle \overset{o_{name}}{\Rightarrow} \langle$ set of properties that are added by $o\rangle$, where o is an operator. The transitions rules that are constructed for our example are: $\langle empty_1\rangle \overset{board}{\Rightarrow} \langle driving_2\rangle$, $\langle driving_2\rangle \overset{disembark}{\Rightarrow} \langle empty_1\rangle$, and $\langle at_1\rangle \overset{drive}{\Rightarrow} \langle at_1\rangle$.

The sets of deleted properties and added properties from the previous example were never empty, but this is not always the case. Consider, for example, the following transition rules from the type that has been inferred for the set of objects { *s1*, *s2*, *p1* }: $\langle at_2\rangle \overset{drive}{\Rightarrow} \langle\rangle$ and $\langle\rangle \overset{drive}{\Rightarrow} \langle at_2\rangle$. The former loses properties while the latter gains properties.

Using the analysis performed by TIM we construct the object equivalence classes as follows. Given an initial state s_0, an inferred type t, two objects $o \in t$ and $o' \in t$ are part of the same equivalence class if the set of properties that are true for o in s_0 can be *exchanged* for the set of properties that are true for o' in s_0 using the transition rules of t and visa versa.

Example 2 *We refer back to the example depicted in Figure 1. Using TIM we can prove that the drivers* d2 *and* d4 *are part of the same equivalence class. Both drivers are part of the inferred type that contain the objects* { d1, d2, d3, d4 }. *The set of properties that are true for* d2 *in the initial state is* $\{at_1\}$ *and the set of properties that are true for* d4 *in the initial state is* $\{driving_1\}$. *We can use the transition rule* $\langle at_1\rangle \overset{board}{\Rightarrow} \langle driving_1\rangle$ *to exchange the set of properties of* d2 *for the set of properties* $\{driving_1\}$. *Likewise we can reach the set of properties that are true for* d2 *in the initial state from the set of properties that are true for* d4 *in the initial state by using the transition rule* $\langle driving_1\rangle \overset{disembark}{\Rightarrow} \langle at_1\rangle$. *Therefore we can conclude that* d2 *and* d4 *are part of the same equivalence class.*

Following TIM's analysis we would conclude that all the drivers, trucks, packages, and locations are part of the same equivalent class for any possible Driverlog problem. However, this analysis only holds if the road network is connected, if the road network is disconnected then this analysis does not hold up. To demonstrate this we revisit example 1 but we make a small change to the road layout; imagine that there is no connection between *s1* and *s2*. In that case the two sets of trucks, { *t1*, *t2*, *t3* } and { *t4* } cannot become equivalent and are not part of the same equivalent class.

In order to differentiate objects that are part of the same type, as detected by TIM, but are not part of the same equivalent class we subdivide the detected types into subtypes. Using the TIM analysis we split up any type that contains a transition rule that gains or loses properties. In the above example we split up the inferred type that contains the objects { *s1*, *s2*, *p1* }, such that every object becomes part of a separate type. Now we can differentiate between objects that are part of different road networks. We further refine the types detected by TIM by comparing facts in the initial state that cannot be affected by any action, these are *static* facts. Let s be the set of static facts that are part of the initial state and contain the object o or o', o and o' cannot be part of the same type if $s \neq \kappa_{o,o'}(s)$.

Image that the Driverlog problem depicted in Figure 1 has the static fact *(is-small t1)* that allows the truck *t1* to access the location *p1*. In that case *t1* is no longer part of the same equivalence class as the other trucks, because the other trucks cannot access *p1*.

	Zeno	Satellite	Storage
Lifted Transitions	5500	4562	170
Grounded Transitions	959530	43290	348660
	Driverlog	Gripper	Depots
Lifted Transitions	528	8	112
Grounded Transition	218300	6204	55936
	Blocksworld	Rovers	
Lifted Transitions	4	7364	
Grounded Transitions	612	423064	

Table 1: Number of transitions per planning domain.

Construction of the Lifted Relaxed Planning Graph

Using the above method we will now describe how we construct the lifted RPG. In order to construct the lifted RPG we partially ground the actions using the types derived using TIM as we have described in the previous section. The number of actions we need to consider is significantly fewer than if we were to ground the entire problem. The number of actions we consider using our approach and the total number of grounded actions are depicted in Table 1 for the largest problems of domains from past *international planning competitions*.

