Dealing with Superfluous Numbers of Agents*

Dionysis Kalofonos
University Carlos III of Madrid, Spain
dkalofon@inf.uc3m.es

Timothy J. Norman
University of Aberdeen, UK
t.j.norman@abdn.ac.uk

Abstract

During problem solving the set of actions considered by the planner results from the instantiation of operators on the constants appearing in the problem description. In multi-agent scenarios the problem description is also populated with the agents appearing in the multi-agent society. Hence those constants result in even larger action sets. As the number of actions increases, so does the branching factor of the search space, hence planners employing systematic search methods have been observed to scale poorly. Moreover, it is very often the case that even the simpler single-agent planning problems are unsolvable when the problem description is populated with hundreds of agents. The goal we set in this research is to achieve constant planning performance regardless of the number of agents participating in the society. In this paper we discuss the search space representation and the search method applied to achieve our goal.

Introduction

A STRIPS planning problem is a tuple $P = (A, O, I, G)$ where A is a set of atoms, O is a set of ground operators, $I \subseteq A$ is the initial state and $G \subseteq A$ is the goal state. The set O is populated through a process named operator instantiation which usually takes place before planning begins. During this process the operator schemata forming the domain description are instantiated on the constants appearing in the problem description. In the STRIPS formalism the resulting ground operators (which from now on we will call them actions) are tuples of the form $a = (Prec, Del, Add)$ where the $Prec$, Del, and Add sets are the preconditions, delete effects, and add effects of the action respectively. The actions that form the O set define the state transitions that can take place in the domain, and the larger the O is the wider the search space becomes. Specifically, each such action $a \in O$ defines the transition function:

$$a(s) = s \setminus Del \bigcup Add \tag{1}$$

Given a specific domain, the change in the cardinality of the set O between problems is the result only of the number of constants appearing in those problems, as they all share the same set of operator schemata. Formally, given a domain and a problem description we can define an upper bound for the cardinality of the set O to be

$$\mathcal{O}\left(\sum_{\forall op \in domain} arity(op) * |Constants|\right) \tag{2}$$

where op is an operator schema, $domain$ is the domain description (a set of operator schemata), and $arity(op)$ is the number of parameters that operator op has.

In practice, the constraints that arise through the type of an operator's parameters as well as the semantics of the operator restrict the values that the parameters can take. Hence the cardinality of the set O rarely reaches the above limit. For instance in the Blocks-world domain the operator (stack ?x - block ?y - block) can be instantiated to constants of type block only. Moreover, instantiations of the form (stack A A) are discarded as the domain semantics do not allow a block to be stack on itself.

Although the above limit is rarely reached, the O sets emerging from the planning problems are significantly large. The situation is even worst in multi-agent planning where problem descriptions are populated with constants representing the agents that appear in the society. And since the main interest is in large such societies (Jung 2003) the number of constants and as a result the cardinality of O increases by several orders of magnitude.

At this point let us investigate how some state of the art planners behave as the cardinality of set O increases. Figure 1 portrays their performance on a Blocks-world problem with

initial state: (clear f) (ontable a) (on b a) (on c b) (on d c) (on e d) (on f e) (handempty a1)

goal state: (on d a) (on a e) (on e b) (on b f) (on f c)

Please note that the (handempty) atom is given a parameter representing the actual arm that is free. In the same way, actions are also given an extra parameter in order to reflect which arm performs the corresponding moves. This is a way of introducing agents in this domain. So we have run the planners on the same block configurations in initial and goal states, but with a number of arms varying from 1 to 140.

In Graphplan's (Blum and Furst 1997) performance we can clearly see how the size of O affects the performance of

*This work has been partially supported by the Spanish MEC project TIN2005-08945-C06-05.

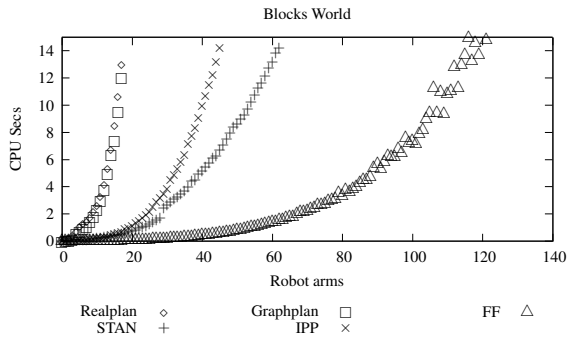


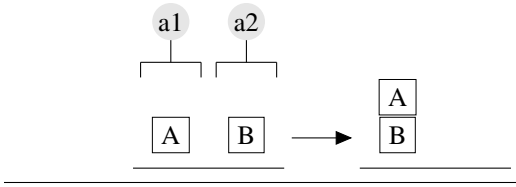
Figure 1: Performance of existing planners.

