Forward Search with Backward Analysis

Shlomi Maliah
Information Systems Engineering
Ben Gurion University
shlomima@post.bgu.ac.il

Ronen I. Brafman
Computer Science Dept.
Ben Gurion University
brafman@cs.bgu.ac.il

Guy Shani
Information Systems Engineering
Ben Gurion University
shanigu@bgu.ac.il

Abstract

We describe a new forward search algorithm for classical planning. This algorithm attempts to maintain a focused search, expanding states using only a subset of the possible actions. Given a state s' that was obtained by applying action a to state s, we prefer to apply in s' only actions a' that require some effect of a which we call *forward* actions. As this is incomplete, we must also consider actions a'' that supply some other precondition of a' and actions a''' that supply preconditions to a'' and so on. We call these *backward* actions, as identifying the relevant actions requires backward reasoning. We show that by giving high priority to the forward actions a' we get improved performance in many domains. The resulting algorithm can be viewed as building on the classic idea of means-ends analysis [Newell and Simon, 1961]. One crucial open problem that arises is how to prioritize the search for backward actions.

1 Introduction

Planning typically requires achieving multiple goals stemming from the existence of multiple sub-goals or multiple preconditions. Unless the plans for these subgoals interact strongly with each other, this usually implies that we have flexibility in ordering them. This, in turn, implies that often, there are multiple permutations of a plan that are also valid plans. Ideally, we would like our search algorithm not to consider alternative permutations. In this paper we formulate a forward search algorithm that uses backwards reasoning, in the spirit of means-ends-analysis [Newell and Simon, 1961], to focus only on certain permutations. More specifically, we try to consider only permutations in which work done for one subgoal is not interleaved with work done for another subgoal.

Thus, while inconsistent with the jittery age we live in, our search process aims to be focused – it tries to focus on achieving one sub-goal at a time. Ideally, if we just applied an action a, we would like the next action to be relevant to it and use one of the effects a. We call such actions *forward* actions.

Forward search with this pruning rule can drastically reduce the branching factor, and solves quite a few classical planning benchmarks. Unfortunately, it is easily seen to be incomplete: Suppose a_1, a_2, a_3 is a solution plan where: a_1, a_2 have some precondition that is true initially, and generate p_1 and p_2 respectively; a_3 requires both p_1 and p_2 and produces the goal. Suppose I is the initial state and we generate $a_1(I)$. At this point, the only action that uses an effect of a_1 is a_3, but it is not applicable.

Action a_3 is not applicable after a_1 because its other precondition, p_2, does not hold. We need to modify the pruning rule so that it allows actions, such as a_2, that establish the missing precondition p_2 of a_1, given $a_1(I)$. We call these *backwards* actions. But a single backwards action may be insufficient. What if a_2 is not applicable after a_1 because one of its preconditions p_3 does not hold? Establishing p_2 may actually require a sub-plan, and this requires a form of backwards relevance reasoning.

Thus, the essence of our algorithm is to move forward using forward actions. When such an action is inapplicable because of a missing precondition, we reason backwards and find a sub-plan that achieves the missing precondition.[1] To make the algorithm efficient, we prioritize forward actions over backward reasoning.

We present the results of a planner based on prioritized forward search with backwards analysis. The results are mixed – sometimes, our algorithm works better than naive forward search, and sometimes worse. It leads to an interesting open question on how to prioritize the expansion of different actions.

2 Forward Backward Search

We consider standard classical planning problems, represented by a tuple $\langle \mathcal{P}, A, I, G \rangle$ where:

- $\mathcal{P}$ is a finite set of primitive propositions (facts).
- A is the action set.
- I is the start state.
- G is the goal condition.

Each action $a = \langle pre(a), \mathit{eff}(a) \rangle$ is defined by its preconditions ($pre(a)$), and effects ($\mathit{eff}(a)$). Preconditions and effects are conjunctions of primitive propositions and literals,

[1]Note that backwards refers to the reasoning mode. Actions are always applied forward.

respectively. A state is a truth assignment over $\mathcal{P}$. G is a conjunction of facts. $a(s)$ denotes the result of applying action a to state s. A *plan* $\pi = (a_1, \ldots, a_k)$ is a solution to a planning task iff $a_k(\ldots(a_1(I)\ldots) \models G$.

