Fast Strong Planning for FOND Problems with Multi-Root Directed Acyclic Graphs

Jicheng Fu, Andres Calderon Jaramillo
Computer Science Department
University of Central Oklahoma
Edmond, OK, USA
{jfu, acalderonjaramillo}@uco.edu

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

***Abstract*— We present a planner for addressing a difficult, yet under-investigated class of planning problems: Fully Observable Non-Deterministic planning problems with strong solutions. Our strong planner employs a new data structure, MRDAG (multi-root directed acyclic graph), to define how the solution space should be expanded. We further equip a MRDAG with heuristics to ensure planning towards the relevant search direction. We performed extensive experiments to evaluate MRDAG and the heuristics. Results show that our strong algorithm achieves impressive performance on a variety of benchmark problems: on average it runs more than three orders of magnitude faster than the state-of-the-art planners, MBP and Gamer, and demonstrates significantly better scalability.**

Keywords*: *Fully Observable Non-Deterministic (FOND) planning, Strong Cyclic Planning, Strong Planning

I. Introduction

Fully-observable nondeterministic (FOND) planning is a challenging research area [1]. For the purpose of addressing nondeterministic planning problems, Cimatti *et al*. [2] present a three-way categorization of planning solutions: *weak* solutions have a chance to achieve the goal; *strong* solutions are guaranteed to achieve the goal; and *strong-cyclic* solutions have a chance to terminate and if they do, they are guaranteed to achieve the goal [2]. Thus, strong solutions are more desirable than weak and strong-cyclic solutions as they are guaranteed to achieve the goal.

Despite the importance of strong planning, it is an under-investigated area of FOND planning. Among the planners that are capable of solving strong FOND problems, the two that are most well-known are arguably MBP and Gamer [3]. Both planners, however, employ symbolic regression breadth-first search to search backward from the goal state to the initial state, which makes it difficult for them to plan efficiently and scale to large problems.

Our goal in this paper is to design a planner that can offer state-of-the-art performance on FOND planning problems with strong solutions. One possibility is to extend state-of-the-art FOND planners such as FIP [4] and PRP [5] so that they return strong solutions. Recall that these two FOND planners are not guaranteed to return a strong solution even if one exists, but since they outperform Gamer and MBP on benchmark strong-cyclic problems by several orders of magnitude, they might be able to beat Gamer and MBP on strong problems if they are extended to return strong solutions.

However, FIP and PRP have a common weakness: they rely on a classical deterministic planner to establish a weak plan from each non-goal leaf state (i.e., a state that has not been assigned an action in the solution state space) to the goal state. The use of classical planners implies less control over planning efficiency. Specifically, when a classical planner runs longer than expected, it is hard to determine whether it needs more time to finish or it is stuck in some hopeless situation. This issue may aggravate if we have to plan under time constraints. If it times out on any single search for a weak plan, the entire planning process will fail.

Given the above discussion, we desire a planner that (1) has full control over how to expand the solution space by not relying on a classical planner, and (2) uses heuristics to ensure planning towards the relevant search direction, thus overcoming the inefficiency inherent in the uninformed search methods employed by MBP and Gamer. There is an additional property desirable of a planner: the ability to handle backtracks efficiently.

To understand the importance of efficient backtracking in strong planning, recall that cycles are constantly encountered and should be avoided during a strong planning process. Suppose that state s has only one applicable action, a. If a cycle is formed due to applying action a to state s, then (1) action a will be made inapplicable to state s; (2) state s becomes a dead-end as its only applicable action a has been made inapplicable; and (3) the algorithm backtracks from state s. Backtrack will continue until it reaches a state that has more than one applicable action. In other words, backtrack has to occur step by step, where in each step, it needs to check the number of actions applicable to each state, and backtrack until it reaches a state with more than one applicable action. Hence, to handle cycles more efficiently, we propose to distinguish states with one applicable action and those with more than one applicable action. In fact, states with only one applicable action are very common. We examined the benchmark problems in the International Planning Competition 2008 (IPC 2008) [6] and found that about 25% of the states had only one applicable action. Moreover, as the planning process goes on, more states will become those with only one applicable action because if an applicable action results in a cycle or a dead-end, this action will be made inapplicable to the state. As a result, the state will have fewer applicable actions.