systematic search. The search scales poorly as larger O sets result in a larger branching factor in the search space. RealPlan (Srivastava, Kambhampati, and Do 2001) is a planner that attacks this problem by decoupling causal and resource reasoning into two independent phases. Hence during causal reasoning an abstract plan is formed which is instantiated during a resource reasoning phase. However, large sets of constants affect the performance of the scheduler as it also performs a search. IPP (Koehler 1999) performs metric planning which treats large number of resources numerically. However, agents cannot be modeled as a numeric resource as each agent is a unique entity. Which leads us to the observation that we can neither treat agents as symmetric entities (STAN (Long and Fox 1999)) as the diversity in their capabilities and their commitments hinders such a classification. Finally, we include FF (Hoffmann and Nebel 2001) which is a heuristic planner that we would expect to scale better as the heuristic estimate is not affected by the number of constants appearing in the problem description. However, all the actions differing only on the agent instantiation appear at the same level in the search tree and are given the same heuristic estimate. Hence the heuristic distribution is noisy and the planner is not well informed.

In our research we set the goal of achieving constant planning performance regardless of the number of agents appearing in the domain description. We focus specifically on agents in order to simplify the discussion but the approach can be easily extended to cover all the constants that can be defined as resources[1]. In this paper we introduce a planner that meets the above objective.

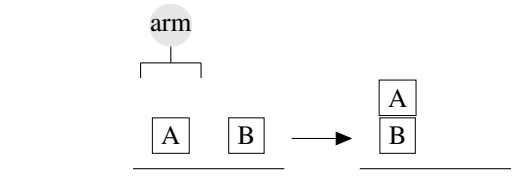
This paper is organised as follows. We start our discussion by looking at an agent classification method based on the capabilities of the agents and an action analysis method applied on the delete effects of the actions. We then focus on the search space representation and the search method that form our planner. Subsequently, we present an intuitive heuristic that deals with superfluous numbers of agents. Having finished with the presentation of the planner we empirically evaluate it and then we conclude this paper.

[1]For a definition of the term resource please consult (Srivastava, Kambhampati, and Do 2001).




```
(define (problem blocks-simplest)
(:domain blocks-world)
(:objects a b a1 a2 arm)
(:init (block a) (ontable a) (clear a)
       (block b) (ontable b) (clear b)
       (handempty a1) (agent a1) (class a1 arm)
       (handempty a2) (agent a2) (class a2 arm))
(:goal (on a b)))
```

Figure 2: A problem description of the Blocks-world domain with two arms.




```
(define (problem blocks-simplest)
(:domain blocks-world)
(:objects a b a1 a2 arm)
(:init (block a) (ontable a) (clear a)
       (block b) (ontable b) (clear b)
       (handempty arm) (agent arm) (class arm arm))
(:goal (on a b)))
```

Figure 3: A lifted problem description of the Blocks-world domain.

Agent classification and action analysis

The purpose of agent classification is twofold:

1. To classify agents based on their capabilities. Within a multi-agent society, agents of different capabilities form teams that promote collaborative effort towards the achievement of a set of goals (Shehory and Kraus 1998). Within such an environment knowledge of who can do what is very welcome, and such a classification makes this knowledge available.
2. To allow the planner of each agent to generate a lifted search space where instead of instantiating operator schemata on specific agents, operator schemata are instantiated on classes of agents. We will explain this process in detail further on.

The process of agent classification can be easily automated as most of the multi-agent frameworks provide a facility where agents register their capabilities (services in multi-agent terminology) (Kalofonos et al. 2006). Hence for each capability registered online the agent can create a list of agents that can provide this capability. However, in this paper we decide to focus on a hand written way of capturing such a classification within a problem description which al-

lows for easier presentation. Moreover, we focus on how such a classification affects the search space of the planner, as our main interest in this paper is on the search method and how it scales.

At this point let us have a look at Figure 2 which portrays a problem description of the Blocks-world domain with two arms. Those arms are captured as agents using the (agent ?x) atom, and classified as 'arm' using the (class ?x ?y) atom. Basically, what this means for the planner is the set $arm = \{a1, a2\}$. Hence now, we can remove from the problem description all the constants referring to agents and replace them with the identifier of the set they belong to, in this case resulting to the problem description shown in Figure 3. Furthermore, during operator instantiation the planner will instantiate operator schemata on set identifiers instead of the actual agent references. In this case the set associated with the set identifier forms the domain of the action's parameter instantiated to the corresponding identifier (Prosser 1993). For example after instantiating the 'stack' operator schema on the abstract problem description the action (stack arm A B) will be formed where the set $arm = \{a1, a2\}$ forms the domain of the first parameter of the action.