An important assumption we make is that actions are in *transition normal form* [Pommerening and Helmert, 2015]. That is, a primitive proposition (or its negation) appears in a precondition iff it (or its negation) appears in the effect of the action. Every problem is easily converted into transition normal form.

2.1 Forward-Backward Search

Forward-backward search is a forward search algorithm with action pruning. Algorithm 1 shows the pseudo-code of its initial version, denoted FBS1. It maintains two open lists which we call the *forward list,* denoted l_{fwd}, and the *backward list*, denoted l_{bwd}. The forward list contains pairs of the form $\langle s, P\rangle$, where s is a state and P is a set of primitive propositions. We can expand states in the forward open list only using actions that have a precondition in P. Initially, this list contains all elements of the form $\langle a(I), \mathit{eff}(a)\rangle$, where a is any action applicable in I.

Unlike regular forward search, confined to actions with satisfied preconditions, we also consider actions a that have a precondition in P but are not applicable in s. We set up a process which attempts to find an action, or possibly a sequence of actions, that achieve the missing preconditions of a. This is done by inserting the pair $\langle s, [a]\rangle$ to the backwards list.

The backward open list contains pairs of the form $\langle s, \mathit{stk}\rangle$, where s is a state and *stk* is a stack of actions. If a appears in the top of the stack and a is applicable in s, we apply a and remove it from the stk, obtaining stk'. If the latter is empty, $\langle a(s), \mathit{eff}(a)\rangle$ is added to the forward list. At this point, we successfully generated a sub-plan that achieved the missing preconditions of a, and can continue forward. Otherwise, $\langle a(s), \mathit{stk}'\rangle$ is added to the backward list. This will allow us to continue and apply the following actions, or potentially add new actions that achieve preconditions that are still missing.

If a appears in the top of the stack and a is inapplicable in s, then we consider all actions a' that achieve a missing precondition of a. For each such action we add a new item into the backward list. This item is identical to the original pair, but with a' pushed into the top of the stack. That is, we continue to reason backwards seeking an action that can help us to achieve a needed precondition.

We denote the above algorithm by FBS1. When expanding a node from one of the lists (lines 10, 23), we select nodes that are minimal in terms of the heuristic value of their state. Below we explain how we optimize the choice between the two lists.

In FBS1, an action was inserted into a stack in the backward list if it supplied a missing precondition of a relevant action. We denote by FBS2 a slightly modified version of the above in which an action is inserted into the backward list even if it supplies a precondition that is currently true, and even if the action that requires this precondition is applicable. In terms of the pseudo-code, the *else* parts starting in line 16 and line 30 are always executed, and with $P' = \mathit{pre}(a)$. As we show later, FBS1 is incomplete, whereas FBS2 is complete.

