FIP: A Fast Planning-Graph-Based Iterative Planner

Jicheng Fu, Farokh Bastani, Vincent Ng, I-Ling Yen, Yansheng Zhang
Department of Computer Science
University of Texas at Dallas
{jxf024000, bastani}@utdallas.edu, vince@hlt.utdallas.edu, {ilyen, yxz037000}@utdallas.edu

Abstract

We present a fast iterative planner (*FIP*) *that aims to handle planning problems involving nondeterministic actions. In contrast to existing iterative planners, FIP is built upon Graphplan's intrinsic features, thus, enabling Graphplan variants, including SGP and FF, to be enhanced with the capability of iterative planning. In addition, FIP is able to produce program-like plans with conditional and loop constructs, and achieves efficient planning via novel algorithms for manipulating planning graphs. Experimental results on several nondeterministic planning problems show that FIP is more efficient than the well-known planner, MBP, especially as the size of the problems increases.*

1. Introduction

Planning refers to the problem of generating a plan of actions that leads us from an initial state to a goal state. Much of the work on this problem has focused on *sequential planning*, in which the generated plan consists of a *sequence* of actions. Implicit in sequential planning is the assumption that each action produces a *deterministic* outcome. Consequently, sequential planners cannot be applied to problems in *nondeterministic* domains where the outcome of an action can be *probabilistic*. While successful Graphplan variants such as SGP [14] and Contingent FF [7] can address a special kind of nondeterministic planning problems in which uncertainty exists in the initial conditions, they are not designed to tackle iterative planning.

To handle general nondeterministic planning problems (i.e., problems in which actions may produce multiple outcomes), a number of planners have recently been developed. Common to these "nondeterministic planners" is their ability to work with nondeterministic actions. Nevertheless, these planners have their own weaknesses. For instance, as an MDP (Markov Decision Process)-based planner, GPT [2] cannot distinguish between acyclic and cyclic solutions [3]. MBP [3], which is widely considered as the best-known AI planner, requires CPU time that may grow exponentially with problem size [11]. Scalability problems have also been reported with Kplanner [12], a recently-proposed iterative planner.

Motivated in part by weaknesses in existing iterative planners, we propose a new iterative planning algorithm, FIP (Fast Iterative Planner), for efficiently addressing general iterative (cyclic) planning problems. One major feature of our approach is that it is *planning-graph-based*, relying only on Graphplan's existing features (e.g., level-off, shortest planning path) to generate program-like plans with conditional and iterative constructs. Consequently, it can be used to improve the expressiveness and power of existing Graphplan variants (e.g., SGP, FF), especially with respect to equipping these planners with iterative planning capability. Above all, FIP achieves fast planning via novel algorithms for manipulating planning graphs. In fact, experimental results on several benchmark planning problems demonstrate that FIP is more efficient than the well-known MBP planner and, more importantly, FIP scales much better as the size of the problems increases.

The rest of this paper is organized as follows. We first formally define a nondeterministic planning domain in FIP's context and then describe the loop construct generated by FIP. After that, we provide the details of FIP. Finally, we present experimental results and conclude by identifying some future research directions.

2. Planning Domain and Notation

2.1. Nondeterministic Planning Domain

To motivate nondeterminism in planning, and to explain the concepts in subsequent sections, we use the omelette problem [3] as our running example. The goal here is to break a certain number of good eggs into a bowl, assuming that an unlimited number of eggs are available. The *break* action is nondeterministic: it may result in a good or bad egg in the bowl, or it may fail to open the egg. If a bad egg is broken into the bowl, a remedial action *discard* is applied to get rid of the whole content of the bowl and the process restarts. Now, let us formally define a nondeterministic domain.

Definition 1 A nondeterministic planning domain is a four-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;
- $\gamma: S \times A \rightarrow 2^S$ is the state-transition function.

According to this definition, a nondeterministic action can generate multiple effects, each of which is a state in S. The effects that lead us to the goal (e.g., a good egg) are called *intended effect*s. Others are *failed effects* [10], representing action failures or undesired external events (e.g., an unbroken or bad egg). It should be noted that intended and failed effects are not static during the planning process. At different time steps, previously intended effects could become failed effects and vice versa. For example, the problem of breaking two good eggs and one bad egg in the bowl involves both "good" and "bad" as intended effects at different time steps. Our planner relies on the following theorem, which shows the relationship among the effects of an action.

