Simple and Fast Strong Cyclic Planning for Fully-Observable Nondeterministic Planning Problems

Jicheng Fu
Computer Science Department
University of Central Oklahoma
jfu@uco.edu

Vincent Ng, Farokh B. Bastani, and **I-Ling Yen**
Computer Science Department
University of Texas at Dallas
vince@hlt.utdallas.edu,{bastani, ilyen}@utdallas.edu

Abstract

We address a difficult, yet under-investigated class of planning problems: fully-observable nondeterministic (FOND) planning problems with strong cyclic solutions. The difficulty of these strong cyclic FOND planning problems stems from the large size of the state space. Hence, to achieve efficient planning, a planner has to cope with the explosion in the size of the state space by planning along the directions that allow the goal to be reached quickly. A major challenge is: how would one know which states and search directions are relevant before the search for a solution has even begun? We first describe an NDP-motivated strong cyclic algorithm that, without addressing the above challenge, can already outperform state-of-the-art FOND planners, and then extend this NDP-motivated planner with a novel heuristic that addresses the challenge.

1 Introduction

Planning is the problem of generating a plan of actions that lead us from an initial state to a goal state. While the problem of deterministic planning has been extensively investigated, the same is not true for planning in nondeterministic domains [Levesque, 2005; Kissmann & Edelkamp, 2009], where the outcomes of an action are uncertain. These nondeterministic actions contribute to the exponential growth in the search space and hence the difficulty of nondeterministic planning. This difficulty remains even in fully-observable nondeterministic (FOND) planning, where the states of the world are fully observable [Kuter *et al*., 2008].

We address an important yet under-investigated class of planning problems: FOND planning problems with strong cyclic solutions. The notion of strong cyclic solutions was first introduced by Cimatti *et al.* [2003], who, for the purpose of addressing nondeterministic planning problems, present a three-way categorization of planning solutions: *weak planning* solutions have a chance to achieve the goal; *strong planning* solutions are guaranteed to achieve the goal; and *strong cyclic planning* solutions have a possibility to terminate and if they terminate, then they are guaranteed to achieve the goal.

Strong cyclic planning problems have been reported hard to solve [Levesque, 2005]. In fact, we can get a better sense of how difficult these problems are by observing the performance of state-of-the-art planners on the FOND benchmark problems from the International Planning Competition 2008 (IPC2008): none of the participating planners can solve more than ten of the 30 problems in the Blocksworld domain [Bryce & Buffet, 2008b].

As mentioned above, a major challenge in strong cyclic planning involves dealing with a very large state space. To circumvent the state space explosion problem, a planner may plan along the directions that allow the goal to be reached quickly in a limited state space involving only the relevant states. However, an interesting but challenging problem arises: how would one know which states and search directions are relevant before the search for a solution has even begun [Magnusson & Doherty, 2008]?

To address this question, we need a basic strong cyclic planner to start with. Consequently, we first implemented an idea motivated by NDP [Kuter *et al*., 2008]: we iteratively expand a graph that only includes states reachable from the initial state until one *solves* (i.e., establishes a path to the goal from) each of its non-goal leaf states. We show for the first time that an NDP-motivated algorithm outperforms state-of-the-art planners such as MBP [Cimatti *et al*., 2003] and Gamer [Kissmann & Edelkamp, 2009].[1] In particular, this "Basic" algorithm can solve *all* of the IPC2008 FOND problems in the Blocksworld domain. In other words, we are proposing extensions to a state-of-the-art planner.

Next, as it is common to encounter states that have been previously solved in the course of strong cyclic planning, we enhance the Basic algorithm with the capability to avoid re-exploring a solved state. Note that the idea of "state reuse" is not new: Mattmüller *et al.* [2010] have already implemented this idea in their solve-labeling procedure. What is new, however, is (1) we show that state reuse can be implemented with little overhead, in contrast to Mattmüller *et al.*'s planner, where state reuse is implemented with substantial overhead; and (2) the little overhead incurred in our implementation allows us to examine the benefits of state reuse in strong cyclic planning, specifically by determining

[1] The original NDP program was not set up to be easily run by other people, so we cannot verify whether the original NDP implementation also outperforms MBP and Gamer.

the extent to which the efficiency of the Basic algorithm is improved via state reuse, in contrast to Mattmüller *et al.*, who did not evaluate the benefits of state reuse. Our results suggest that state reuse can improve our state-of-the-art Basic algorithm. In contrast, even with state reuse, Mattmüller *et al.*'s planner cannot solve all the Blocksworld problems.