Algorithm 1: The FwdBwd Algorithm

```
1  FwdBwd()
2      $l_{fwd} \leftarrow$ the empty list
3      $l_{bwd} \leftarrow$ the empty list
4      foreach Action $a$ executable at $I$ do
5          Add $\langle a(I), \mathit{eff}(a)\rangle$ to $l_{fwd}$
6      while goal not achieved do
7          ExpandForward($l_{fwd}$)
8          ExpandBackward($l_{bwd}$)
9  ExpandForward($l_{fwd}$)
10     $\langle s, P\rangle \leftarrow$ extract min from $l_{fwd}$
11     if $s$ is a goal state then
12         trace back solution and terminate
13     foreach $a \in A$ s.t. $\mathit{pre}(a) \cap P \neq \emptyset$ do
14         if $s \models \mathit{pre}(a)$ then
15             Add $\langle a(s), \mathit{eff}(a)\rangle$ to $l_{fwd}$
16         else
17             $P' \leftarrow \mathit{pre}(a) \setminus s$
18             foreach $a' \in A$ s.t. $\mathit{eff}(a') \cap P' \neq \emptyset$ do
19                 $\mathit{stack} \leftarrow$ the empty stack
20                 Push $a'$ into $\mathit{stack}$
21                 Add $\langle s, \mathit{stack}\rangle$ to $l_{bwd}$
22 ExpandBackward($l_{bwd}$)
23     $\langle s, \mathit{stack}\rangle \leftarrow$ extract min from $l_{bwd}$
24     Pop $a$ from $\mathit{stack}$
25     if $s \models \mathit{pre}(a)$ then
26         if stack is empty then
27             Add $\langle a(s), \mathit{eff}(a)\rangle$ to $l_{fwd}$
28         else
29             Add $\langle a(s), \mathit{stack}\rangle$ to $l_{bwd}$
30     else
31         Push $a$ into $\mathit{stack}$
32         $P' \leftarrow \mathit{pre}(a) \setminus s$
33         foreach $a' \in A$ s.t. $\mathit{eff}(a') \cap P' \neq \emptyset$ do
34             $\mathit{copy} \leftarrow$ a copy of $\mathit{stack}$
35             Push $a'$ into $\mathit{copy}$
36             Add $\langle s, \mathit{copy}\rangle$ to $l_{bwd}$
```

Optimizations

First, as in most search algorithms, it is useful to avoid repeated visits to the same state. In our case, this is somewhat more complicated, because, e.g., the same state can be visited many times, following different actions. Still, it is straightforward to add bookkeeping mechanisms to Algorithm 1 to avoid adding duplicates to the forward and backward lists.

Intuitively, forward actions advance the plan towards the goal, while backward actions are a necessary setback because a needed action cannot be executed. Following this intuition, we can give priority to actions that use an effect of the last action. That is, in the main loop of the FwdBwd algorithm, we expand more states from the forward list than from the backward list.

Similarly, the search backwards for relevant actions can distinguish between actions that supply a condition that is untrue at present (as in FBS1) and actions that supply a condition that is true at present (allowed by FBS2). The latter can be given lower priority, ensuring completeness, while having little effect on computation time.

In many domains the goal is a conjunction of several facts. The algorithm, as described, will not be able to handle such goals because it cannot search forward once a sub-goal is achieved (when subgoals are independent). One way to avoid this is to add an artificial action that takes as precondition all these facts, supplying a single artificial goal fact. Using this technique in our algorithm, however, is problematic. This is because in many domains once a subgoal is achieved, the entire planning process is turned into a backward analysis in order to obtain the missing goal facts. We can overcome this by using a "reset" whenever a goal fact is achieved, allowing all executable actions to be executed in the following state, as we do for the initial state.

The backward search is very expensive in our implementation because we consider all actions that achieve a precondition, and it is unclear how to heuristically rank these actions. Heuristics that rely on the current state are not informative for these unexecuted actions. We now suggest a third version of our algorithm that avoids the backward search altogether.

In forwards-backwards search 3 (FBS3), when an action a' has a precondition supplied by the previous action a, but cannot be executed at the current state s, we find all actions a'' that are relevant to a', and can be executed at s. This can be done by regressing the preconditions of a', terminating whenever reaching actions that can be executed at s. All these actions are then executed, and the resulting pairs are inserted into the forward list. We still maintain a backward list, to allow us to prioritize forward expansions over backward analysis, but the backward list no longer contains a stack of actions, only pairs $\langle s, P \rangle$, where P is the set of facts to regress.

The result is a less focused algorithm, because we no longer maintain the "reasons" for the backward analysis, but one that avoids the problematic prioritization of backward expansions. This also avoids the special treatment after achieving goal facts, because the backward regression is similar to the "reset" operation, although more focused.

3 Properties

We now discuss the soundness and completeness of FBS.