In light of the three desirable properties mentioned above, we present a planner that builds upon two novel ideas. First, we propose a new data structure, MRDAG (multi-root directed acyclic graph), which defines how the solution space should be expanded by distinguishing states with one applicable action from those with more than one applicable action. Second, we equip a MRDAG with heuristics that define the order in which the actions applicable to a state within the MRDAG should be chosen.

We conducted extensive experiments to evaluate the proposed planner and compare performance between our planner and other state-of-the-art planners, i.e., MBP and Gamer. To ensure fairness in our evaluation, all the planning domains were derived from the FOND track of IPC 2008 [6]. Experimental results show that our strong algorithm achieves impressive performance on a variety of benchmark problems: on average, it runs more than three orders of magnitude faster than MBP and Gamer and demonstrates significantly better scalability.

We believe that our results are a very strong indication that our approach is significantly more suitable for efficient strong planning than symbolic regression breadth-first search, the search method adopted by Gamer and MBP.

II. 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; 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 $\langle s_0, g, \Sigma\rangle$ consists of three components, namely, the initial state s_0, the goal condition g, and the planning domain Σ.

Definition 3: Given a planning problem $\langle s_0, g, \Sigma\rangle$, 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 $(s, a) \in \pi$. We use $S_\pi(s)$ to denote the set of states reachable from s using π.

Definition 4: (taken from Bryce & Buffet [7]). 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 *acyclic* with respect to s_i iff there is no trajectory $(s_i, \pi(s_i), s_{i+1}, \pi(s_{i+1}),\ldots, s_j, \pi(s_j), \ldots, s_k, \pi(s_k), \ldots, s_n)$ with j and k such that $i \le j < k \le n$ and $s_j = s_k$. π is a *strong* solution for the non-deterministic problem iff π is closed, proper, and acyclic with respect to the initial state s_0.

Note that an acyclic π defines (and hence can be equivalently represented as) a directed acyclic graph (DAG) $G_\pi = \{V_\pi, E_\pi\}$, where $V_\pi = S_\pi \cup \{\gamma(s, \pi(s)) \mid s \in S_\pi\}$ is the set of vertices 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. $G_\pi(s_0)$, a directed acyclic graph (DAG) rooted at s_0, initially contains only the initial state s_0. Our strong planner aims to augment π (or equivalently, G_π) by using a special data structure, MRDAG, to guide the expansion of the solution space, as discussed next.

III. Multi-Root Directed Acyclic Graph

In this section, we define a MRDAG and its properties formally. We begin by presenting an informal overview of it.

Figure 1 shows an example of how MRDAGs control the expansion of the solution space. Each M_i is a MRDAG, which consists of a set of DAGs. The set of roots of the DAGs in a MRDAG is called the *rootset* of the MRDAG. The black nodes in Figure 1 are the states in the rootset of a MRDAG. Except for the initial state s_0, a state is in a rootset iff it has more than one applicable action.

The search process begins by expanding the rootset of the first MRDAG, M_1, which has only one element, s_0. The process of state expansion continues until every leaf node either is a goal node or has more than one applicable action. The non-leaf nodes expanded so far belong to M_1, and the set of non-goal leaf nodes defines the rootset of M_2. Each state in the rootset of M_2 is expanded in a similar manner until each leaf node either is a goal node or has more than one applicable action, and those non-goal leaf nodes belong to the rootset of M_3. This process produces a sequence of MRDAGs and stops when all leaf nodes are goal nodes.

Hence, the MRDAGs define how the solution space is expanded: they separate the “easy” states (i.e., states with only one applicable action) from the “hard” states (i.e., states with more than one applicable action). The questions then are (1) how to impose an ordering on the actions to be chosen for a hard state, and (2) how to impose an ordering on the states to be expanded in the same rootset? As we will see, heuristics will be used to impose these orderings.