The method we use to construct a lifted RPG shares some similarity with the way FF constructs the RPG, but there are some significant differences. We construct the lifted RPG by applying actions to the current fact layer in order to construct the next fact layer. However, whereas the action layers in an RPG contain grounded actions we use partially grounded actions and whereas the fact layers in RPGs contain grounded atoms we use lifted atoms. This makes the construction of the lifted RPG harder than the grounded variation. Given a set of facts and a partially grounded action, we need to find all sets of facts that satisfy all the preconditions. If there are n facts that satisfy each precondition then we need to check n^m combinations, where m is the number of preconditions. This problem does not exist for the grounded case, because in the grounded case $n = 1$.

In order to reduce the number of combinations we need to check, we split the preconditions into a number of sets. The sets are constructed in such a way such that if we find a satisfying set of facts for each set of preconditions and take the union of these sets of facts then this union satisfies all the

preconditions of the action. If we split the preconditions in l sets such that each set contains m preconditions and there are n facts that satisfy each precondition then the number of combinations we need to check is ln^m, so the smaller the sets the less overhead we incur whilst checking for sets of facts that satisfy the preconditions of an action.

We must put constraints on how the preconditions are split in order to guarantee that any union of the sets of facts that satisfy the preconditions satisfies all the preconditions. This can only be guaranteed if every pair of preconditions that shares a variable is either in the same set or if the shared variable is grounded. Returning to Example 1, consider the partially grounded action *(board { d1, d2, d3, d4 } { t1, t2, t3, t4 } s1)* the sets of preconditions are: { *(at { t1, t2, t3, t4 } s1)* } and { *(at { d1, d2, d3, d4 } s1)* }. Note that because the location *s1* is grounded we are guaranteed that the union of any facts that satisfy the sets of preconditions satisfies all the preconditions of the action.

Given a typed planning problem and the set of partially grounded actions we constructed the lifted RPG as follows:

1. Initialise the first fact layer with s_0.
2. Given the current fact layer, create a mapping from every object to the set of equivalent objects using Definition 3 and update the variable domains of all the facts using this mapping.
3. Remove all facts that are identical.
4. Create a new fact layer by copying all the facts from the previous layer by applying a NOOP.
5. Apply any partially grounded actions as described above and add the resulting facts to the new fact layer.
6. If we are not at the level-off point then repeat (2).

In the worst case scenario the lifted RPG will be identical to the RPG, this is the case when no objects can become equivalent. However, in most cases we find that the lifted RPG contains far fewer action layers, fact layers, and the number of actions in the fact layers is also reduced. The time it takes to construct the lifted RPG depends on the number of objects that can be proven to be equivalent, but for most domains we have tried this method on we are able to construct the lifted RPG quicker than the RPG (Ridder and Fox 2011). In the same work we also prove that the facts in the final fact layer when we constructed the lifted RPG till the level-off point is identical to the set of facts in the last fact layer when we construct the RPG till the level-off point. So if a goal is not present in the final fact layer then we know that we have reached a dead end.

Example 3 *The lifted RPG that is constructed for the problem depicted in Figure 1 is depicted in Figure 2. As we can see, this lifted RPG has two action layers, three fact layers, and contains 14 actions. Compare this to the RPG for the same problem that contains four action layers, five fact layers and 104 actions. In larger planning problems this difference in the number of action layers, fact layer, and number of actions is even larger.*

Calculating the heuristic

Given a lifted RPG that is constructed to the level-off point we will now describe how we extract a relaxed plan. The method we use is closely related to the method FF uses. We initialise an open list G with the facts in the last fact layer that correspond with the facts in s_g. While G is not empty, we select a fact $f \in G$ such that there is no other fact in G that is part of a fact layer with an higher index. Subsequently we select an action that achieves f, if multiple actions are available we choose the action with the lowest *action cost*, and add the preconditions to G. The length of the relaxed plan is the heuristic estimate.

Because the size of the lifted RPG is at most as large as the RPG we can extract the heuristic estimate quicker. However, the heuristic extracted is not as informative as the FF heuristic.