Claim 1. FBS *is sound.*

Proof. Each state in a pair $\langle s, X \rangle$ (where X is either P or $stack$), generated in FBS is obtained by applying an action to a state that was previously generated, starting at the initial state. Thus, all generated states are reachable, and if a goal state is found, there must be a plan. □

As noted earlier FBS1 is incomplete, and we provide a counter-example later. We now prove, though, that FBS2, that uses backward analysis even when preconditions are satisfied, is complete.

It remains a key open question whether weaker conditions suffice to ensure completeness. In strong stubborn sets [Wehrle and Helmert, 2014], for example, it is sufficient to move backward only over a single precondition of an action, considering also actions that interfere with a needed precondition. We conjecture that by using a similar condition in FBS1, we can attain completeness. Specifically, given $\langle s, stk \rangle$, if a is at the top of the stack and it is applicable in s, and a' is an action that interferes with a, we also add $\langle s, stk' \rangle$ where stk' is obtained by pushing a' to stk. In practice, when the optimizations described earlier are used over current benchmarks, we never expand backwards states that were added for satisfied preconditions.

For our completeness proof, we assume that the goal is a single proposition, which can be achieved with a simple transformation.

We use the following definitions in our proofs: The *causal structure* of a valid plan π [Karpas and Domshlak, 2012], denoted $CS(\pi)$ is a DAG whose nodes are the actions of π. a is a parent of a' iff a precedes a' in the π, and a has an effect, say p, that is a precondition of a', and no action between a and a' produces p. This is often called a causal link between a and a' in π [Tate, 1977]. As we assume that the goal is a single literal, there is a single leaf node in $CS(\pi)$. We use $InvCS(\pi)$ to denote $CS(\pi)$ with edge directions reversed, which by the above is a DAG with a single root node. Finally, we say that a plan is *minimal* if whenever any subset of action instances is removed from the plan, it is no longer a valid plan (i.e., it is either not executable or does not achieve the goal).

Lemma 1. *Let a, a' be two actions in a plan π such that a precedes (not necessarily immediately) a' in π, and p appears in the description of a and a' (possibly negated). Then, a is an ancestor of a' in $CS(\pi)$.*

Proof. The proof is by induction on the number of actions between a and a' in π in whose description p appears. First, suppose that there are no such actions. Because we assume action descriptions are in transition normal form, then p (possibly negated) appears in both the preconditions and effects of a and a'. Therefore, a must supply the correct value of p to a'. Consequently, by definition, a is a parent of a' in $CS(\pi)$.

For the inductive step, suppose the above holds when there are k actions between a and a', and consider the case where there are exactly $k+1$ actions, $a^1, a^2, \dots, a^{k+1}$ between a and a' that contain p in their description. By the inductive hypothesis, a^1 is an ancestor of a', and by the argument above, a is a parent of a^1, and thus, an ancestor of a'. □

An immediate consequence of the above Lemma and the definition of post-order traversal of a graph is:

Lemma 2. *The order of actions that mention p in their descriptions in any post-order traversal of $InvCS(\pi)$ is identical.*

Proof. By Lemma 1, every two actions that mention p have an ancestor/descendant relation, which must be maintained in any post-order traversal. □