Theorem 1. Consider an action o that may create a set of effects $E = \{e_1, e_2, \ldots, e_n\}$. Then,

$$\forall e_i \in E: \bigwedge_{k=1 \& k \neq i}^{n} \neg e_k \Rightarrow e_i \quad (1 \le i \le n)^1.$$

Proof outline: Informally, the theorem states that if it is already known that except for effect e_i, no other effects occur, then the generated effect must be e_i. It follows from the fact that an action must generate at least one effect.□

We use STRIPS [5] to formulate actions with multiple effects. Our method is not limited to STRIPS, but can be applied to PDDL [13] as well. In STRIPS, an action o is defined as a triple, $o = \langle pre(o), add(o), del(o) \rangle$, where $pre(o)$ are the preconditions of o; $add(o)$ are the add effects of o; and $del(o)$ are the delete effects of o. Similar to the formulation in [6], an action with multiple effects can be decomposed into multiple actions. Let o be an action with multiple possible effects $E = \{e_1, e_2, \ldots, e_n\}$. Then, $\forall e: e \in E$, an action φ is created such that:

(1) $pre(\varphi) = pre(o)$ and (2) $add(\varphi) = e$.

To differentiate these decomposed actions from the original deterministic actions, we define the set of decomposed actions as ***D-actions*** and the conventional deterministic actions as ***G-actions***. We further differentiate D-actions by defining ***S-actions*** and ***F-actions***, which are the D-actions associated with the intended and failed effects, respectively. The identification of S- and F-actions is problem dependent and will be determined during the planning phase.

Consider again the omelette example. The decomposed actions for the "*break*" action are: *D_Break_GOOD* (which generates the effect of good egg), *D_Break_BAD* (which produces the effect of bad egg), and *D_Break_UNBROKEN* (which produces the effect of unbroken). If the goal is to get good eggs, *D_Break_GOOD* will be the S-action and the rest will be the F-actions.

2.2. General Loop Construct for Nondeterministic Actions

Loops arise naturally in nondeterministic domains. To see the reason, consider an action o with multiple effects $E = \{e_1, e_2, \ldots, e_n\}$, with e_g being the intended effect. A loop needs to be formed if o keeps on generating failed effects, and will terminate only if e_g is produced.

Definition 2 gives a *general* loop construct generated from an action o. First, o is executed. If it produces an effect other than the intended effect e_g, then the loop starts. The loop body will judge what effect is produced by the action. For each failed effect, some remedial actions $r_{i,j}$ will be taken[2] (within an "if" construct). A remedial action may (or may not) lead the system to a state out of the loop. If such an out-of-the-loop transition occurs, then a goto s_x statement will appear in the loop.

Definition 2 General Loop Construct

```
o;
while(¬e_g){
        if(e_1) { r_{1,1}; …; r_{1,t1}; }
        else if(e_2) { r_{2,1}; …; r_{1,t2}; }
        … …
        else if(e_n) { r_{n,1}; …; r_{1,t2}; }
        o;
}
```

For example, to break the first good egg into the bowl, we will need the loop construct shown in Figure 1 (a). The remedial action for a bad egg is "*discard*". Note that the effect *unbroken* does not change the state of the world; so no remedial action is needed and the corresponding conditional branch can be removed. If we want to break a second egg into the bowl and the egg turns out to be bad, the remedial action *discard* resets the system to the initial state, which is out of the

[1] Throughout, only proof outlines will be presented due to space limitations.

[2] Here, we do not consider actions that generate fatal effects resulting in termination of the planning process with failure, since it is not the focus of this paper. However, such kinds of fatal effects are not difficult to handle: The planner only needs to take into account the fatal effects during the planning process and generate (fatal-effect, return) pairs.

loop. The loop for breaking the second good egg is in Figure 1 (b). Here, s_0 is the initial state corresponding to "the empty and clean bowl".

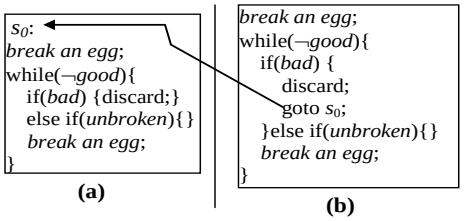

Figure 1: The loops for breaking two good eggs

If every "if" branch inside the loop ends by moving to some other state out of the loop, then the loop structure can be reduced to a conditional construct:

Definition 3 Conditional Construct

o;
if(e_1) { …, "transit out of the loop"}
else if(e_2) { …, "transit out of the loop"}
… …
else if(e_n) { …, "transit out of the loop"}

FIP can handle both loop and conditional constructs, making it full-fledged in dealing with nondeterminism.