As a result, the cardinality of the set O is affected by the diversity on agents' capabilities rather than the agent population, i.e. by the number of such sets. However if there is one to one mapping between agents and capabilities then this would make such a classification irrelevant.

Let us make one final remark on this topic by saying that such a classification can be easily extended over all the constants appearing in the domain description. Having sets of constants with common characteristics allows the planner to expand a search space over sets of constants instead of specific constants, resulting in a search space with a smaller branching factor. Such a classification can be automated with domain analysis tools (Fox and Long 1998). For instance in the gripper domain all the balls appearing in room A could be classified in the set $ballsA = \{a_1, \ldots, a_n\}$. Finally, although in this research we make the assumption that each action has only one such a lifted parameter for simplicity, such an assumption can be easily relaxed.

Let us now shift our attention to the analysis applied on actions. However, before we carry on let us introduce some terminology.

Definition 1 *An atom/action is lifted if some of the atom's/action's parameters are instantiated to a set identifier.*

Definition 2 *An atom/action is grounded if none of the atom's/action's parameters are instantiated to a set identifier.*

Each time an operator schema is instantiated the resulting action is analysed. The analysis is focused on the delete effects of each action and particularly the atoms appearing in the delete set are marked as lifted or grounded based on the instantiation of their parameters. This representation is used during the inference of action conflicts which we discuss in detail later on.

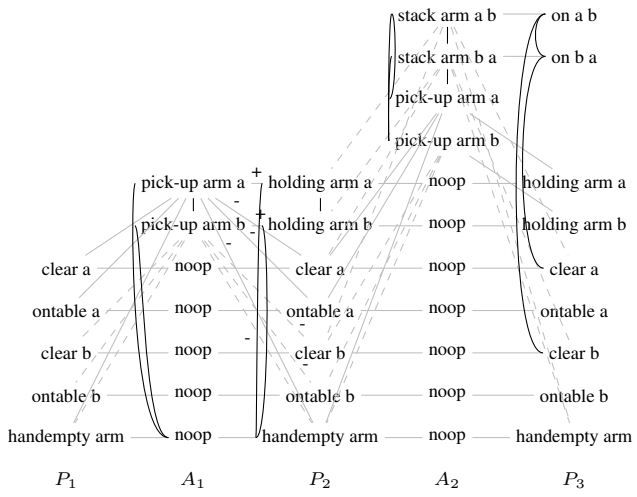


Figure 4: A lifted planning graph (some nodes and edges are omitted for clarity).

The search space

The search space used by our planner is based on the lifted planning graph (Srivastava, Kambhampati, and Do 2001). The lifted planning graph captures only the causal relations of actions, however we augment this representation by populating the graph with binary constraints and exclusivity relations. Hence the resulting graph captures the causal relations of actions, the action conflicts over lifted atoms, and the action conflicts over grounded atoms.

In detail, the lifted planning graph is a layered graph, with each layer (called time-step) consisting of a set of lifted and grounded atoms and a set of lifted and grounded actions (see Figure 4). The atoms that appear at the first time-step of the lifted planning graph (Figure 4:P_1) are the atoms that form the initial state generated during agent classification as previously discussed. Having placed the atoms into the first time-step, the planner populates the action set with the actions whose preconditions are present in the layer. Since the time-step consists of grounded and lifted atoms (for instance in Figure 4:P_1, 'clear a' is grounded while 'handempty arm' is lifted), some of the actions are grounded while others are lifted (for instance in Figure 4:A_1, 'noop clear a' is grounded while 'pick-up arm a' is lifted).

The nodes placed in the lifted planning graph are connected with three types of edge (Figure 4).

Precondition edges: The precondition edges connect an action node with the atoms of the time-step that form the preconditions of the action.

Delete edges: The delete edges connect the action nodes of a time-step n with the atoms appearing at time-step $n + 1$ that constitute the negative effects of the actions.

Add edges: The add edges connect the action nodes of a time-step n with the atoms appearing at time-step $n + 1$ that constitute the positive effects of the actions.

In the case of the classic lifted planning graph at this

point we would stop as all the conflicts between actions are ignored. In our representation though, the lifted planning graph is populated with binary constraints and exclusivity relations through the following rules which are extensions of the corresponding rules found in Graphplan:

Rule of competing needs: The rule states that two actions are marked as mutually exclusive if a precondition of the first is mutually exclusive with a precondition of the second. The extension we have applied is that the actions under consideration inherit the conflict of their preconditions; i.e. when the preconditions are connected with an exclusivity relation the actions are connected with the same relation, while when the preconditions are connected with a binary constraint the actions are also connected with a binary constraint.