Finally, we address the aforementioned challenge by proposing a novel heuristic, *goal alternative*. This heuristic aims to improve planning efficiency by heuristically identifying an alternative goal that is typically closer to a state to be solved than the overall planning goal and hence can be reached more quickly. Incorporating goal alternative into our planner brings it to the next level of performance.

The rest of the paper is organized as follows. Section 2 defines nondeterministic planning. In Section 3, we present our NDP-motivated Basic algorithm for generating strong cyclic plans. Section 4 introduces two extensions to the Basic algorithm. We present evaluation results in Section 5, discuss related work in Section 6, and conclude in Section 7.

2 Nondeterministic Planning

We introduce the definitions and notation in nondeterministic planning that we will rely on in the rest of this paper.

Definition 1. A nondeterministic planning domain is a 4-tuple $\Sigma = (P, S, A, \gamma)$, where P is a finite set of propositions; $S \subseteq 2^P$ is a finite set of states in the system; A is a finite set of actions; and $\gamma : S \times A \rightarrow 2^S$ is the state-transition function.

Definition 2. A planning problem is a triple $\langle s_0, g, \Sigma \rangle$, where s_0 is the initial state, g is the goal condition, and Σ is the planning domain.

Definition 3. Given a planning problem $\langle s_0, g, \Sigma \rangle$ with $\Sigma = (P, S, A, \gamma)$, a policy is a function $\pi : S_\pi \rightarrow A$, where $S_\pi \subseteq S$ is the set of states to which an action has been assigned. In other words, $\forall s \in S_\pi : \exists a \in A$ such that $(s, a) \in \pi$. Given a state $s \in S_\pi$, the applicable action in the solution is $\pi(s)$. Therefore, π defines how to act under a specific state. We use $S_\pi(s)$ to denote the set of states reachable from s using π.

Definition 4 (taken from the rules of IPC2008 [Bryce & Buffet, 2008a]**).** A policy π is *closed* with respect to s iff $S_\pi(s) \subseteq S_\pi$. π is *proper* with respect to s iff the goal state can be reached using π from all $s' \in S_\pi(s)$. π is a *valid* solution to the non-deterministic problem iff π is closed and proper with respect to the initial state s_0.

3 Basic Strong Cyclic Algorithm

In this section, we describe our NDP-motivated Basic strong cyclic algorithm and illustrate how it works via an example.

3.1 Algorithm

Our Basic strong cyclic algorithm takes as input a planning problem, $\langle s_0, g, \Sigma \rangle$, and outputs a policy π that is a valid solution to the given problem. It starts with an empty π and iteratively augments it with state-action pairs.

Before we describe the algorithm, it helps to realize that π defines (and hence can be equivalently represented as) a digraph $G_\pi = \{V_\pi, E_\pi\}$, where $V_\pi = S_\pi \cup \{\gamma(s, \pi(s)) \mid s \in S_\pi\}$ is the set of nodes in G_π and $E_\pi = \{(s, s') \mid s \in S_\pi \text{ and } s' \in \gamma(s, \pi(s))\}$ is the set of edges. Note that there are two types of nodes in V_π : leaves and non-leaves. S_π are the non-leaves, since according to Definition 3, an action has been assigned to each state in S_π; on the other hand, $\{\gamma(s, \pi(s)) \mid s \in S_\pi\}$ subsumes the set of leaves of G_π. Assume that $G_\pi(s_0)$ represents a sub-graph of G_π rooted at s_0. Since the algorithm starts with an empty π, $G_\pi(s_0)$ initially contains only the initial state s_0. Informally, this Basic algorithm aims to augment π (or, equivalently, expand G_π) so that each node in $G_\pi(s_0)$ is *solved* (i.e., there are no non-goal leaf nodes).