Lemma 3. *Every post-order traversal of $InvCS(\pi)$ is a valid plan.*

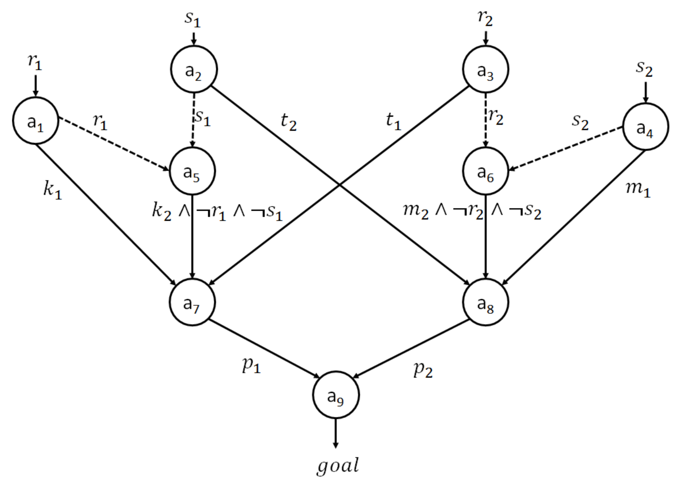

Figure 1: Counter Example

Proof. In any post-order traversal of the graph, for every action a, the relative order of all actions supplying a with some precondition must be the same. Therefore, the value of the propositions in the precondition of a prior to the execution of a will be identical to their value prior to the execution of a in π, and therefore, the preconditions of a are satisfied and a is executable, and hence the entire sequence is executable, and in particular the last action that achieves the goal. □

We now prove:

Theorem 1. FBS2 *is complete.*

Proof. Suppose a planning problem is solvable. Let π be such a plan. We show that FBS2 generates a post-order traversal $InvCS(\pi)$, which by Lemma 3 is a plan.

We start with the first action in π. It must be a leaf node of $InvCS(\pi)$. Given this leaf node, a, after executing it, we apply actions forward until we reach a_p, the first ancestor of a that has other children. At this point, we would like to apply a descendant of the other children of a_p. Let a_l be such a descendant that is a leaf node. In FBS2, we are guaranteed that this action is considered in the current state. Next, we apply a_l, and continue with its parent, until we apply the relevant child of a_p. Note that a_l must be applicable since it is a leaf node and we are assuming TNF. □

We end this section with a counter-example to the completeness of FBS1 (Figure 1). In this example we see a plan with nine actions $a_1, \cdots, a_9$, where a_9 achieves the goal. a_1, a_2, a_3, a_4 are applicable in the initial state and require the preconditions r_1, s_1, r_2, s_2, respectively. They do not delete these preconditions. Recall that in transition normal form, this implies that these preconditions are also their effects (denoted by the dotted edges). a_5 requires r_1 and s_1, and a_6 requires r_2 and s_2 as preconditions.

The key difficulty in this example is that a_5 and a_6 remove necessary preconditions for a_1, a_2, a_3, a_4 that cannot be later generated. These actions must be executed before actions a_7 and a_8. Hence, a_1, a_2 must be executed before a_5, and a_3, a_4 before a_6. However, executions that follow the FBS1 algorithms always execute either a_5 or a_6 before some of the actions $a_1, ..., a_4$, as illustrated below.

Focusing on the left side (the right side is symmetric), the execution can start with either a_1, a_2 or a_5, whose preconditions are satisfied initially. If we start with a_5, r_1, s_1 are deleted, blocking the execution of both a_1 and a_2 which produce required propositions for later actions.

Consider the execution in which first apply a_1. There are two actions that use an effect of a_1: a_5 and a_7. a_5 can be immediately executed without backward reasoning, and this implies that a_2 is blocked, and will not be considered by the algorithm. Without it, a_8 cannot be executed later.

Another option we can consider is to apply a_7 after a_1, applying backward reasoning, and then a_3. After a_3 we can apply a_6 which blocks a_4, or a_5 which blocks a_2 again.

On the other hand, FBS2 will apply backward reasoning from a_5 even though its preconditions are satisfied, and will discover the path executing a_2 before a_5.

It is interesting to note that we did not manage to generate a smaller and simpler counter example, which may point to the rarity of domains for which FBS1 is incomplete. When using the stubborn sets rule of moving backwards to interfering actions, this counter example is no longer valid.

4 Empirical Evaluation

We now provide an empirical analysis of our FwdBwd algorithm, comparing it to naive forward heuristic search. To provide a clean analysis of our new approach, we avoided comparison to mature classical planners, containing many optimizations, and implemented all algorithms on an identical framework. As such, differences between algorithms result from their properties, not from better implementation.