Rule of interference: The rule states that two actions are marked as mutually exclusive if one deletes a precondition or positive effect of the other. We extend this rule by utilising the representation of the delete effects of actions previously discussed. Hence when the conflict arises over a lifted delete effect the two actions are linked with a binary constraint, while when the conflict arises over a grounded delete effect the two actions are connected with an exclusivity relation.

As with the action conflicts, the atoms are marked with exclusivity relations and binary constraints. The two types of conflict are captured through the following rule:

Rule of inconsistent support: Two atoms are marked as mutually exclusive if all the actions that achieve the first are mutually exclusive to all the actions that achieve the second. Since we have two types of action conflict (binary constraints and exclusivity relations) the test operates in the following fashion:

- When all the pairs of actions that achieve the atoms in question are linked with an exclusivity relation, the two atoms are linked with an exclusivity relation as well.
- If there is at least one pair of actions that achieve the atoms in question which are connected with a binary constraint, then the two atoms are linked with a binary constraint.
- If there is at least one *unconnected* pair of actions that achieve the atoms in question, then the atoms are left unconnected.

Consequently, two actions may be connected with both types of conflict, while two atoms of a given layer with only one. The difference between binary constraints and exclusivity relations is that two actions may be applied in parallel if the binary constraint is relaxed, but they can never be applied in parallel when an exclusivity relation holds. Hence, when both conflicts emerge between two actions the exclusivity relation has higher precedence over the binary constraint and the planner never tries to relax the binary constraint in that case.

The search algorithm

The search algorithm is a direct extension of the algorithm used in Graphplan. The search backtracks in a chronologi-

```
1: domain-size ← 2; conflicts ← ∅; backjumping ← FALSE;
       plan ← a time-step containing the goals of the problem;
2:
3: FUNCTION search( time-step ): Boolean
4: BEGIN
5: if time-step ⩽ 0 then return TRUE;
6: atom-layer ← goal set in the first layer of the plan;
7: action-layer ← action set in the first layer of the plan;
8: goal ← an unsupported atom in atom-layer ;
9: actions-add ← set of actions from graph[time-step −1] that support goal;
10: for all act ∈ actions-add do
11:     mutex-free ← mutexFree( action-layer ⋃ {act} );
12:     if mutex-free = FALSE then continue;
13:     backjumping ← FALSE;                                  # progressing #
14:     action-layer ← action-layer ⋃ {act};
15:     if domain-size < |action-layer| then
16:         domain-size ← |action-layer|;
17:     domain ← {y_1, ..., y_n : y ∈ domain of act, n = domain-size };
18:     repeat
19:         if backjumping = TRUE then
20:             if conflicts ≠ ∅ ∧ act ≠ x_1 : x ∈ conflicts then
21:                 result ← FALSE;
22:                 break;
23:             else
24:                 conflicts ← conflicts \ {x_1};
25:                 backjumping ← FALSE;                      # progressing #
26:         if the instance of goal ≠ ⊥ then
27:             domain ← domain ⋂ goal instance;
28:         act is instantiated to y_1 : y_1 ∈ domain;
29:         domain ← domain \ {y_1};
30:         if consistent( action-layer ) = FALSE then continue;
                                          # consistent(·) populates conflicts #
31:         if there are unsupported atoms in atom-layer then
32:             result ← search( time-step );
33:         else
34:             plan[time-step −1] ← preconditions of the actions in action-
                    layer;
35:             result ← search( time-step −1 );
36:             if result = FALSE then remove the first layer from plan;
37:         if result = TRUE then break;
38:     until domain = ∅;
39:     if result = TRUE then break;
40:     else action-layer ← action-layer \{act};
41: if result = FALSE then backjumping ← TRUE;
                        # with the result being false and having exhausted all
    the actions in the current level and their domains, we set backjumping to true and
    we trace our steps back until the conditions for progressing are met again, that is
    either picking up a new action (first progressing comment), or trying a new value
    for an action in the conflict set (second progressing comment) #
42: return result;
43: END
```

Figure 5: Pseudo-code description of the search method.

cal fashion over action selections while it is capable of performing conflict-directed backjumping (Prosser 1993) over agent allocations. The search traverses the lifted planning graph layer by layer starting from the goals and going backwards. Figure 5 presents a pseudo-code description of the main function of the algorithm.