Figure 1 outlines the Basic strong cyclic algorithm, which is similar in spirit to NDP [Kuter *et al.*, 2008], but otherwise differs in various implementation details. As mentioned before, the algorithm takes as input a planning problem $\langle s_0, g, \Sigma \rangle$ and initializes π to be an empty policy (line 1). L stores the set of non-goal leaf states in $G_\pi(s_0)$ (line 3), which initially contains only the initial state s_0. If L is empty (line 4), the algorithm terminates and returns a solution. Note that this solution, π, is *closed* and *proper* (i.e., there are no non-goal leaf states and all the non-goal states can reach the goal along some paths) and is therefore a valid solution to the given planning problem. If L is not empty, a non-goal leaf state s is arbitrarily selected from L (line 5), and a *SEARCH* process (line 6) is created to obtain a path from state s to the overall planning goal g, which represents a weak plan. According to Kuter *et al.* [2008], this weak plan can be converted into an equivalent policy π'. In fact, the conversion is straightforward, since the path returned by *SEARCH* is a sequence that alternates between states and actions. Hence, if a path is found, the state-action pairs can be directly obtained from the path and incorporated into π (line 11).

```
Procedure Basic_Strong_Cyclic ($\langle s_0, g, \Sigma \rangle$)
1.  $\pi \leftarrow \emptyset$
2.  loop
3.      $L \leftarrow$ {all non-goal leaf states in $G_\pi(s_0)$}
4.      if $EMPTY(L)$ then $RETURNSOL(\pi)$ ; endif
5.      arbitrarily choose $s \in L$;
6.      $\pi' = SEARCH(s, g, \Sigma)$;
7.      if $\pi'$ = FAILURE then
8.          if $s = s_0$ then return FAILURE; endif
9.          $BACKTRACK(s, \pi, G_\pi(s_0), \Sigma)$;
10.     else
11.         $\pi \leftarrow \pi \cup \{(s, a) \in \pi' \mid s \notin S_\pi(s_0)\}$ ;
12.     endif
13. endloop
```

Figure 1: Outline of the Basic Strong Cyclic Algorithm

However, if *SEARCH* fails to find a path from s to the goal, then s is a dead-end (line 7), and any paths leading to state s will be futile. In particular, if s is the initial state, then it is not possible to establish a path from the initial state to the goal state and hence there is no solution to the given planning problem (line 8). On the other hand, if s is not the initial state, we use the *BACKTRACK* procedure (line 9) to prune all the paths in $G_\pi(s_0)$ that lead to s. Formally, $\forall s' \in S_\pi(s_0)$, $a \in A$ such that $s \in \gamma(s', a)$, *BACKTRACK* will update: $\pi \leftarrow \pi - \{(s', a)\}$, $L \leftarrow L \cup \{s'\}$, and make action a inapplicable at s' in Σ.

Therefore, the non-goal leaf set L in line 3 is dynamic in the planning process. This Basic strong cyclic algorithm is sound and complete if the *SEARCH* procedure is sound and complete [Kuter *et al*., 2008]. In our implementation of this Basic algorithm, we use FF [Hoffmann & Nebel, 2001] to find a weak plan in *SEARCH*.

An implementation detail deserves mention. While we describe this Basic algorithm in terms of how G_π is expanded, G_π is only a conceptual structure: all we need to keep track of is π and L, not G_π.

3.2 An Illustrative Example

To better understand this Basic algorithm, we apply it to generate a strong cyclic plan for the following problem from the Beam domain [Cimatti *et al*., 2003], which is shown in Figure 2. Recall that in Beam, the goal is to have the agent move from the beginning (*down*$_0$) to the end (*up*$_n$) of the beam. There are three actions: Jump, Climb, and Moveback. The action Jump is nondeterministic: if the agent is at *up*$_{i-1}$, it may successfully jump to *up*$_i$ or fall to *down*$_i$. If it falls, it has to move all the way back to the leftmost down position (*down*$_0$) via the Moveback action, after which the Climb action will bring it to *up*$_0$. Here, Moveback and Climb are deterministic.