Example 4 *Consider the lifted RPG depicted in Figure 1. If the goal to be achieved is* (at d1 s2)*, then the relaxed plan extracted from the lifted RPG is:* ((disembark { d1,d2,d3,d4 } { t1,t2,t3,t4 } s2)) *so the heuristic estimate is 1.*

The reason why the heuristic estimate is poor compared to the FF heuristic (which would return the optimal heuristic estimate of 4) is because we introduce shortcuts in the lifted RPG due to the object equivalences. In the example above *d1* becomes equivalent with the driver *d2* once it reaches location *s1*, because *d2* can reach the location *p1*. At the same time the drivers *d2* and *d4* become equivalent because *p4* can reach the location *s1* and *d2* can board the truck *t1*, this means that all these three drivers become equivalent in the second fact layer. In the third fact layer the drivers *d2* and *d3* become equivalent, because *d2* can board *t4* and *d3* can reach the location *s1*, this now means that all drivers are equivalent in the third fact layer. This is why the action that is selected to achieve the goal *(at d1 s2)* is a NOOP with the precondition *(at d3 s2)* and the action that is achieved to achieve this precondition is *(disembark { d1,d2,d3,d4 } { t1,t2,t3,t4 } s2)*.

This is clearly undesirable, in order to partially remedy this situation we augment the heuristic estimate with the cost of making *substitutions*. In the previous example we effectively substituted *(at d1 s2)* with *(at d3 s2)* which resulted in heuristic estimate that is too low. In the next section we present two methods to calculate the cost of making a substitution. Neither method will produce the same heuristic estimate as FF because of the shortcuts that have been introduced in the construction of the lifted RPG.

Substitutions

In order to account for the discrepancies between the identity of the objects in the relaxed plan we augment the heuristic whenever a substitution needs to be made. In order to detect whenever a substitution needs to be made we change the algorithm we use to extract a relaxed plan; Instead of adding the preconditions of an achiever to G we *instantiate* the action using the preconditions and the fact we want to achieve and we add the instantiated preconditions to G. A substitution needs to be made whenever a variable domain

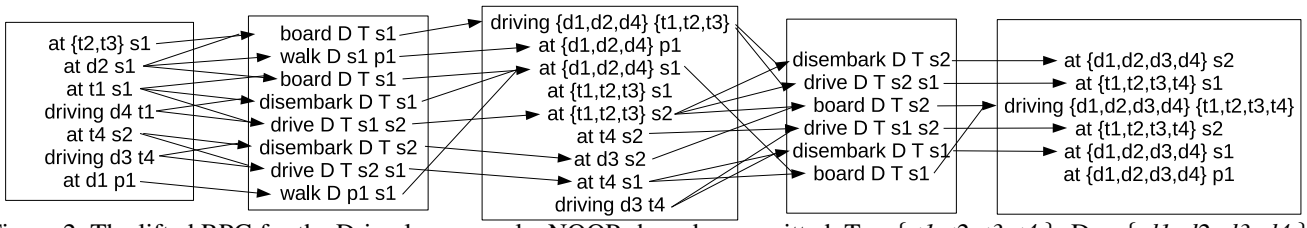

Figure 2: The lifted RPG for the Driverlog example, NOOPs have been omitted. T = { *t1*, *t2*, *t3*, *t4* }; D = { *d1*, *d2*, *d3*, *d4* }.

of the achiever becomes empty, in that case we use the variable domain of the fact we want to be achieved to instantiate the action.

In Example 4 the achiever *(disembark { d1,d2,d3,d4 } { t1,t2,t3,t4 } s2)* is selected as the achiever of the goal *(at d1 s2)*. Instead of adding the preconditions of the achiever to the open list G we *instantiate* the action using the goal *(at d1 s2)* and the preconditions in the previous fact layer. This yields the action: *(disembark {} t4 s2)* this action contains an empty variable domain so it necessary to make a substitution. Using variable domains of the goal to fill in the empty variable domain gives us the action *(disembark d1 t4 s2)* which yields the preconditions: *(at t4 s2)* and *(driving d1 t4)* that are added to G.

In order to account for any substitutions that need to be made, we present the following two methods:

- **ObjectSub**: Given a fact $f = \langle p, V \rangle$ that is achieved by the effect $f' = \langle p', V' \rangle$ of an *instantiated* action, for every $i \in \{1, \ldots, \mid V \mid\}$ where $D \cap D' = \emptyset \mid D \in V_i, D' \in V_i'$ we find the first fact layer where $o \in D$ and $o' \in D'$ become equivalent and add the layer number to the total of the heuristic.
- **GoalSub**: Given a fact $f = \langle p, V \rangle$ that is achieved by the effect $f' = \langle p', V' \rangle$ of an *instantiated* action a, we update the variable domains of every action parameter $\langle t', D' \rangle \in V_i'$ as $\langle t', D' \rangle = \langle t', D' \cap D \rangle \mid D \in V_i, i \in \{1, \ldots, \mid V \mid\}$ if $D' \cap D \neq \emptyset$, otherwise $D' = D$. The preconditions $p \in a_{preces}$ with the updated variable domains are added to G.