The parameter of the function is the index of the time-step that forms the frontier of the search, initially set to the last time-step of the planning graph (Figure 5:3). When the search reaches time-step zero it returns successfully (Figure 5:5). The function starts by initialising *atom-layer* to the layer of the plan that contains the current goal set (Figure 5:6), *action-layer* to the layer of the plan where the actions achieving the above goals should be placed (Figure 5:7), *goal* to an atom from atom-layer (Figure 5:8), and *actions-add* to the set of actions from the graph that add the goal (Figure 5:9).

The function consists of two nested loops, the first (Figure 5:10 – 40) iterates through the actions that support the goal while the second (Figure 5:18 – 38) iterates through the values of the domain of the last selected action.

For each action selected, first we check if the action is mutex-free with all the actions previously selected (Figure 5:11). It is very important to note that the mutex-free check considers only the exclusivity relations that appear among the actions. If the action set is not mutex-free we move on directly to the next available action or we backtrack if the set is exhausted (Figure 5:12). If the mutex-free test returns successfully, the picked up action is added to the action layer (Figure 5:14), and the domain of the action is obtained (Figure 5:17).

At this point let us have a look at Figure 5:15 – 16. Using this test we set the variable 'domain-size' to the size of the largest local set of actions considered so far in the search. This value is used to restrict the size of the domain of each action considered to a subset of size equal to the value of 'domain-size'. The concept behind this test is that given a set of conflicting actions those actions can be performed in parallel if they are allocated (in the worst case) a unique agent as the binary constraints will be satisfied. Hence considering any set of agents larger than the current set of actions is redundant. We will investigate this in more detail later on.

The second loop expands over lines 18 – 38 of Figure 5. At first the domain of the last picked up action is further refined. When the action achieves a goal that is lifted, the domain of the action becomes the intersection of the original set and the set consisting only of the goal allocation (Figure 5:26 – 27), thus the action can be given only one value which is that of the goal it supports. Otherwise the domain is left intact. In this way we ensure that the causal dependencies between actions are preserved as their allocations are propagated through their preconditions. Subsequently a constraints consistency check is performed (Figure 5:30). The check fails when violated constraints or uninstantiated actions are met; actions might be uninstantiated at this point only if their domain is empty due to the fact that it has been exhausted through the previous trials, or the intersection of lines 26 – 27 is empty, or when there are no agents for a given action. During this check whenever a constraint violation is met the target action node is stored in the conflict set of the newly instantiated action. During backtracking the algorithm tries a new agent allocation for a given action only when the action appears in the conflict set (Figure 5:24 – 25). Otherwise the algorithm immediately backtracks further (backjumps) ignoring the remaining agent allocations for an action not appearing in the conflict set (Figure 5:20 – 22). This decision can be justified by considering the following. If an action does not appear in the conflict set then this action's allocation does not interfere with any allocations made subsequently. Hence the action's allocation is not the reason why the search could not progress from the last allocation made. Hence trying a new value for this action is pointless. This is the main observation behind CBJ (Prosser 1993).

The following code fragment in lines 31 – 36 is the recursive point of the algorithm. The planner moves to the next unsupported atom from the current atom-layer if one exists (Figure 5:32). Otherwise, it creates a new atom-layer by collecting the preconditions of the selected actions (Figure 5:34) and starts working on the newly created layer (Figure 5:35).

It is important to pinpoint the reasons why an action may end in the conflict set.

- An action already appearing in the layer is given a value that violates the constraint with the newly inserted into the layer action. That action is added into the conflict set as the search might have to revisit it and change its value.
- An action is added in the conflict set if it is the root of a dead-end tree. Let us investigate this in more detail. As we said above in the case in which an action supports a lifted goal inherits the value of the goal. This is done in order to preserve the causal links between actions. If this allocation leads to a dead-end the value of that specific goal needs to be undone. However, that goal is a precondition of some other action and so on. Hence we need to follow this action-goal path until we find the action that was first instantiated to this erroneous value, i.e. the root of the tree. The action located at the root of this tree is placed in the conflict set. However, this process of traversing up the tree every time a conflict is met is expensive, hence in practice every time an action inherits the value of a goal it also inherits a reference to the action that is the root of that specific tree.

Resource curtailment

In the previous section we briefly discussed the fragment of Figure 5:15 – 16. We stated that the purpose of this fragment is to limit the size of the domain of each action to the size of the largest local set of actions considered so far in the search. This is an intuitive heuristic that relies on the following observation.