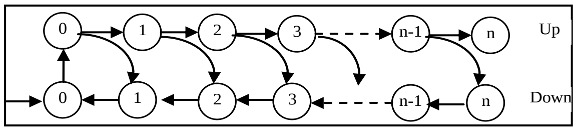

Figure 2: The Beam Problem

When applying the basic algorithm to this problem, L is initialized to {down$_0$}, since the initial state is the only non-goal leaf state (line 3). Next, the algorithm selects {down$_0$} from L (line 5), and uses *SEARCH* to find a weak plan *wp* from *down*$_0$ to the goal *up*$_n$ (line 6), where *wp* = Climb(*down*$_0$, *up*$_0$); Jump(*up*$_0$, *up*$_1$); Jump(*up*$_1$, *up*$_2$); …; Jump(*up*$_{n-1}$, *up*$_n$). The corresponding policy π' = {(*down*$_0$, Climb(*down*$_0$, *up*$_0$)), (*up*$_0$, Jump(*up*$_0$, *up*$_1$)), …, (*up*$_{n-1}$, Jump(*up*$_{n-1}$, *up*$_n$))} (line 6) is then merged with π (line 11).

Then, the algorithm proceeds to the next iteration. It begins by updating L (line 3). At this point, L = {*down*$_1$, *down*$_2$, …, *down*$_n$}. For each non-goal leaf state *down*$_i$ in L, the Basic algorithm will generate a weak plan in the form of Moveback(*down*$_i$, *down*$_{i-1}$); …; Moveback(*down*$_1$, *down*$_0$); Climb(*down*$_0$, *up*$_0$); Jump(*up*$_0$, *up*$_1$); …; Jump (*up*$_{n-1}$, *up*$_n$). Note that many states (e.g., *down*$_0$, *up*$_0$, etc.) are repeatedly explored.

4 Two Extensions to the Basic Algorithm

4.1 Goal-Alternative Heuristic

Recall that the Basic algorithm needs to solve every non-goal leaf node by establishing a path from the node to the goal. For many planning problems, the goal can be far away from a non-goal leaf node. As a result, computing the heuristic distance between the two nodes needed for heuristic search could be costly, thus causing planning to be inefficient. Our first extension to the Basic algorithm, the goal alternative heuristic, aims to improve planning efficiency by identifying an alternative goal that is closer to a non-goal leaf node than the overall goal, so that a path can be established between the leaf node and this alternative goal more quickly.

Before describing the heuristic, let us motivate it using an example. Recall from the beam example that the states *down*$_1$ and *up*$_1$ are both generated by the nondeterministic action Jump(*up*$_0$, *up*$_1$). One effect of this action is the state *up*$_1$, which is included in the weak plan produced by *SEARCH* and is called the *intended effect* [Kuter, 2004]. The other effect, *down*$_1$, is not included in the weak plan and is called a *failed effect*. In the Basic algorithm, every failed effect of a nondeterministic action is inserted into L (and therefore needs to be solved at some point). We call the heuristic of setting the intended effect as the goal when searching for a weak plan for a failed effect *goal alternative*.

More formally, when given the planning problem $\langle s, g, \Sigma\rangle$ for some $s \in L$, instead of searching for g directly, a *SEARCH* algorithm that exploits the goal alternative heuristic will first try to search for $\hat{s}$, the intended effect of the nondeterministic action that produces s. If a weak plan (path) is found, then the *SEARCH* is done. Otherwise, the *SEARCH* will be restarted to search for the original goal g. A special case deserves mention, though. During the first iteration of the algorithm when s_0 (the initial state) is the only non-goal leaf node, s_0 does not have an intended effect. Consequently, *SEARCH* will directly search for the goal g when given the planning problem $\langle s_0, g, \Sigma\rangle$.

Correctness. It is easy to determine the correctness of goal alternative for *SEARCH*(s, g, Σ). By definition, an intended effect $\hat{s}$ must be included in some path *wp* to the goal g, while the failed effect s is ignored in path *wp*. Since we have already found a path from $\hat{s}$ to g, if we can find a path from s to $\hat{s}$, then the path from s to $\hat{s}$ can be the solution to $\langle s, g, \Sigma\rangle$. Thus, much effort is saved by avoiding repeating the search from $\hat{s}$ to g. However, if no path can be found from s to $\hat{s}$, then we will resort to establishing a path from s to g.