The pseudocode of extracting a relaxed plan using substitutions is shown in Algorithm 1.

We will now show how these two approaches improve the heuristic estimate we calculated in Example 4.

The relaxed plan contains a single action, *(disembark { d1,d2,d3,d4 } { t1,t2,t3,t4 } s2)*. When we instantiate this action as we have done above we end up with an empty variable domain. This is because the intersection of the variable domain of the precondition – { *d3* } – with the corresponding variable domain of the effect – { *d1* } – is empty. Using the *ObjectSub* substitution method we need to make a substitution between the objects *d3* and *d1*. From the lifted RPG depicted in Figure 2 we can see that *d3* and *d1* become equivalent in the third fact layer so we add 2 to the total of the heuristic so the heuristic evaluation becomes 3 (note that the first fact layer counts as 0).

If the *GoalSub* substitution method is used, then the algorithm used to extract relaxed plans uses a different method to select an achiever for a goal. Instead of selecting the achiever with the lowest action cost we prefer achievers that do not necessitate any substitutions and only select NOOPs if there is no other option. This changes the extracted relaxed plan, because instead of choosing the NOOP to achieve the fact *(at d1 s2)* we select the achiever *(disembark { d1, d2, d3, d4 } { t1, t2, t3, t4 } s2)*. When we instantiate this action we get *(disembark { d1 } { t1, t2, t3 } s2)* which yields the preconditions *(driving d1 { t1, t2, t3 })* and *(at { t1, t2, t3 } s2)* that are added to G. The next goal that we process is *(at { t1, t2, t3 } s2)*, this goal is achieved by the achiever *(drive d4 t1 s1 s2)* that does not necessitates any substitutions and all the preconditions are part of the first fact layer. The next goal that we process is *(driving d1 { t1, t2, t3 })* that is achieved by the achiever *(board { d1, d2, d3, d4 } { t1, t2, t3, t4 } s1)*. Instantiating this achiever yields *(board { } { t2, t3 } s1)* which yields the preconditions *(at { t2, t3 } s1)* – which is part of the initial state – and the newly created precondition *(at d1 s1)* that is added to G. This last fact is achieved by the achiever *(walk { d1, d2, d3, d4 } p1 s1)* and this completes the construction of the relaxed plan. The length of the final relaxed plan is 4, which is identical to the heuristic estimate extracted from the RPG.

While the second substitution method produces better heuristic estimates in most cases, we cannot use it if we need to make a substitution when the achiever is a NOOP. Consider for example the case where we have a NOOP with the

Algorithm 1: Extract a Relaxed Plan

Data: The set of fact layers fl, the set of action layers al, the initial state s_0, the goal state s_g, and the substitution method M (if any).
Result: The relaxed plan T.
begin
  $G \longleftarrow \{\langle f, \mid fl \mid \rangle \mid f \in s_g \}$;
  while $G \neq \emptyset$ **do**
    $\langle f, i \rangle \in G \mid \neg\exists_{\langle f', j \rangle \in G} j > i$;
    $G \longleftarrow G \setminus \langle f, i \rangle$;
    $a \in al_{i-1} \mid a$ is the cheapest achiever for f;
    if $M \in \{$*ObjectSub*, *GoalSub*$\}$ **then**
      Augment the heuristic estimate and add facts to G according to M;
    if $a \notin T$ **then**
      $T \longleftarrow T \cup a$;
    if $i - 1 \neq 0$ **then**
      $G \longleftarrow G \cup \langle p, i - 1 \rangle \mid p \in a_{precs}$;

precondition *(at d1 s1)* and *(at d2 s1)* is the fact that we want to achieve, if we use the second method then the new goal that is added is identical to the original goal, *(at d2 s1)*, and we end up in an infinite loop. Whenever we need to add a goal that is identical to the fact we want to achieve, we fall back to the first substitution method to calculate the cost of the substitution.

Implementation of the Planning System

We will now discuss the implementation of the planning system that utilises the lifted RPG heuristic. We will first discuss how helpful actions are extracted from the lifted RPG, we will then introduce a novel pruning technique and finally we present a number of different configurations of this planner that we have tested against FF.