3. The FIP Algorithm

Since FIP involves Graphplan, we first briefly introduce the basic concepts of Graphplan as shown in Figure 2. If no plan exists, the termination condition for Graphplan is when two adjacent proposition levels of the planning graph are identical, i.e., they contain the same set of propositions and have the same exclusivity relations. This is called **level off**. In conventional planning, when level off occurs, no appropriate plans are found and the algorithm terminates with failure [1].

Repeat
Graph expansion phase:
Add initial state to the initial proposition level;
Repeat
Apply actions enabled by the proposition level to derive the action level;
Effects generated by the action levels (including no-ops) form the next proposition level;
Until achieved a necessary condition for plan existence;
Solution extraction phase:
Use backward chaining search on the graph to identify a plan;
If found one, then return it;
Else the last *proposition-level* represents the set of new *initial conditions*.
Until a valid plan is obtained

Figure 2: The Graphplan algorithm

FIP includes two novel concepts. First, it makes use of the level-off property for loop construction. We force the graph planner to level off by holding back the S-actions (and remedial actions that may lead to the goal). This makes the planner explore all the F-actions and the corresponding remedial actions so that a loop construct with all the failed effects will be formed.

Before forcing level-off as discussed above (which is the basis of FIP), we need to identify the S-actions and the possibility of obtaining a plan via introducing the second novel concept of FIP: the generation of a weak plan [3] by classical Graphplan. The plan is weak because it only indicates a possible path to achieve the goal. All the D-actions in the weak plan are S-actions since they generate the intended effects. The weak plan also indicates that there is a possibility to achieve the goal. Most importantly, the weak plan represents the optimistic shortest backbone path to the goal because Graphplan always returns the shortest path, i.e., optimal number of time steps [1]. Since the search for a final plan is along the shortest path, a great performance gain is achieved. Moreover, due to the use of the weak plan as a backbone plan to guide the search, the second phase planning only requires a forward search. No backward chaining search is needed and, hence, this further reduces the cost significantly. In the following sub-sections, we present the details of the FIP algorithm, which operates in two phases.

3.1. Phase 1: Weak Plan Generation

In the first phase, the classical Graphplan is used to generate a weak plan. This plan cannot contain any F-actions according to the following theorem.

Theorem 2 The F-actions are bypassed by the S-actions and, thus, cannot be included in the weak plan.
Proof outline: The theorem is based on the fact that Graphplan always returns a plan with the shortest path [1]. F-actions that produce failed effects need some remedial actions to correct the effects and, therefore, cannot produce the shortest path. □

Consider again the omelette example. If the planning problem is to break two good eggs into a bowl, then the generated weak plan is as shown in Figure 3, including only S-actions. Note that it does not consider the situation where a bad egg is broken into the bowl.

D_Break_GOOD;
D_Break_GOOD;

Figure 3: The weak plan for the omelette problem

3.2. Phase 2: Complete Plan Generation

If no D-actions are included in the weak plan from the first phase, then the planning problem is a classical one that is solvable by classical Graphplan and the

solution from the first phase is the final plan. Otherwise, we proceed to the second phase, which contains the following steps:

(a) Expand the planning graph based on the weak plan;
(b) Locate the remedial actions for each failed effect;
(c) Assemble loops, if any, from the planning graph;
(d) Generate the final plan.

3.2.1. Expand the planning graph. FIP goes through the weak plan action by action and uses these actions to generate the new planning graph level by level. More specifically, when reaching the first S-action, say a_S^1, FIP performs a "sub-expansion", which holds back a_S^1 and expands the graph with all the F-actions corresponding to a_S^1. Then, at the new proposition level, all the applicable actions, except a_S^1, are used for further sub-expansion until level-off happens. According to Theorem 1, only after all the failed effects have been explored, can the intended effect be implied. Hence, starting from the level-off point, FIP expands the graph using a_S^1 (so that the planning graph reaches the same states as those in the weak plan). Next, FIP continues on the weak plan after a_S^1, action by action, and repeats the same steps as above till it exhausts the entire weak plan. Note that during the sub-expansion for some a_S^i, it is possible that some other D-actions and/or remedial actions may cause the planning graph to reach the goal. In this case, FIP marks the proposition state and stops expansions from it.