Rationale and completeness. An intended effect and its corresponding failed effects are generated from the *same* nondeterministic action. Heuristically, the distance between a failed effect and the intended effect is shorter than that between a failed effect and the overall planning goal.

Since goal alternative is a heuristic, there are situations where the true distance to the intended effect is longer than that to the overall planning goal or no paths exist at all to the intended effect. In practice, however, the goal alternative heuristic works well on most of the benchmark problems.

4.2 State Reuse

Our second extension aims to enhance the Basic algorithm with the capability to avoid re-exploring a solved state. Specifically, during the *SEARCH* for a weak plan (line 6), if we encounter a solved state, the search stops. Since this extension can be incorporated easily into the Basic algorithm, there is almost no overhead in employing state reuse.

To see how state reuse helps improve planning efficiency, consider again our running example. Assume without loss of generality that $L = \{down_1, down_2, \ldots, down_n\}$ and that the states are selected from L in the same order as they are listed (i.e., $down_i$ is selected before $down_{i+1}$). If the Basic algorithm also adopts the goal alternative heuristic, it will initiate $SEARCH(down_1, up_1, \Sigma)$; otherwise, it will initiate $SEARCH(down_1, up_n, \Sigma)$. Note that both searches will result in the same weak plan that consists of the single action Moveback($down_1$, $down_0$), since $down_0$ is a solved state and hence the search stops. By the same token, for $down_i$ where $i > 1$, $SEARCH$ will generate the weak plan consisting of only one action, Moveback($down_i$, $down_{i-1}$). Note that the effectiveness of the state reuse extension depends slightly on the order in which the states are selected from L. For instance, if $down_{i+1}$ is selected before $down_i$, then $down_i$ cannot be reused for $down_{i+1}$, even though $down_i$ can reuse the states generated by the search for $down_{i+1}$. In our implementation of the state reuse extension, L is a queue, which allows the states to be chosen in the order in which they are inserted. Performance-wise, however, we found that this first-in-first-out ordering of the states offers only marginal benefits over a random ordering of the states.

5 Evaluation

In this section, we evaluate our planner, FIP (Fast Incremental Planner), which consists of the Basic algorithm augmented with the state reuse extension and the goal alternative heuristic. As baseline systems, we employ two state-of-the-art planners for strong cyclic FOND planning problems: MBP and Gamer (the winner of the FOND track at IPC2008). To gauge the usefulness of our two extensions, we also report the performance of the Basic algorithm. All problem instances belong to the benchmark domains of the IPC2008 FOND track (i.e., Blocksworld [bw], Forest, Faults, and First-responders [f-r]). Our experiments were conducted on a PC with an Intel Pentium-4 3GHz processor and 1 GB of memory. The operating system is Linux. We set the cutoff time to 1,200 seconds to prevent a planner from running indefinitely.

5.1 Results

Table 1 shows the problem coverage (i.e., the number of problems for which a strong cyclic plan was found) of four planners: MBP, Gamer, the Basic algorithm, and FIP.[2] As we can see, FIP and Basic have the same problem coverage, which is substantially higher than that of the two baselines. FIP and Basic also solve more problems than LAO* with PDB [Mattmüller *et al.*, 2010], which can solve only 94 problems. For the notoriously difficult blocksworld benchmark problems [Bryce & Buffet, 2008b; Mattmüller *et al.*, 2010], no existing planners can solve more than ten problems in this domain. In contrast, FIP and Basic can solve all of the 30 problems efficiently. As Basic is a state-of-the-art planner, any improvements of FIP over Basic shown below can be interpreted as improvements over the state of the art.

Domain	Gamer	MBP	Basic	FIP
blocksworld (30)	10	1	30	30
faults (55)	38	16	55	55
first-responders (100)	21	11	75	75
forest(90)	7	0	7	7
Total (275)	76	28	167	167