We experiment with two main heuristics — the FF heuristic, and preferred operators [Hoffmann, 2001]. The FF heuristic is a very popular and effective heuristic, analyzing the number of actions in a plan over a delete relaxation of the original problem. The preferred operators heuristic gives priority to actions in the relaxed plan. We find it important to compare to preferred operators, because this technique also restricts the set of actions that are considered at each state. The original preferred heuristic does not ignore other actions, only prioritizes them differently, for completeness, but for our analysis we ignored all non-preferred actions.

When combining preferred operators and FBS, we restrict our attention to actions that appear in the relaxed plan, both in the forward search and in the backward analysis. That is, when we expand a state either in the forward expansion (Algorithm 1, line 13), or in the backward expansion (line 33), we consider only actions that appear in the relaxed plan computed through the FF heuristic.

We experiment with a number of domains from the International Planning Competition (IPC). Our inefficient implementation did not allow us to solve many such domains, and we hence restrict our analysis only to 5 domains, where our forward search with preferred operators managed to solve larger instances. For these domains we experiment with the first 20 problems, as the larger instances could not be solved by our

	Time (secs)					Actions					Coverage				
	PO	S	FB	Fwd	FBPO	PO	S	FB	Fwd	FBPO	PO	S	FB	Fwd	FBPO
	IPC														
elevators	9.46	50.93	21.55	21.01	29.73	72.9	52.4	61.5	56.8	70.8	**20**	17	**20**	5	19
openstacks	1.74	4.59	4.05	3.75	39.76	81.4	81.6	81.2	53.6	71.2	**20**	**20**	**20**	10	18
parcprinter	0.37	1.75	1.3	28.12	0.34	36.1	44.9	36.1	27.1	38.3	12	**20**	**20**	11	17
pegsol	4.93	17.89	10.09	20.04	11.86	19.5	19.4	18.6	19.4	19.5	**19**	**19**	17	**19**	**19**
scanalyzer	23.63	36.05	43.86	39.16	35.4	23.7	14.2	17	25.6	28.7	**19**	10	11	13	18
	CoDMAP														
depot	52.7	3.31	1.81	86.39	35.41	32	21	22	34	55.5	5	3	3	2	**11**
driverlog	12.8	1.07	18.26	2.48	4.76	26.4	16.8	17.4	20.9	24.1	**16**	13	14	15	13
elevators	10.33	64.84	22.31	41.52	22.94	72.7	53.3	62.3	59.7	75	**20**	18	**20**	3	**20**
logistics	0.36	1.6	0.59	X	0.95	52.3	51.9	53.2	0	60.9	**20**	**20**	**20**	8	**20**
MALogistics	0.49	2.15	0.55	16.91	0.3	71.4	67.3	66.5	75.7	81.2	**20**	19	18	15	**20**
rovers	7.19	32.5	24	X	18.99	64	33.8	41.6	X	60.1	**20**	5	8	0	18
satellites	15.94	70.25	77.59	58.13	19.12	47.7	39.1	37.2	58.5	33.9	**17**	13	10	3	8
taxi	0.03	8.54	0.14	2.52	0.03	21.9	21.4	21.2	21.8	23.2	**20**	**20**	19	**20**	**20**
zenotravel	31.6	42.64	29.59	15.78	11.51	47.7	26.6	34	33.7	29.8	**19**	14	16	15	14
Sum											247	211	216	139	235

Table 1: Comparing heuristic forward search (S), forward only (Fwd), Forward-Backward (FB), and their preferred operators versions (PO, FBPO), over classical planning domains from IPC, and over unified multi-agent domains from CoDMAP.

forward search implementation.

In addition, we experiment with domains from the multi-agent collaborative CoDMAP competition. We believe that these domains contain a more factored search space, which can be exploited by our forward backward approach. We hence took multi-agent domains from the CoDMAP benchmark set, and unified them into single-agent domains.

Table 1 compares the performance of the different algorithms. Looking at coverage, we can see that almost always heuristic forward search with preferred operators achieves the best coverage. The second best method is the FBPO variant, which uses a forward-backward approach with preferred operators. In one domain, depot, which appears to be the most difficult domain in our benchmark set, forward-backward with preferred operators achieved much better coverage than all other approaches.