Helpful actions

There is no difference in the way we extract helpful actions from a lifted RPG compared to how FF extracts helpful actions from an RPG. Given a relaxed RPG and an extracted relaxed plan, we call an action helpful if it is applicable in the initial state and achieves a fact that is part of the second fact layer of the lifted RPG and is a precondition for any action in the relaxed plan. Unfortunately we have found that while this pruning technique does reduce the search space it might misdirect the search effort. For example, the set of helpful actions for the relaxed plan { *board* { *d1,d2,d3* } { *t1,t2,t3,t4* } *s2*, *(drive* { *d1,d2,d3* } { *t1,t2,t3,t4* } *s2 s1)*, *(disembark* { *d1,d2,d3* } { *t1,t2,t3,t4* } *s1)*, *(walk* { *d1,d2,d3* } *s1 p1)*, *(walk* { *d1,d2,d3* } *p1 p2)* }, that has been extracted to achieve the fact *(at d3 p2)* using the second substitution approach, is { *(walk d1 s1 p1)*, *(walk d2 p1 p2)*, *(board d3 t4 s2)* }. Of this list only the last action is actually helpful, the other helpful actions will not reduce the heuristic value nor bring us closer to a solution to the planning problem. We expect that using helpful actions might have adverse effect for this reason.

Preserve goals

The lifted RPG heuristic relaxes the plan by ignoring the delete effects. This means that the relaxed plan will sometimes destroy goals that have been achieved or utilise resources that are no longer available. In order to calculate better heuristic estimates we restrict the way in which lifted RPGs are constructed. Given the set of goals s_g we do not allow any actions that delete any of these goals. We found that using this technique has a good synergy with helpful actions and partially alleviates the problem that helpful actions can misdirect the planner.

We found that restricting the construction of the lifted RPG such that no goals can be deleted generates better heuristic estimates. However, it is possible that imposing these restrictions on the construction of the lifted RPG might make it impossible to find a relaxed plan. If that is the case then we reconstruct the lifted RPG without these constraints in order to find a relaxed plan.

Configuration Name	prune unhelpful actions	preserve goals	substitution method
NoSub	No	No	None
ObjectSub	No	No	ObjectSub
ObjectSub (h)	Yes	No	ObjectSub
ObjectSub (h+p)	Yes	Yes	ObjectSub
GoalSub	No	No	GoalSub
GoalSub (h)	Yes	No	GoalSub
GoalSub (h+p)	Yes	Yes	GoalSub

Table 2: The different configurations of our planning system.

Results

In order to test the strengths and weaknesses of this planning system we introduce seven configurations that we have tested against FF. The configurations are listed in Table 2 and we will describe each configuration below.

NoSub

The first configuration does not use any substitutions nor any pruning techniques. This configuration serves as the baseline to test the benefits of the two substitution methods in combination with helpful actions and *preserve goals*. We expect that this configuration is able to explore a larger part of the search space because it is able to calculate heuristic estimates faster, but due to the poor quality of these heuristics (see Example 4) we do not expect this configuration to be as good as the other configurations.

ObjectSub

The second configuration uses the *ObjectSub* substitution method. The heuristic estimates it calculates, as we have seen in Example 4, is a massive improvement over the baseline and the overhead of resolving substitutions is very small so we expect that this configuration will solve many more problems. We also ran experiments where we used helpful actions to prune the search space. However, as we have explained in the previous section, helpful actions do not always help the planner and can misdirect it. We have included this configuration to test this hypothesis. The final configuration uses helpful actions to prune the search space and it constrains the construction of the lifted RPG by trying to preserve goals. We hope that the helpful actions extracted from the constrained lifted RPG will better guide the planner to a goal state.

GoalSub

The third and last configuration uses the *GoalSub* substitution method. This configuration calculates the best heuristic estimates. However, the overhead to calculate these heuristic estimates is the highest of all configurations because it is possible that many new goals are added to the lifted RPG before we find a heuristic estimate. Like the *ObjectSub* configuration we test the effectiveness of pruning the search space with helpful actions and test if constraining the lifted RPG by preserving goals helps the quality of the heuristic estimate and the helpful actions. An additional benefit of constraining the lifted RPG by preserving goals is that it limits the number of substitutions we need to make which will improve the speed of calculating heuristic estimates.

Experiments

In this section we will compare the seven configuration of our planner depicted in Table 2 against FF. We compare the number of problems solved and the memory used to solve these problem instances. We will also present results that demonstrates that our planner can solve significantly larger problem instances compared to FF.