Table 1: Problem Coverage

Table 2 includes a subset of the benchmark problems that are hard to solve, i.e., at least one planner took 15 seconds to search for a solution or at least one planner timed out on that particular problem. To ensure a fair comparison of the planners, the preprocessing time (e.g., reachability analysis) and the plan output time are excluded. In other words, we only compared the pure search time needed to find a solution. The left half of Table 2 compares the performance of the four planners. Each planner is evaluated in terms of (1) the search time (expressed in seconds) needed to solve a problem (see the columns marked '*t*'); and (2) the plan size (expressed in terms of the number of state-action pairs in the solution policy; see the columns marked '*s*'). Note that "---" indicates that the corresponding planner did not return a solution within the time limit. In terms of planning time, FIP is on average 3844 times faster than Gamer. While Basic performs better than MBP and Gamer, FIP is 8.2 times faster than Basic on average. Importantly, as the complexity of the benchmark problems increases, FIP can be more than 100 times faster than Basic. In terms of solution size, FIP's plans are 2.8 times smaller than Gamer's and 3.4 times smaller than Basic's on average. Overall, FIP's results on these benchmark problems are very impressive.

5.2 Contributions Made by the Two Extensions

To measure the contributions of our two extensions to FIP's performance, we conducted ablation experiments, where we ran two versions of FIP. Each version contains exactly one of the two extensions to the Basic algorithm. Specifically, FIP-GA-only extends Basic with only goal alternative, whereas FIP-SR-only extends Basic with only state reuse. Results of these two ablation experiments are shown in the two rightmost columns of Table 2.

First, we determine the extent to which state reuse reduces planning time and solution size. Comparing Basic and FIP-SR-only, we see that on average, the addition of state reuse allows FIP-SR-only to run 1.4 times faster than Basic, but it does not allow FIP-SR-only to generate significantly smaller plans than those of Basic. Comparing FIP and FIP-GA-only, we see that the absence of state reuse causes FIP-GA-only to run 1.2 times slower than FIP, but again FIP-GA-only's solutions have almost the same size as those of FIP. Hence, these results suggest that while state reuse plays an important role in improving planning efficiency, it does not contribute significantly to generating compact solutions.

Second, we determine the extent to which goal alternative reduces planning time and solution size. Comparing Basic and FIP-GA-only, we see that the addition of goal alternative allows FIP-GA-only to run 7.5 times faster than Basic and

[2] We use the original implementations of MBP and Gamer to obtain the results for these two planners. We downloaded MBP from http://mbp.fbk.eu/download/mbp_1.01.tar.gz, and obtained Gamer directly from its authors.

generate plans that are 3.4 times smaller than those of Basic. Comparing FIP and FIP-SR-only, we can see that in the absence of goal alternative, FIP-SR-only runs 8 times slower than FIP and generates solutions that are 3.4 times larger than FIP's. Therefore, these results substantiate the claim that goal alternative contributes to both faster planning time and smaller solution size.

Next, we examine how well goal alternative performs on different domains. Blocksworld and Faults are the domains where goal alternative demonstrates substantial benefits. On the other hand, the role played by goal alternative in First-responders is comparatively less significant. To see the reason, note that most of the nondeterministic actions in this domain can generate one of two possible effects: (1) $\gamma(s, a) = s'$ and $s' \neq s$, i.e., moving to another state s'; or (2) $\gamma(s, a) = s$, i.e., remaining in the current state. However, for the failed effect of remaining in the current state (i.e., case 2), there is no need to initiate a search for it because the current state s has already been associated with the nondeterministic action a, i.e., $(s, a) \in \pi$. As a result, the first weak plan generated, which is a plan from s_0 to g, is almost the final solution as most of the failed effects generated by the nondeterministic actions are already solved. Finally, using goal alternative is slightly harmful for Forest, where planning time was even a little faster when goal alternative was not used (see FIP-SR-only). However, the time difference between using and not using goal alternative is fairly minor. Hence, even when goal alternative provides the wrong direction to the search, the overhead of failing to find a path to the intended effect and resorting to the original planning goal is low.

On average, FIP-GA-only runs more than 5 times faster and creates plans that are 3.4 times smaller than FIP-SR-only. These results provide suggestive evidence that goal alternative plays a more crucial role than state reuse in improving planning efficiency and reducing plan size.