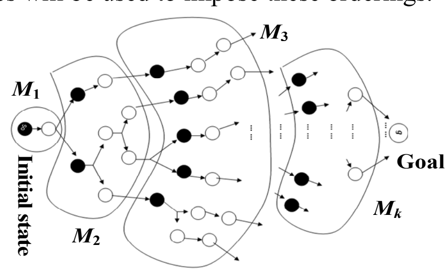

Figure 1. Solution expansion with MRDAGs

Next, we define a MRDAG and its properties formally.

Definition 5: A MRDAG $M = \{S_{Mr}, \pi_M\}$ consists of two elements, namely, a rootset S_{Mr} and a policy π_M, with the following properties:

(1) $S_{Mr} = \{s_{r1}, s_{r2}, \ldots, s_{rk}\} \subseteq S_{\pi M}$ consists of a set of states, where $S_{\pi M}$ denotes the set of states contained in π_M;

(2) $\forall(s, a) \in \pi_M, s \notin S_{Mr} \rightarrow |A(s)| = 1$, where $A(s)$ is the set of actions applicable to state s. That is, if s is not in S_{Mr}, then it has exactly one applicable action.

Intuitively, before a MRDAG is expanded, its rootset S_{Mr} includes all non-goal leaf states in $G_\pi(s_0)$. For convenience, we will say that a state s **belongs to** M if $s \in S_{\pi M}$.

Definition 6: A state s is called an outsider of a MRDAG $M = \{S_{Mr}, \pi_M\}$ if one of the following two conditions is satisfied:

(1) s is a goal;

(2) there exists $(s', a') \in \pi_M$ such that $s \in \gamma(s', a')$; in addition, $|A(s)| > 1$ and s does not belong to any of M's *ancestry* MRDAGs (i.e., MRDAGs constructed prior to M).

Definition 6 implies that the outsiders of a MRDAG M are not part of M. These outsiders represent the set of all non-goal leaf states generated by M in $G_\pi(s_0)$.

Definition 7: A MRDAG M_c rooted at S_{Mcr} is a child of MRDAG M_p if S_{Mcr} is the set of all non-goal outsiders of M_p. M_p is called the parent of M_c.

Definition 7 implies that a MRDAG can have at most one child MRDAG. Definition 6 and Definition 7 together imply the following property for MRDAG expansion.

Property (MRDAG Expansion): Given a MRDAG $M = \{S_{Mr}, \pi_M\}$, if (1) there exists a state s' that does not appear in M's ancestry MRDAGs; (2) $|A(s')| = 1$; and (3) there exists $(s, a) \in \pi_M$ such that $s' \in \gamma(s, a)$, then $(s', a') \in \pi_M$, where a' is the only applicable action of s'.

Definition 8: A MRDAG $M = \{S_{Mr}, \pi_M\}$ is feasible if the following three conditions are satisfied:

(1) $\forall(s, a) \in \pi_M$, applying a to s does not lead to a cycle in $G_\pi(s_0)$;

(2) $\forall(s, a) \in \pi_M$, applying a to s does not lead to a dead-end; and

(3) the child of M, if any, is also feasible.

Definition 9: A set of states $S_{Mr} = \{s_{r1}, s_{r2}, \ldots, s_{rk}\}$ is called a feasible rootset if a feasible MRDAG rooted at S_{Mr} can be created.

Definition 10: A strong solution is $\pi = \pi_{M1} \cup \pi_{M2} \cup \ldots \cup \pi_{Mn}$, where $\pi_{M1}, \pi_{M2}, \ldots, \pi_{Mn}$ are the policies of a sequence of MRDAGs $M_1, M_2, \ldots, M_n$, if the following three conditions are satisfied:

(1) M_1 is rooted at s_0, i.e., the initial state;

(2) M_i is the parent of M_{i+1} for $i = 1, 2, 3, \ldots, n - 1$; and

(3) all the outsiders of M_n are goal states.