We ran all our experiments on an Intel Core i7-2600 running at 2.4GHz and allowed 2GB of RAM and 30 minutes of computation time. We have taken seven problem domains from various planning competitions, and we now discuss the results obtained in these domains.

The number of planning problems solved for each configuration are listed in Table 3. We can see that the baseline – *NoSub* configuration of our planner – solves the fewest problems. We have highlighted the problem with this approach in Example 4 so this does not come as a surprise. We see that the number of problems solved almost doubles when we account for substitutions. Interestingly we solve fewer problems with the *GoalSub* configuration, that makes substitutions by adding new goals to the lifted RPG, than the *ObjectSub* configuration, even though we find that the former provides better heuristic estimates on average. This is because calculating the heuristic estimate using *GoalSub* takes longer compared to *ObjectSub*, thus even though we get better heuristic estimates on average it does not make up for the longer time it takes to compute it.

However, when we prune the search space using helpful actions – which means we need to evaluative fewer states – and constrain the construction of the lifted RPG – which speeds up the calculating of the heuristic – we see that the better heuristic estimates pays off.

	NoSub	ObjectSub			GoalSub			FF
Domain			h	h+p		h	h+p	
Driverlog	9	14	13	15	14	15	**18**	15
Zeno	6	13	13	15	12	14	14	**20**
Blocksworld	3	**20**	**20**	**20**	**20**	**20**	**20**	14
Storage	17	17	17	17	**18**	17	16	**18**
Depots	4	8	8	13	6	9	12	**20**
Satellite	7	18	17	17	7	18	18	**20**
Rovers	11	18	18	18	13	18	19	**20**
Total	57	108	106	115	90	110	117	**127**

Table 3: Number of problems solved. h means helpful actions enabled. p means that goals are preserved.

Memory results To verify that our approach uses less memory than FF we have recorded the memory usage for each problem that has been solved for each configuration. The memory results for *NoSub* are depicted in Figure 3. We can see that we use significantly less memory compared to FF to solve the planning problems. For certain domains we seem to scale worse than FF, most notably Rovers and Storage. The reason is that in these domains very few objects can become equivalent.

The results for the *ObjectSub* configuration are depicted in Figure 4. We see that a few of the Blocksworld planning problems seem to scale badly. On these problems we

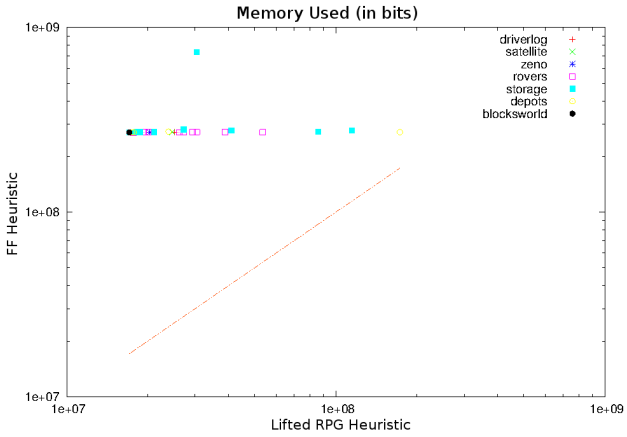

Figure 3: Results for the *NoSub* configuration.

reached plateaus and had to store a great number of states before we could escape the plateaus or discover that we could not find a solution by pruning actions which are not helpful.

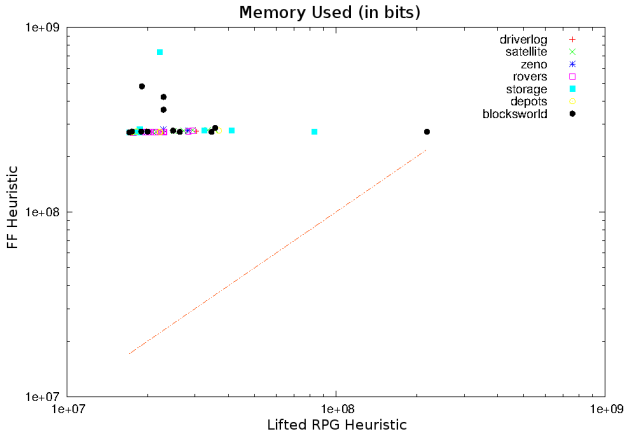

Figure 4: Results for the *ObjectSub (h+p)* configuration.