Problem	Gamer		MBP[3]	Basic		FIP		FIP-SR-only		FIP-GA-only	
	t	*s*	*t*	*t*	*s*	*t*	*s*	*t*	*s*	*t*	*s*
bw-1	38.748	10	*3556.517*	0.011	12	0.007	8	0.010	12	0.007	8
bw-2	29.215	16	---	0.008	10	0.006	7	0.008	10	0.005	7
bw-3	32.742	21	---	0.014	10	0.008	10	0.008	10	0.010	10
bw-4	37.186	26	---	0.016	16	0.011	14	0.012	16	0.013	14
bw-5	37.506	13	---	0.020	12	0.010	12	0.014	12	0.013	12
bw-6	30.847	19	---	0.013	10	0.009	10	0.010	10	0.011	10
bw-7	32.249	28	---	0.032	22	0.010	17	0.016	22	0.015	17
bw-8	37.944	19	---	0.019	14	0.010	13	0.012	14	0.012	13
bw-9	28.632	10	---	0.017	15	0.009	12	0.009	12	0.010	15
bw-10	28.650	13	---	0.015	22	0.009	10	0.010	10	0.010	13
bw-12	---	---	---	3.507	225	0.285	45	0.725	131	1.216	138
bw-22	---	---	---	44.615	362	5.510	163	12.117	362	10.880	163
bw-24	---	---	---	180.247	915	9.204	263	25.490	646	12.453	263
bw-25	---	---	---	452.642	537	59.230	312	112.235	453	280.510	381
bw-27	---	---	---	31.683	280	2.904	61	11.738	247	4.850	61
bw-30	---	---	---	42.755	102	3.126	48	7.458	102	7.249	48
faults-4-3	0.251	18	64.290	0.006	37	0.003	14	0.006	37	0.003	14
faults-5-5	10.403	63	---	0.010	66	0.004	17	0.011	66	0.004	17
faults-6-6	81.695	125	---	0.020	130	0.005	20	0.021	130	0.005	20
faults-7-7	90.996	235	---	0.043	258	0.005	23	0.045	258	0.006	23
faults-8-8	1106.105	325	---	0.101	514	0.007	26	0.111	514	0.007	26
faults-9-9	830.272	511	---	0.217	848	0.007	29	0.246	848	0.008	29
faults-10-7	---	---	---	0.816	2140	0.007	32	0.960	2140	0.008	32
faults-10-10	---	---	---	0.859	2050	0.009	32	1.101	2050	0.009	32
f-r-1-8	6.839	10	59.327	0.003	10	0.002	10	0.003	10	0.003	10
f-r-2-3	0.142	12	63.388	0.003	11	0.003	11	0.003	11	0.003	11
f-r-4-2	0.118	8	---	0.003	6	0.003	6	0.003	6	0.003	6
f-r-6-2	1.016	7	---	0.003	7	0.003	7	0.003	7	0.003	7
forest-2-2	3.164	44	---	0.007	44	0.008	44	0.007	44	0.009	44
forest-2-5	0.661	56	---	0.007	56	0.008	56	0.007	56	0.009	56
forest-2-6	4.769	50	---	0.008	50	0.008	50	0.007	50	0.009	50
forest-2-7	8.122	44	---	0.007	44	0.008	44	0.007	44	0.009	44
forest-2-8	0.638	56	---	0.007	56	0.008	56	0.007	56	0.009	56
forest-2-9	0.607	42	---	0.006	42	0.007	42	0.006	42	0.008	42
forest-2-10	0.927	44	---	0.007	44	0.008	44	0.006	44	0.009	44

Table 2: Planning Time and Plan Size Comparison among Four Planners: Gamer, MBP, Basic, and FIP

[3] MBP outputs too much information to enable easy counting of the number of meaningful state-action pairs.

6 Related Work

MBP [Cimatti *et al.*, 2003] and Gamer [Kissmann & Edelkamp, 2009] are widely considered as the best known planners that are able to deal with strong cyclic planning. MBP formulates planning as model-checking. It is built upon Binary Decision Diagrams (BDDs), which allow a compact representation and can therefore explore a large state space. It starts with a universal policy $Uni\pi$ that contains all the possible state-action pairs, and iteratively eliminates from $Uni\pi$ the state-action pairs leading to states not covered by $Uni\pi$ and state-action pairs that are dead-ends. After the iterative process terminates, if the initial state is defined in the final policy, then a plan is found. The inherent weakness of MBP is that it has to deal with a very large state space and therefore its execution time grows exponentially with the problem size [Kuter & Nau, 2004]. Gamer is also based on BDDs. It transforms a nondeterministic planning problem into a two-player turn-taking game so that tools can be applied to infer a minimized state encoding, which results in smaller BDDs than MBP. As a result, Gamer is more efficient and has better scalability than MBP. However, both MBP and Gamer use uninformed symbolic regression search, which may negatively impact planning efficiency.