Of the methods that do not use preferred operators, the forward-backward approach achieves a slightly higher coverage than heuristic forward search. This is an encouraging result, showing that the forward-backward approach has the potential to improve upon regular forward search.

The forward-only approach, considering only actions that have some precondition that was generated by the previous action, fails completely on 6 of the 14 domains that we checked, but solves many instances in the other 8. This perhaps shows that these domains are, in a way, easier to solve. Still, even in domains where many instances were solved, forward-only search, although drastically limiting the set of considered actions is not necessarily faster than other methods. It may also generate longer plans, as in MALogistics.

Looking at plan length (number of actions), we can see that heuristic search with preferred operators often does not find the best plan. For example, in elevators (both versions), the PO variant produces much longer plans. The FBPO variant also produces longer plans in some cases, such as MALogistics. Comparing only heuristic search and forward-backward, the results are inconclusive — in parcprinter FB finds shorter plans, while in logistics heuristic search is better.

5 Conclusion

We suggested a new search paradigm, which we call forward-backward search, allowing us to limit the number of actions that are considered at each expansion, while maintaining the space of plans that can be computed.

We define forward actions — actions that require a precondition supplied by the last action. We show that for completeness one must also consider actions that supply a precondition for a forward action. We search for such actions using what we call backward reasoning.

We provide completeness proofs for our methods, and a negative example for an intuitive, yet incomplete variant, where we search backwards only for missing preconditions.

We provide an experimental evaluation of our approach, comparing our methods to standard heuristic forward search, showing that our methods produce slightly better coverage, and in some cases shorter plans.

One obvious future direction is to implement our methods into an existing planner such as FF or FD. This would allow us to test our approach in a competitive highly optimized planner, and see whether they improve upon heuristic search.

Most closely related to our work are various action pruning techniques, and in particular, strong stubborn sets [Wehrle and Helmert, 2014]. Strong stubborn sets is an optimality preserving method for action pruning. Given a set s, one computes a set $\mathcal{A}_s$ of actions that contain (i) all actions that can achieve one (arbitary) sub-goal that does not hold at s, (ii) for all actions $a \in \mathcal{A}_s$ not applicable in s, $calA_s$ contains all actions that can achieve one preconditions of these actions (iii) for all actions $a \in \mathcal{A}_s$ applicable in s, $\mathcal{A}_s$ contains all actions a' that interfere with a whose preconditions do not contradict those of a. Strong stubborn sets do not have the focused element driving the forward part of our search, they strongly resemble the type of backwards computation that determines what additional actions to consider. In particular, it is similar to the backwards computation used in FBS3, where a stack is not maintained. As we indicated earlier, we believe that condition (iii) is required for completeness of FBS1.

Acknowledgments: This work was supported by ISF Grant 933/13, by the Helmsley Charitable Trust through the Agricultural, Biological and Cognitive Robotics Center of Ben-Gurion University of the Negev, and by the Lynn and William Frankel Center for Computer Science.

References

[Hoffmann, 2001] J. Hoffmann. FF: The fast-forward planning system. *AI magazine*, 22(3):57, 2001.

[Karpas and Domshlak, 2012] Erez Karpas and Carmel Domshlak. Optimal search with inadmissible heuristics. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*, 2012.

[Newell and Simon, 1961] Allen Newell and Herbert Alexander Simon. Gps, a program that simulates human thought. Technical report, DTIC Document, 1961.

[Pommerening and Helmert, 2015] Florian Pommerening and Malte Helmert. A normal form for classical planning tasks. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015.*, pages 188–192, 2015.

[Tate, 1977] Austin Tate. Generating project networks. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence. Cambridge, MA, August 1977*, pages 888–893, 1977.

[Wehrle and Helmert, 2014] Martin Wehrle and Malte Helmert. Efficient stubborn sets: Generalized algorithms and selection strategies. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*, 2014.