Let us suppose that action x is the last action inserted into set A. Let us also assume that action x is on a path leading to the goals and that the domain of x is $Dom = \{a_1, ..., a_w\}$. Since action x is on a valid path, x can be given a value from the set Dom that satisfies all the constraints with the actions in A that is linked against. The value given to x will have an

index $k : 1 \leq k \leq w$. In the worst case, all the actions in A are connected with a binary constraint hence a unique value allocation for each action is needed for all the constraints to be satisfied. Consequently, the highest value given to k will never exceed the cardinality of the action set considered ($k \leq |A|$), and let us suppose that in this case k is indeed given the highest value ($k = |A|$). In this setting the search would have to try out the first k values from Dom before the right assignment is found that meets all the constraints, i.e. the set tried out is $D_1 = \{a_1, \ldots, a_k : a \in Dom\}$.

Now let us assume that x is on a path that is a dead-end and that the search will need to backtrack over x. The search can start backtracking only after all the values from the set Dom have been tried out. In this case the search will try out the values $D_1 = \{a_1, \ldots, a_k : a \in Dom\}$ as well as the values $D_2 = \{a_{k+1}, \ldots, a_w : a \in Dom\}$.

D_1 is a set whose size is restricted by the number of constraints linking the actions considered. D_2 is a result of the equation $D_2 = Dom \setminus D_1$ hence its size is $|D_2| = |Dom| - |D_1|$. It is obvious at this point that the size of D_2 will increase as Dom becomes larger, i.e. more agents are inserted in the domain.

With resource curtailment we aim at the restriction of the size of the set D_2. For this purpose we use the variable 'domain-size' (ds for sort) and we set it to the cardinality of the largest local set of actions considered so far, i.e

$$ds = \forall A \; max|A| \tag{3}$$

This defines an upper bound for k of the form

$$1 \leq k \leq ds \tag{4}$$

When the number of agents significantly exceeds ds the upper bound will usually be significantly smaller than w

$$1 \leq k \leq ds \ll w \tag{5}$$

and

$$Dom = \{a_1, \ldots, a_k, \ldots, a_{ds}, \ldots, a_w\} \tag{6}$$

while when ds exceeds the number of agents the whole set Dom is considered, i.e.

$$1 \leq k \leq ds = w \tag{7}$$

and

$$Dom = \{a_1, \ldots, a_k, \ldots, a_w\} \tag{8}$$

which is usually the case when the agent population is very small.

Given a set of actions X and ds, the sets D_1 and D_2 for the set X are defined as shown below.

$$f(X) = \begin{cases} D_1 = \{a_1, \ldots, a_{|X|}\}, \\ D_2 = \{a_{|X|+1}, \ldots, a_{ds}\} & \text{if } |X| < ds \\ D_1 = \{a_1, \ldots, a_{ds}\}, \\ D_2 = \emptyset & \text{if } |X| = ds \end{cases} \tag{9}$$

In this case the remaining values from Dom are discarded. The set of the remaining values can be defined as

Blocks World	Elevator
$0x + 0$	$0x + 0$
$0x + 0.02$	$0x + 0$
$0x + 0.02$	$0x + 0$
$0x + 0.01$	$0x + 0.01$
$0x + 0.06$	$0x + 0.01$
$0x + 0.08$	$0x + 0.01$
$0x + 0.12$	$0x + 0.01$
$0x + 0.27$	$0x + 0.01$
$0x + 0.25$	$0x + 0.06$
$0x + 0.38$	$0x + 0.06$
$0x + 0.80$	$0x + 0.06$
$0x + 0.80$	$0x + 0.07$
$0x + 1.26$	$0x + 0.07$
$0x + 6.29$	—
$0.01x + 25.45$	—
$0x + 1.73$	—

Table 1: Regression lines of the data collected by running the planner on each problem with the number of agents varying from 2-200.

$$D_3 = Dom \setminus (D_1 \bigcup D_2) \tag{10}$$

Since both D_1 and D_2 are bound to a value that does not change with the number of agents present in the domain, the search is not going to be affected by the size of the agent population. Instead the search is affected only by the size of the action sets considered.

Empirical evaluation

The evaluation is targeted towards the search method of the planner, as this is the main focus of the paper. Hence throughout the evaluation instead of referring directly to the planner as a whole we investigate the following flavours of the planner.

CBJ+RC: This is the planner on its fullest. This flavour consists of the search method described in this paper as well as the resource curtailment heuristic.

CBJ: This flavour consists of the search method discussed in this paper, but without the resource curtailment heuristic.

CB+RC: This flavour consists of the search method discussed in this paper without the fragment related with backjumping (lines 19 – 25 of Figure 5). Hence this search backtracks in a chronological fashion over both action selections and value allocations. Moreover, this method makes use of the resource curtailment heuristic.

CB: This flavour consists of the search method without backjumping as discussed above, and without the resource curtailment heuristic. This flavour is identical to the search performed by Graphplan.