IV. STRONG PLANNING ALGORITHM

A. Algorithm Outline

Figure 2 outlines our strong planning algorithm. In line 1, the rootset R of the first MRDAG is initialized to be the initial state s_0 of the planning problem $\langle s_0, g, \Sigma\rangle$. The policy π, which stores the union of the policies of the MRDAGs constructed up to this point, is initialized to be an empty set. While R is not empty (line 2), the function *GET-NEXT-SET-OF-ACTIONS* assigns an applicable action to *each* state in R, and the resulting state-action pairs are inserted into π_M, the policy associated with the current MRDAG (line 3). Note that *GET-NEXT-SET-OF-ACTIONS* enumerates all possible combinations of actions applicable to the states in R, and returns a different combination of actions for the same rootset every time it is invoked. For example, assume that there are two states in R, namely, s_{r1} and s_{r2}. If $|A(s_{r1})| = 2$ and $|A(s_{r2})| = 3$, then there are 6 possible combinations for creating π_M. Each time *GET-NEXT-SET-OF-ACTIONS* returns one combination to π_M (line 3). If all the combinations have been exhausted (line 4), *GET-NEXT-SET-OF-ACTIONS* will return an empty set, which means R is not a feasible rootset (see Definition 9), i.e., no feasible MRDAG can be built from R. When this happens, the algorithm will check whether R includes only s_0 (line 5). If so, there is no solution to the given planning problem. However, if R includes some states other than s_0, backtrack will occur (line 6). As no feasible MRDAGs can be built based on R, the MRDAG leading to R (i.e., R's parent MRDAG) is not feasible and should be discarded. Specifically, all state-action pairs in the policy of R's parent MRDAG are discarded (the policy π is updated accordingly) and only its rootset is kept. Hence, the result of the backtrack is to assign the parent's root to R. Therefore, in the next iteration (line 3), *GET-NEXT-SET-OF-ACTIONS* will assign a different set of actions to the states in R so that the algorithm will seek an alternative solution by building a different MRDAG.

```
Global Variables: π, ⟨s0, g, Σ⟩
Function STRONG_PLANNING
1.  R ← {s0}; π ← ϕ        /*R is the rootset of the MRDAG*/
2.  while R ≠ ϕ do
3.      πM ← GET-NEXT-SET-OF-ACTIONS(R)
4.      if πM = ϕ then
5.          if R = {s0} then return FAILURE else
6.              BACKTRACK(R)
7.          endif
8.      else
9.          if BUILD-MRDAG(πM) <> FAILURE then
10.             π ← π ∪ πM
11.             if All-GOAL-OUTSIDERS(R, πM) then
12.                 return π
13.             else
14.                 R ← GET-OUTSIDERS(R, πM)
15.             endif
16.         endif
17.     endif
18. endwhile
```

Figure 2. Outline of the strong planning algorithm

If *GET-NEXT-SET-OF-ACTIONS* returns a non-empty set (line 8), the algorithm attempts to build a feasible MRDAG by invoking the function *BUILD-MRDAG* (line 9). Figure 3 illustrates how to build a feasible MRDAG. If a feasible MRDAG is not found (i.e., *BUILD-MRDAG* returns failure), the current iteration ends. In the next iteration (line 3), *GET-NEXT-SET-OF-ACTIONS* will return a different set of actions to the states in R.

On the other hand, if a feasible MRDAG can be built (i.e., *BUILD-MRDAG* returns success), the algorithm adds the state-action pairs in π_M to the solution policy π (line 10). Then, it checks whether the outsiders of the current MRDAG are all goal states (line 11). If so, a solution has been found (line 12) according to Definition 10. Otherwise, the set of non-goal outsiders of the current MRDAG is assigned to R, which will be the rootset of the child MRDAG (line 14). The algorithm then continues to the next iteration and attempts to build a feasible child MRDAG based on the new rootset.