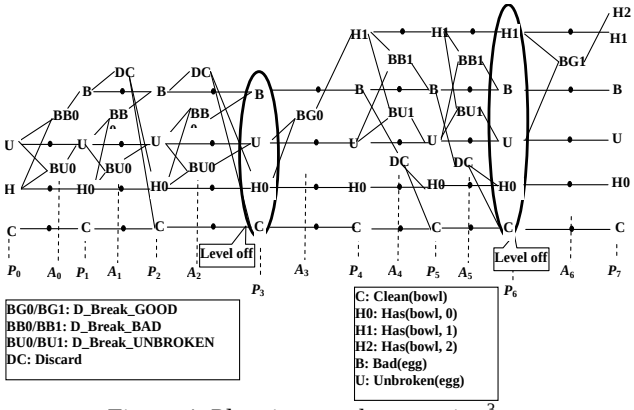

Figure 4: Planning graph expansion[3]

Figure 4 shows the planning graph expansion based on the weak plan in Figure 3. The first action in the weak plan is the S-action *D_Break_GOOD* (BG0). Thus it is held back and only *D_Break_BAD* (BB0) and *D_Break_UN-BROKEN* (BU0) are applied at action level A_0. Then, all the applicable actions are used to expand the graph until level-off happens at P_3. After level-off, BG0 is applied. The next action in the weak plan is again an S-action, BG1. So, BG1 is substituted by BB1 and BU1 at level A_4. After that, all the applicable actions are used to expand the graph till the next level-off. Following level-off, BG1 is applied and the expansion is done. The formal proof for the correctness of the processing steps is omitted here due to space limitations.

Based on the new plan, the loops can be identified. Specifically, the levels in the new planning graph constructed in each sub-expansion step form a loop. To construct the loop, additional processing is required to extract the failed effects and remedial actions as described in Section 3.2.2.

3.2.2. Locate the remedial actions. For each failed effect e_f, we need to determine the sequence of remedial actions. When level-off happens, the remedial actions, if any, have already shown up in the planning graph. FIP simply starts at the F-action for e_f, following the paths, and stops if the path leads to a state that enables the corresponding S-action or a state that satisfies $\neg e_f$ (i.e., the failed effect has been fixed). The search also stops if it reaches a state s that pre-exists, i.e., either s has appeared previously or appears in the backbone plan. In this case, we can simply use "goto s" to simplify the subsequent processing.

For example, to fix the failed effect of "*Bad*(egg)" generated by BB0, the search starts from P_1 in Figure 4. It leads to the remedial action "*Discard*" which generates an effect containing the precondition of the action "break an egg" (decomposed as BB0/BU0/BG0). Then the search stops. The search can go as far as the proposition level enabling the S-action. However, in most cases, it terminates earlier. Also, the remedial path for BB1 leads to a state transition that jumps out of the loop (this is how "goto s_0" appears in the code in Figure 1 (b)).

No remedial action is needed for the effect of "*unbroken*" because it does not change the state that enables BB0/BU0/BG0. For the benchmark problems that we have explored, the distance from the failed effects to the proposition level enabling the S-action is just 2-3 levels apart. Hence, the search for the remedial actions can be done very quickly.

During a search, how can we determine whether a state pre-exists and whether a pre-existing state is out of the loop boundary? To answer these questions, we use a heuristic method to locate the proposition level, where a state might appear for the first time in the planning graph. Specifically, we use the level-

[3] This is a simplified planning graph, which is arranged in levels alternating between proposition and action levels. BB0/BU0 and BB1/BU1 are applied when the bowl has 0 and 1 good eggs, respectively. We use *Has*(bowl, 1) representing the effect of the first good egg being broken into the bowl and *Has*(bowl, 2) for the second good egg.

membership (*lms*) function introduced in [8] to locate the possible proposition level of a state. Let p be a proposition that appears in proposition level i for the first time. Then, $lms(p) = i$. Given a state s, the function *LevelId* returns the id of the first proposition level which contains s:

$LevelId(s) = max\ (lms(p) \mid p \in s)$.