The domains we use in this evaluation are extensions of the Elevator, Logistics and Blocks-world from IPC_2, Gripper from IPC_1, and Shuttle which is distributed with the Realplan planner. The extension we applied is the introduction

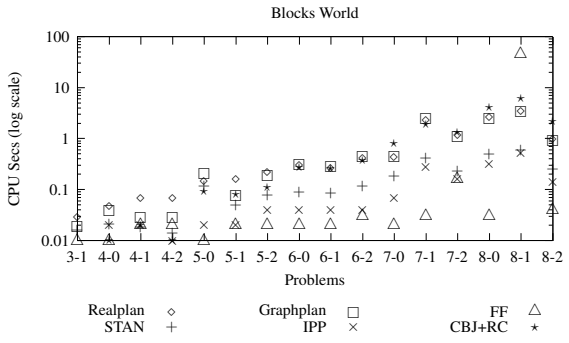


Figure 6: Comparison of the performance of the planners on problems from the Blocks-world domain (16 problems in total).

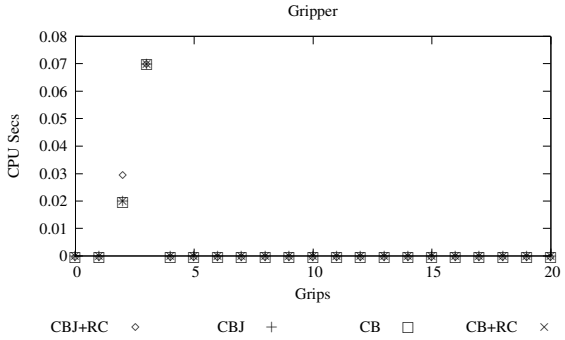


Figure 9: Comparison of the performance of the four search flavours (20 problems in total).

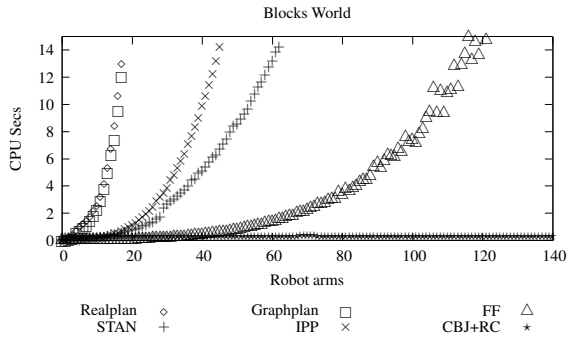


Figure 7: Comparison of the performance of the planners on the shuffle domain instance (140 problems in total).

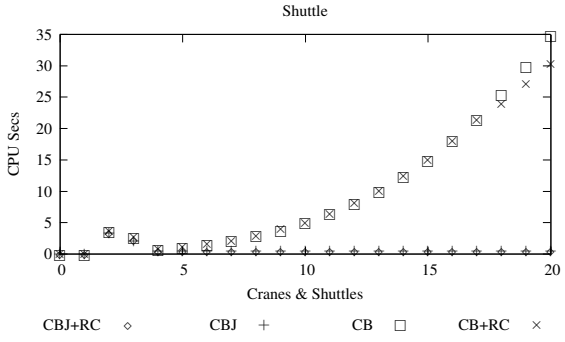


Figure 10: Comparison of the performance of the four search flavours (20 problems in total).

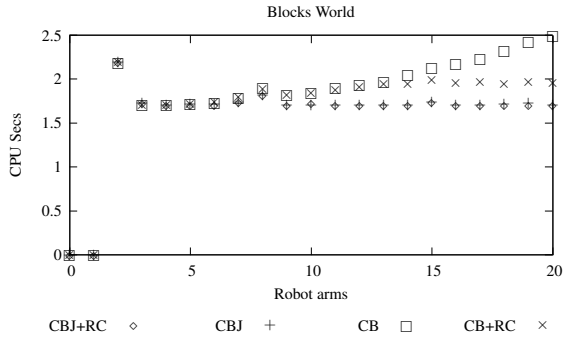


Figure 8: Comparison of the performance of the four search flavours (20 problems in total).

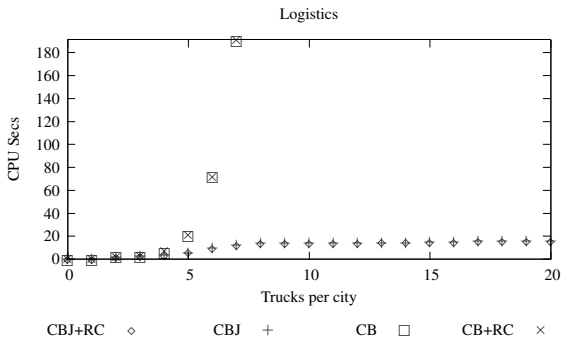


Figure 11: Comparison of the performance of the four search flavours (20 problems in total).