B. Building a Feasible MRDAG

Figure 3 illustrates how to build a feasible MRDAG. A copy of π_M, the input argument to *BUILD-MRDAG*, is saved to π_{root} (line 1). Since the current MRDAG has not yet been expanded, π_M contains only the state-action pairs for states in the rootset at the moment. Then, the algorithm expands each state s in the rootset by invoking the recursive function *EXPAND-MRDAG* (line 3).

```
Function BUILD-MRDAG (π_M)
1.  π_root ← π_M
2.  foreach (s, a) ∈ (π_root) do
3.      if EXPAND-MRDAG(π_M, s, a) = FAILURE then
4.              return FAILURE
5.      endif
6.  endfor
7.  return SUCCESS
```

Figure 3. Algorithm for building a feasible MRDAG

EXPAND-MRDAG is shown in Figure 4. For each state s' $\in$ $\gamma(s, a)$ that is not a goal state (line 1), the algorithm checks whether s' has already been assigned an action in π or π_M (line 2). If so, the algorithm uses Tarjan's algorithm [8] to check whether a cycle has been formed in the graph represented by the union of π and π_M as a result of applying a to s (line 3).[1] If a cycle is detected, the use of action a violates the acyclic property of MRDAG (see Definition 8), and the algorithm returns FAILURE (line 4 in Figure 4). Otherwise, the use of a is safe. Since s' has been already assigned an action, there is no need to expand it.

```
Function EXPAND-MRDAG (π_M, s, a)
1.  foreach s′ ∈ γ(s, a) & NOT-GOAL(s′) do
2.      if s′ ∈ S_π or s′ ∈ S_πM then
3.          if DETECT-CYCLE(π ∪ π_M) = TRUE then
4.              return FAILURE
5.          endif
6.      elseif |A(s′)| = 1 then
7.          π_M ← π_M ∪ {(s′, a′)} with a′ ∈ A(s′)
8.          if EXPAND-MRDAG (π_M, s′, a′) = FAILURE then
9.              return FAILURE
10.         endif
11.     elseif |A(s')| = 0 then /*dead-end*/
12.             return FAILURE
13.     endif
14. endfor
15. return SUCCESS
```

Figure 4. Helper function for building a feasible MRDAG

[1] When detecting cycles, we need to take the union of π and π_M because a cycle could be formed among the states belonging to different MRDAGs.

If s' has not been assigned any action, the algorithm checks the number of actions applicable to s'. If there is only one applicable action (line 6), it should belong to the current MRDAG (see Definition 5 and Property (MRDAG expansion)). Hence, the algorithm includes s' in the current MRDAG (line 7) and then recursively expands s' (lines 8–10). On the other hand, if s' has no applicable actions (line 11), it is a dead-end and a failure has been detected (line 12), since a feasible MRDAG should not lead to any dead-end (see Definition 8). Note that the algorithm does not handle the case where $|A(s')| > 1$. The reason is that s' is an outsider of the current MRDAG according to Definition 6.

C. An Illustrative Example

To better understand our strong planning algorithm, we apply it to the following blocksworld problem (see Figure 5), which will serve as our running example. In this problem, three actions are possible: the deterministic action *put-down*(B) puts block B onto the table, whereas the two actions *pick-up*(A, B) and *put-on*(A, B) are nondeterministic since the held block A may fall onto the table. The aim is to move the blocks so that goal g can be reached from initial state s_0.