In FIP, *lms* is implemented using a hash table to help achieve efficient search. Not only can *LevelId* be used to quickly locate a state, it can also help determine whether a state is out of the loop very efficiently, by simply comparing the returned value with the loop boundary.

Let $R_a = a_1; a_2; \ldots; a_k$ be a sequence of remedial actions for fixing the failed effect generated by a D-action ϕ. If an action a_i in R_a is also a D-action but a_i and ϕ are decomposed from different nondeterministic actions, then a_i is an S-action that has bypassed its related D-actions. The S-action processing steps can be applied to it ***recursively*** as discussed in the previous section.

`

3.2.3. Assemble Loops. After the planning graph expansion is done and the remedial actions for each failed effect have been determined, we can generate loop(s) from the planning graph as follows. First, find the first action level containing all the F-actions. The F-actions represent the first operation "*o*" generating failed effects so that the loop starts as shown in Definition 2. Second, we use the remedial actions for each failed effect to generate an "if" branch. Third, the application of the S-action after level-off represents the action "*o*" in the loop, which eventually generates the intended effect in Definition 2.

For example, BB0 and BU0 are contained in action level A_0 in Figure 4. It represents the first "break an egg" in Figure 5, but generates only failed effects so that the loop starts. Then, the sequence "*D_Break_BAD*; *discard*" located from the Action level A_0 and A_1 is translated into "*if* (*bad*) {*discard*;}". Finally, *D_Break_GOOD* (BG0) is applied in Action A_3. It corresponds to the second "break an egg" in Figure 5.

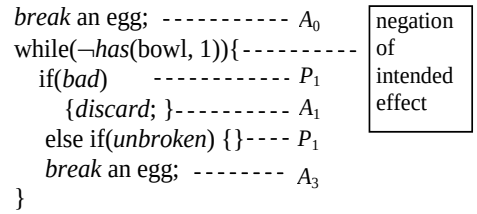

Figure 5: The loop construct for breaking the first good egg

Figure 5 summarizes how the first loop is assembled. The way to assemble the second loop is similar to the first one except that there is a state transition to s_0 (shown in Figure 1 (b)). This is because $LevelId(add(discard)) = 0 < 4$; while the proposition level P_4 is the level where the loop starts.

3.2.4. Generate the Final Plan. This step follows the standard planning procedure.

4. Experimental Study

We have implemented FIP based on an open source Graphplan system, namely, JPlan [4]. Besides performance evaluations, we also examine FIP's ability in generating loop and conditional constructs for nondeterministic actions. For the omelette problem, FIP generates a loop construct conforming to Definition 2 for each "break an egg" action; while in the beam domain, FIP generates a conditional construct conforming to Definition 3 for each nondeterministic action. We also use a robot tele-control problem to evaluate FIP's ability in solving practical problems.

4.1. The Omelette Problem

Recall that the goal of the omelette problem is to break a certain number of good eggs into a bowl. This problem may not be as easy as it appears: Kplanner [12] cannot generate a plan for 5 eggs or more within a reasonable time (> 3 weeks). In the rest of this subsection, we will compare the performance of FIP and MBP on this problem.

Following the common practice used in Contingent FF [7], some predicates are used to specify the domain. For instance, one of the predicates is *addone*(?*nb*, ?*na*), which specifies that increasing the number of ?*nb* by one gives the number ?*na*. As shown in Figure 6 (A), the CPU time needed by FIP increases slowly. On the other hand, the CPU time required by MBP increases very rapidly. This is because the use of logical expressions introduces more state variables to the planning domain, which in turn increases the number of system states dramatically.

As a result, we alter the problem specification, in which we remove all the predicates and instead use only constant objects to specify the nondeterministic action. The goal is to examine whether MBP would perform more efficiently under a condition that is more favorable for it. The results are shown in Figure 6 (B). As we can see, even though reducing the number of states in the system by removing the predicates helps improve MBP's efficiency, its CPU time still grows exponentially.

The reason that FIP handles this problem more efficiently than MBP is that the planning graph is not

the complete search state space graph. The first proposition level P_0 contains only initial conditions. All the proposition levels and actions levels are derived based on P_0 and contain the information only derivable from P_0. Hence, it is less likely to be affected by irrelevant states. With the mutual exclusion relations among propositions and actions, the search space can be further reduced.