NDP [Kuter *et al.*, 2008] uses classical AI planners to solve nondeterministic problems without requiring internal modifications to the classical planners. It calls a classical planner to find a weak plan for each non-goal leaf node to the goal. It does not maintain the intermediate search results and, hence, unlike FIP, it cannot avoid exploring the same states repeatedly, resulting in wasted efforts.

Mattmüller *et al.* [2010] use LAO* [Hansen & Zilberstein, 2001], an informed explicit state search algorithm on an AND/OR graph, in combination with pattern database heuristics [Edelkamp, 2001] to generate strong cyclic solutions. LAO* explicitly constructs a graph during planning, but maintaining this graph incurs much overhead because determining whether a state is solved and checking whether a solution is found are very expensive operations. More specifically, the solve labeling procedure in Mattmüller *et al.* [2010] is a nested fixed point computation, in which an outer loop and two inner nested loops compute a set of nodes backwardly reachable from the goal. As a result, this method does not scale very well.

7 Conclusions

We have examined an important class of nondeterministic planning problems: FOND problems with strong cyclic solutions. Our work makes two contributions to planning research. First, we showed how the state reuse extension can help avoid re-exploring a solved state with little overhead and evaluated for the first time the benefits of state reuse for strong cyclic FOND planning problems. Second, we propose a novel heuristic, goal alternative, which aims to improve planning efficiency by heuristically identifying an alternative goal that is typically closer to the current state and can therefore be expected to be reached more quickly. Experimental results on the benchmark domains of the IPC2008 FOND track show that our planner, FIP, significantly outperforms state-of-the-art planners in terms of problem coverage, CPU time, and solution size. These results suggest that our two extensions to the Basic algorithm are both very effective.

References

[Bryce & Buffet, 2008a] D. Bryce and O. Buffet. 6th International Planning Competition: Uncertainty Part. In *Proceedings of IPC*, 2008.

[Bryce & Buffet, 2008b] D. Bryce and O. Buffet. International Planning Competition Uncertainty Part: Benchmarks and Results. In *Proceedings of IPC*, 2008.

[Cimatti *et al.*, 2003] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.

[Edelkamp, 2001] S. Edelkamp. Planning with pattern databases. In *Proceedings of ECP*, pages 13–24, 2001.

[Hansen & Zilberstein, 2001] E. A. Hansen and S. Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence* 129(1–2):35–62, 2001.

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

[Kissmann & Edelkamp, 2009] P. Kissmann and S. Edelkamp. Solving fully-observable non-deterministic planning problems via translation into a general game. In *Proceedings of KI*, Vol. 5803 of LNCS, pages 1–8, Springer, 2009.

[Kuter, 2004] U. Kuter. Pushing the limits of AI planning. In *Proceedings of ICAPS Doctoral Consortium*, 2004.

[Kuter & Nau, 2004] U. Kuter and D. Nau. Forward-chaining planning in nondeterministic domains. In *Proceedings of AAAI*, pages 513–518, 2004.

[Kuter *et al.*, 2008] U. Kuter, D. Nau, E. Reisner, and R. P. Goldman. Using classical planners to solve nondeterministic planning problems. In *Proceedings of ICAPS*, 2008.

[Levesque, 2005] H. Levesque. Planning with loops. In *Proceedings of IJCAI*, 509–515, 2005.

[Magnusson & Doherty, 2008] M. Magnusson and P. Doherty. Deductive planning with inductive loops. In *Proceedings of KR*, pages 528–534, 2008.

[Mattmüller *et al.*, 2010] R. Mattmüller, M. Ortlieb, M. Helmert, and P. Bercher. Pattern database heuristics for fully observable nondeterministic planning. In *Proceedings of ICAPS*, pages 105–112, 2010.