The results for the *GoalSub* configuration are depicted in Figure 5. We see that we do not have many outliers in the Blocksworld domain. This is because the heuristic estimate is more accurate so we do not get stuck on as many plateaus as with the *ObjectSub* configuration.

Larger domains To test our hypothesis that our planner can solve larger problem instances we generated new problem instances of the IPC domains. These problem instances have vastly more *resources* (e.g. trucks, satellites, rovers, etc), whilst keeping number of other objects the same as the benchmarks domains of the IPC. The number of goals to achieve has been reduced to one. In previous work we have already demonstrated that our planner can work on bigger problem instances (Ridder and Fox 2011). In this section we will show problem instances we can solve but FF, due to memory constraints, cannot. The results are depicted in Figure 6. With the exception of three satellite instances we solve more problem instances and consume less memory than FF that can only solve some of the smallest problem instances before running out of memory.

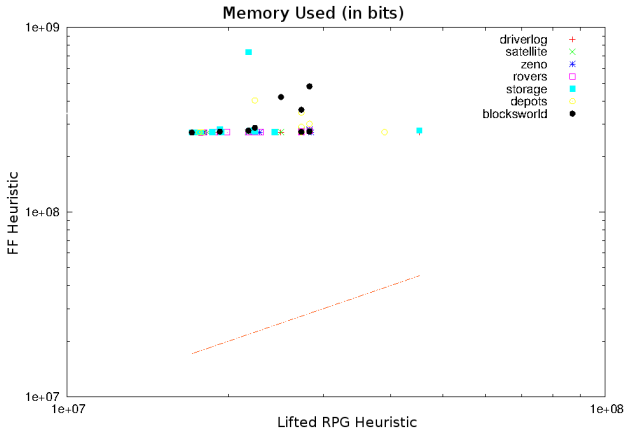

Figure 5: Results for the *GoalSub (h+p)* configuration.

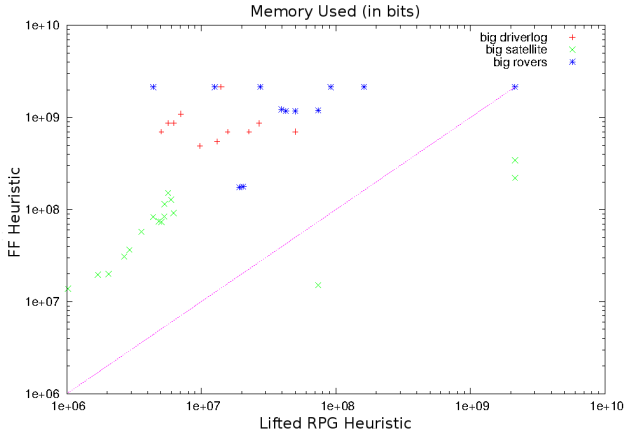

Figure 6: Results for the GoalSub (h+p) configuration on larger domains.

Related literature

In this section we will give an overview of work that is closely related to the work presented in this paper.

Lifted heuristics

Most of the heuristics that are used in state-of-the-art planners (Haslum and Geffner 2000; Bonet and Geffner 2001; Helmert 2006) require the domain to be grounded. Some heuristics, like merge & shrink (Helmert et al. 2007) and pattern databases (Edelkamp 2002), bound the memory allowed. However, the construction of the data structures for these heuristics still requires grounding. Least-commitment planners, like UCPOP (Penberthy and Weld 1992), do use a lifted heuristic. For example, UCPOP uses the number of *flaws* in a plan as the heuristic estimate. A better heuristic estimated is counting the number of *open conditions* (Gerevini and Schubert 1996). However, these heuristic estimates proved to be very poor. VHPOP (Younes and Simmons 2003) adapted the additive heuristic (Bonet, Loerincs, and Geffner 1997) that can be calculated by *binding* the variable domains of the actions as needed. However, in order to calculate the heuristic most actions will be grounded so it is not clear how this method provides any benefit. Indeed the configuration used in IPC 3 (Long and Fox 2003) grounded the entire domain.

Symmetry breaking

The definition of object equivalence in this paper is closely related to symmetry breaking. In this section we present previous methods that have been developed to detect and exploit symmetry relationships.