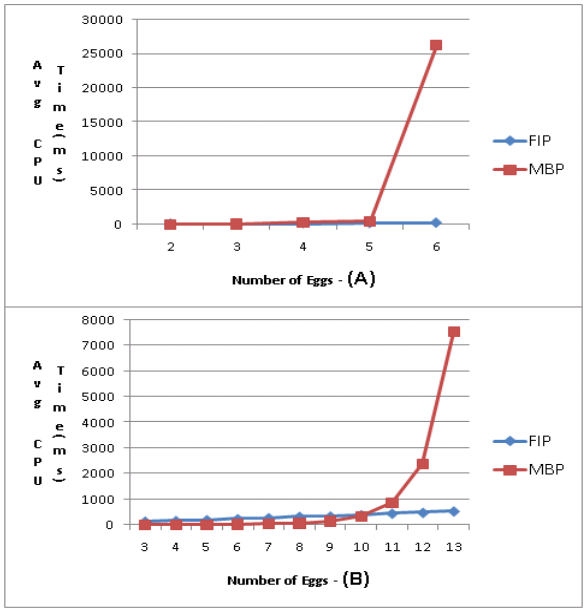

Figure 6: Results for the omelette problem

4.2. The Beam Problem

Next, we evaluate the beam problem, which is introduced in [3]. As shown in Figure 7, the goal is to have the agent walk from the beginning (up_0) to the end (up_n) of the beam. The action "walk" is nondeterministic since the agent may move from up_{i-1} to up_i (succeed) or fall to $down_i$ (fail). If it falls, it has to move all the way back to the left most down position. Then a "Go up" action will bring it to up_0.

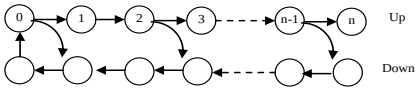

Figure 7: The beam problem

This problem is more challenging than the omelette problem in the sense that the remedial actions are a sequence of move-back actions to reset the system state when the walk from up_{i-1} to up_i fails, whereas the omelette problem just has a single remedial action, namely, "*discard*".

Figure 8 compares the efficiency of FIP and MBP on the beam problem. Once again, FIP outperforms MBP. The CPU time of MBP increases much more sharply than that of FIP. The reason why FIP outperforms MBP is similar to that for the previous experiment: FIP does not need to plan with a complete state space. As a result, it is less likely to be affected by the increasing number of irrelevant states due to increasing the number of positions.

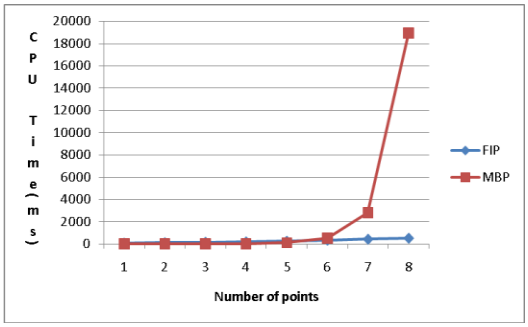

Figure 8: Result for the beam problem

Although beam is an iterative planning problem in general, FIP is smart enough to generate a conditional construct conforming to Definition 3 for each "walk" action. To see how FIP achieves this, note that "walk" has two possible effects: "up_i" or "$down_i$". For the failed effect "$down_i$", a remedial action "move back" results in a state that represents the immediately preceding down position. As mentioned in the previous section, FIP can easily detect that the state transition is out of the loop boundary. Since it has only one failed effect and has a state transition out of the loop, the loop construct can be reduced into a conditional construct. In addition, as mentioned before, FIP can detect that the system is transitioning to a previous proposition level. Hence, FIP can determine that the problem involves iterative planning because a backward state transition in a planning graph leads to an iterative process. As a result, FIP can distinguish between acyclic and cyclic solutions, a big improvement over GPT [2].

4.3. The Robot Tele-Control Problem

To evaluate FIP's ability in solving practical problems, we introduce a new nondeterministic planning problem, the robot tele-control problem, in which a robot simulator is implemented. Here, the robot is required to follow a predefined route to reach the destination. The robot can rotate around its center or move forward. If the robot needs to move from waypoint A to waypoint B, it first checks if it is already facing the given target point. If not, the robot will turn itself to point toward the target waypoint and then move to it. Due to mechanical limitations, the robot's

"rotate" action is nondeterministic: it may turn and face the target point successfully or fail to do so.