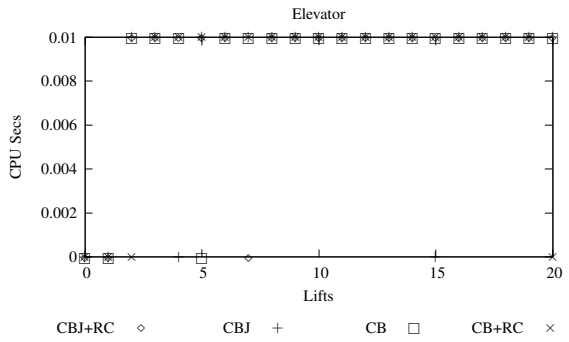


Figure 12: Comparison of the performance of the four search flavours (20 problems in total).

of numerous agents. The fact that the domains are trivial to solve allows for the exposure of the complexity due to a large branching factor in the search space.

We evaluated the four flavours of the planner presented above in the following way. We picked up a random problem from each of the domains presented above, and we made numerous versions of it, each version consisting of a different number of agents. Then we run the planner on those problems and we collected the results. Before we move on, we would like to remind the reader that in this research we are not only interested in good performance but we have set the goal of achieving *constant* performance per problem regardless of the number of agents appearing in it. Let us now present the results.

In Figure 6 we present the results from the comparison of the performance of Realplan, STAN, Graphplan, IPP, FF, and CBJ+RC in various problems of the Blocks-world domain with a constant number of agents set to two. This is the only case in which we do that as we want to show that the planner still scales exponentially as the difficulty of the problem increases. Figure 7 is the figure we have already seen in the introduction of this paper, in which we have added the results from running CBJ+RC. We have done this in order to place the planner in context with existing work, and to show that even though the problem is trivial to solve indeed the number of agents appearing in it can transform the problem into a hard one. Hence constant performance is very beneficial.

In table 1 we summarise the lines of the data we have collected by running the planner on several problems from the Blocks-world and Elevator domains, varying the agents appearing in each problem from 2 – 200. From the slopes of the lines presented we understand that the planner achieves constant performance regardless of the agents appearing in those problems. From the intercepts we understand that the planner scales exponentially as the problems become harder.

From now on we will focus on the four flavours of the planner and we will directly compare them. Figures 8, 9, 10, 11, 12 portray the results. Mainly two points are interesting in these results. First that the resource curtailment heuristic contributes even when backjumping is not used. Hence CB+RC, which in other words is Graphplan's search with resource curtailment scales very well. And when CBJ is combined with RC then the planner is able to achieve this constant performance. The second interesting observation is that CBJ without the RC heuristic still performs very well and in most of our experiments performs equally to CBJ+RC. This is due to the fact that the results presented in the above figures cover up to 20 agents hence the overhead of considering the whole domain of each action is quite small. However, as the number of agents considered increases we expect CBJ+RC to significantly surpass CBJ in performance.

Conclusion

In this research we have set the goal of achieving constant planning performance regardless of the number of agents appearing in the problem description. In this paper we have mainly focused on the search space representation used and the search method with its heuristic applied to achieve our goal. Through the empirical evaluation we have shown that the planner's performance is indeed unaffected by the number of agents appearing in the problems.

References

Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281 – 300.

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

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

Jung, H. 2003. *Conflict resolution strategies and their performance models for large-scale multiagent systems*. Ph.D. Dissertation, University of Southern California.

Kalofonos, D.; Karunatillake, N.; Jennings, N. R.; Norman, T. J.; Reed, C.; and Wells, S. 2006. Building agents that plan and argue in a social context. In *Proceedings of the 1st International Conference on Computational Models of Argument*.

Koehler, J. 1999. Metric planning using planning graphs - a first investigation. Technical Report 127, Institute for Computer Science, Albert Ludwigs University.

Long, D., and Fox, M. 1999. Efficient implementation of the plan graph in STAN. *Journal of Artificial Intelligence Research* 10:87 – 115.

Prosser, P. 1993. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence* 9(3):268 – 299.

Shehory, O., and Kraus, S. 1998. Methods for task allocation via agent coalition formation. *Artificial Intelligence* 101(1-2):165 – 200.

Srivastava, B.; Kambhampati, S.; and Do, M. B. 2001. Planning the project management way: Efficient planning by effective integration of causal and resource reasoning in Realplan. *Artificial Intelligence* 131(1–2):73 – 134.