Methods to detect symmetry have initially been developed in the context of model checking (Emerson, Sistla, and Weyl 1994) and later been extended to break symmetry in constraint satisfaction problems (Puget 1993) mostly to reduce the search space to a more manageable size. These methods have been adopted and integrated in planning systems to make planning systems more scalable in large domains that exhibit a lot of symmetry. One of the first papers to explore the use of symmetry in planning systems (Fox and Long 1999; 2002) searches for groups of symmetrical objects, defined in the paper as objects that are indistinguishable from one another in terms of their initial and final configurations. These symmetric groups are used in the planner STAN (Fox and Long 2001), based on the GraphPlan (Blum and Furst 1995) architecture for pruning search trees.

Subsequent work focused on breaking symmetry in the context of forward-chaining planners. This work focused on exploiting symmetry groups by : (1) proving that any two states s and s' are *symmetrical* (Pochter, Zohar, and Rosenschein 2011); and (2) given a state and two actions, if both actions are *symmetrical* then we only have to consider one of the actions and we can ignore the other (Rintanen 2003).

The method most relevant to our work is the latter. If two objects, o and o', are equivalent in a state s and we have a sequence of actions that achieves a fact f then we can reach the fact f' – that is the same as f except all occurrences of o and o' are transposed – by executing the same sequence of actions with all occurrences of o and o' transposed.

Conclusions

In this paper we have demonstrated a new planning system that utilises a lifted heuristic and compared it to FF. We have shown how we derive object equivalence classes of object which significantly reduces the amount of grounding we have to do. By exploiting object equivalence relationships we can construct a lifted RPG and extract a heuristic estimate faster compared to FF using an RPG. We have identified a weakness with the relaxed plan that is extracted from a lifted RPG and provided two methods to augment the heuristic estimate by making substitutions. We presented empirical results that shows the efficiency of both methods.

We have shown that we are able to solve larger problem instances compared to FF because we use significant less memory. In addition we have shown that we do not suffer from the weaknesses of previous lifted heuristics that have proven to be very uninformative. In addition we have shown that our novel pruning technique allows our planning system to solve significantly more problem instances.

References

Blum, A., and Furst, M. 1995. Fast planning through planning graph analysis. *Artificial Intelligence* 90:1636–1642.

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

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A robust and fast action selection mechanism for planning. In *AAAI*, 714–719. MIT Press.

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In *AIPS*, 274–283.

Emerson, E.; Sistla, A.; and Weyl, H. 1994. Symmetry and model checking. In *CAV*, volume 5, 463–478.

Flórez, J. E.; de Reyna, Á. T. A.; García, J.; López, C. L.; Olaya, A. G.; and Borrajo, D. 2011. Planning multi-modal transportation problems. In *ICAPS*. AAAI Press.

Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research* 9:367–421.

Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning problems. In *IJCAI*, 956–961.

Fox, M., and Long, D. 2001. Hybrid STAN: Identifying and managing combinatorial optimisation sub-problems in planning. In *IJCAI*, 445–452. Morgan Kaufmann.

Fox, M., and Long, D. 2002. Extending the exploitation of symmetries in planning. In *AIPS*, 83–91.

Gerevini, A., and Schubert, L. 1996. Accelerating partial-order planners: Some techniques for effective search control and pruning. *Journal of Artificial Intelligence Research* 5:95–137.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *AIPS*, 140–149. AAAI Press.

Helmert, M.; Freiburg, A.-L.-U.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *ICAPS*, 176–183.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Hoffmann, J. 2001. FF: The fast-forward planning system. *AI magazine* 22:57–62.

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research* 20:1–59.

Penberthy, J. S., and Weld, D. S. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *KR*, 103–114. Morgan Kaufmann.

Pochter, N.; Zohar, A.; and Rosenschein, J. S. 2011. Exploiting problem symmetries in state-based planners. In *AAAI*.

Puget, J. 1993. On the satisfiability of symmetrical constrained satisfaction problems. In *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems*, ISMIS '93, 350–361. London, UK: Springer-Verlag.

Ridder, B., and Fox, M. 2011. Performing a lifted reachability analysis as a first step towards lifted partial ordered planning. In *PlanSIG*.

Rintanen, J. 2003. Symmetry reduction for SAT representations of transition systems. In *ICAPS*, 32–41. AAAI Press.

Srivastava, B. 2000. Realplan: Decoupling causal and resource reasoning in planning. In *AAAI/IAAI*, 812–818. AAAI/MIT Press.

Younes, H. L. S., and Simmons, R. G. 2003. VHPOP: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research* 20:405–430.