An automated planning system (APS) is introduced to control the robot. It communicates with the robot through TCP/IP socket links. FIP is the key element in APS: Given a set of waypoints, FIP returns a plan by which the robot automatically moves from the initial waypoint to the target waypoint.

We have compared the performance of FIP not only with MBP, but also with classical Graphplan and MEA-Graphplan [9]. To enable classical Graphplan and MEA-Graphplan to deal with this problem, some domain specific functions are included in these two classical planners. These functions help calculate the angles that the robot should turn and check whether the robot is facing the next waypoint. The "rotate" action is also assumed to be deterministic for these two classical planners.

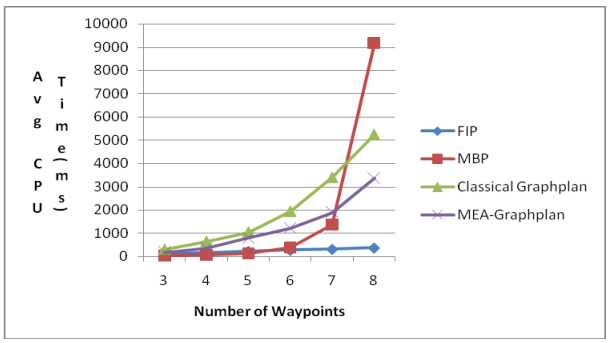

Figure 9: Result for the robot tele-control problem

Experimental results are shown in Figure 9. A few points about the results deserve to be mentioned. First, FIP outperforms all other planners, with the CPU time growing steadily. Second, while MBP outperforms Graphplan and MEA-Graphplan when the number of waypoints is less than 7, its CPU time increases exponentially. Third, MEA-Graphplan outperforms Graphplan. The reason is that MEA-Graphplan adopts the technique of means-ends analysis, which makes the graph expansion phase goal-oriented. More specifically, MEA-Graphplan first grows the planning graph in the backward direction by regressing goals over actions, and then uses the resulting regression-matching graph to guide the standard Graphplan algorithm.

The major reason why the classical planners perform worse than FIP is that the domain specific functions introduce new objects into the planning domain, which in turn makes the search space grow very rapidly. As FIP and MBP can handle nondeterministic actions, they do not have to use domain specific functions to deal with nondeterministic conditions.

4.4. Analysis

Our experimental results demonstrate that FIP consistently outperforms MBP. The main reason for its superior performance can be attributed to the use of planning graphs. As mentioned before, a planning graph is not a complete search state space graph. In particular, (1) the first proposition level P_0 contains only initial conditions, and (2) the proposition levels and actions levels are derived based on P_0 and contain only information that is derivable from P_0. As a result, FIP is less likely to be affected by irrelevant states than MBP. Furthermore, the search space is further reduced by the mutual exclusion relations among propositions. In addition, the search performed by FIP on a planning graph is very efficient, for several reasons. First, FIP conducts forward search along the shortest backbone path generated by the weak plan, and so the overall search distance is optimal. Second, with our novel state transition detection technique, the search distance for remedial actions is greatly reduced. Finally, the search cost is substantially reduced by the fact that FIP only conducts forward search based on the weak plan and does not require any backward chaining search in the second phase, where FIP constructs the full plan.

5. Related Works

MBP [3] is the best known AI planner capable of working on nondeterministic domains. It is based on BDD-based symbolic model checking and is able to create "weak", "strong", and "strong cyclic" planning solutions. "Strong planning" solutions are guaranteed to achieve the goal and "Strong Cyclic planning" solutions have a possibility to terminate and, if they terminate, then the goal is achieved.

GPT [2] is a planning system based on Markov Decision Processes (MDPs). It is able to handle nondeterminism, probabilities, and partial information. The key idea is to represent a planning problem as an optimization problem and use MDP techniques to search through the stochastic space. It can overcome many limitations of classical planning methods. However, it heavily depends on the accuracy and availability of statistical estimations, e.g., probabilities. Moreover, it cannot distinguish acyclic and cyclic solutions [3].

Kplanner [12] uses a different method that separates plan generation from verification. In order to handle some unknown quantity, the algorithm imposes two bounds on the planning parameter *F*. The smaller bound is used to generate a plan under the assumption that *F* will not exceed the bound. The larger bound is used for testing whether the plan is correct. Although

this is a novel solution for iterative planning, it is mainly capable of solving problems of small size.

Winner and Veloso proposed a two-phased method [15] dedicated to plan analysis. Given a totally ordered plan, the first phase algorithm builds a needs tree in the backward direction from the goal state to identify the dependencies between actions. Then, the second phase algorithm, SPRAWL, constructs a partial ordering plan that can help explain the rationale behind the given totally ordered plan. This approach does not generate plans from scratch. Instead, it works on a given totally ordered plan. The purpose of the method is also different: It helps us understand the rationale of the given plan and facilitates plan reuse.

6. Conclusions and Future Work

We have presented FIP, a novel fast iterative planner that can generate iterative and conditional constructs. FIP exploits Graphplan's intrinsic features, i.e., "level-off", producing plans that inherit Graphplan's appealing features (e.g., shortest paths), and exhibits a sound theoretical basis.

As classical Graphplan is both sound and complete, it finds a plan as long as at least one plan exists. This property makes FIP sound because FIP conducts search based on the weak plan from the classical Graphplan. As a result, FIP will not return any plan that is not a solution. Since FIP uses planning graphs to conduct a search, the size of the planning graph and the time required to expand it are polynomial in the size of the planning problem [1]. Several novel techniques, e.g., the use of weak plans to facilitate the final plan generation, proposition level ids serving as intuitive indicators for state transitions, etc., are devised to manipulate planning graphs to improve both the performance and the processing power.

Experimental results on several benchmark planning problems demonstrate that FIP is more efficient than the well-known MBP planner and, more importantly, FIP scales much better as the size of these problem increases.

One future research direction is to apply PDDL to model planning domains to improve FIP's expressiveness. As we do not use any special search techniques in FIP, the performance of FIP has great potentials to be improved. Future research directions also include the development of planner generalization techniques [10] for the large family of Graphplan variants so that classical planning algorithms can be upgraded to handle nondeterministic planning problems. This work will lead to more expressive and higher-performance planning algorithms that can deal with iterative planning.

7. References

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

[2] Bonet, B. and Geffner, H., 2001, GPT: A tool for planning with uncertainty and partial information, *In Proc. of the International Joint Conference on Artificial Intelligence* (*IJCAI*), 82–87.

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

[4] EL-Manzalawy Y. 2006. *http://sourceforge.net/projects/jplan*.

[5] Fikes, R. E. and Nilsson, N. 1971. STRIPS: A new approach to the application of theorem proving to problem solving, *Artificial Intelligence*, 2, 189 – 208.

[6] Gazen, C. and Knoblock, C. A. 1997. Combining the expressivity of UCPOP with the efficiency of Graphplan, In *Proceedings of ECP*-97, 221–233.

[7] Hoffmann, J. and Brafman. R. Contingent Planning via Heuristic Forward Search with Implicit Belief States, *Proceedings of the* 15*th International Conference on Automated Planning and Scheduling*, Monterey, CA, USA, June 2005.

[8] Hoffmann, J. and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search, in *Journal of Artificial Intelligence Research*, Volume 14, 253–302.

[9] Kambhampati, R.; Paeker, E.; and Lambrecht, E. 1997. Understanding and extending graphplan, In *Proc. 4th European Conference on Planning*.

[10] Kuter, U. 2004. Pushing the Limits of AI Planning, *Proc. of the Doctoral Consortium in the* 14*th International Conference on Automated Planning and Scheduling* (*ICAPS*-04).

[11] Kuter, U. and Nau, D. 2004. Forward-chaining planning in nondeterministic domains, In *Proc. of* 19*th the National Conference on Artificial Intelligence* (*AAAI*-2004), 513–518.

[12] Levesque, H. 2005. Planning with loops, in *Proc. of the IJCAI*-05 *Conference*, Edinburgh, Scotland.

[13] McDermott, D., et al. 2004. The PDDL Planning Domain Definition Language, *the AIPS*-2004 *Planning Competition Committee*.

[14] Weld, D.; Anderson, C.; and Smith, D. 1998. Extending GRAPHPLAN to Handle Uncertainty and Sensing Actions. In *Proc. of the* 15*th National Conference on Artificial Intelligence*, 897 – 904.

[15] Winner, E. and Veloso, M. Analyzing plans with conditional effects. *In Proceedings of the* 6*th International Conference on Artificial Intelligence Planning and Scheduling* (*AIPS*-02), Toulouse, France.