The algorithm begins by setting the rootset R of the first MRDAG to $\{s_0\}$ (line 1 in Figure 2). Next, it computes the policy π_M based on R (line 3 in Figure 2). Since *pick-up*(B, A) is the only applicable action to s_0, π_M is set to $\{(s_0,$ *pick-up*(B, A)$)\}$. Since π_M is not empty, the algorithm attempts to build a feasible MRDAG by invoking *BUILD-MRDAG* (line 9 Figure 2). Subsequently, *BUILD-MRDAG* invokes *EXPAND-MRDAG* to recursively expand the MRDAG (line 3 in Figure 3). Applying *pick-up*(B, A) to s_0 results in two states (line 1 in Figure 4). One is the goal since block B may fall onto the table. The other is state s_1 in Figure 5, which is a state in which B is held. The goal state will be an outsider of the current MRDAG (see Definition 6). Since s_1 has three applicable actions, namely, *put-down*(B), *put-on*(B, A), and *put-on*(B, C), it is also an outsider of the current MRDAG. Then, a new MRDAG, M_1, is created with $\pi_{M1} = \{(s_0,$ *pick-up*(B, A)$)\}$, and s_1 is the non-goal outsider. The control of the algorithm returns to line 9 of Figure 2. The solution π is updated to be $\pi \leftarrow \pi \cup \pi_{M1} = \{(s_0,$ *pick-up*(B, A)$)\}$ (line 10 of Figure 2). Since state s_1 is not a goal state and it is an outsider of the current MRDAG, $R = \{s_1\}$ (line 14 in Figure 2).

The algorithm begins the next iteration by selecting an applicable action for s_1. Let us assume that the algorithm selects *put-on*(B, C) (line 3 of Figure 2). It then invokes *BUILD-MRDAG* with the argument $\pi_M = \{(s_1,$ *put-on*(B, C)$)\}$ (line 9 of Figure 2). Subsequently, *BUILD-MRDAG* invokes *EXPAND-MRDAG* to recursively expand the MRDAG (lines 1–3 of Figure 3). Applying *put-on*(B, C) to s_1 also leads to two states, namely, (1) the goal state, since B may fall onto the table, and (2) state s_2, as shown in Figure 5 (line 1 in Figure 4). State s_2 has only one applicable action, i.e., *pick-up*(B, C). So the algorithm adds s_2 to the current MRDAG and recursively invokes *EXPAND-MRDAG* (lines 8–10 of Figure 4). Now, $\pi_M = \{(s_1,$ *put-on*(B, C)$), (s_2,$ *pick-up*(B, C)$)\}$. Applying *pick-up*(B, C) to s_2 results in two states, namely, the goal state and a previously explored state, i.e., s_1. The handling of the goal state is the same as above, so let us focus on state s_1. The algorithm detects that s_1 has been assigned an action (line 2 of Figure 4) and then finds that a cycle has been formed between s_1 and s_2 (line 3 of Figure 4). Hence, *EXPAND-MRDAG* returns failure (line 4 in Figure 4). Subsequently, *BUILD-MRDAG* also returns failure (line 4 in Figure 3). Hence, policy $\pi = (s_0, \textit{pick-up}(\text{B}, \text{A}))$ is not updated. Then, the strong planning algorithm (Figure 2) selects another action, say *put-down*(B), for $R = \{s_1\}$, and invokes *BUILD-MRDAG* (line 9 in Figure 2) with $\pi_M = \{(s_1, \textit{put-down}(\text{B}))\}$. *BUILD-MRDAG* then invokes *EXPAND-MRDAG*. As *put-down*(B) is a deterministic action, it only results in a single state, which is the goal (line 1 of Figure 4). The goal state is an outsider of the MRDAG. Since there are no other states generated by *put-down*(B) , the algorithm will return SUCCESS (line 15 in Figure 4) to *BUILD-MRDAG* and then to the strong planning algorithm in Figure 2. The current MRDAG M_2 includes the policy $\pi_M = \{(s_1, \textit{put-down}(\text{B}))\}$. Then, policy π is updated by $\pi \leftarrow \pi \cup \pi_M$, i.e., $\pi = (s_0, \textit{pick-up}(\text{B}, \text{A})) \cup \{(s_1, \textit{put-down}(\text{B}))\} = \{(s_0, \textit{pick-up}(\text{B}, \text{A})), (s_1, \textit{put-down}(\text{B}))\}$ (line 10 of Figure 2). Since the outsiders of MRDAG M_2 include only the goal state, the algorithm terminates and returns the final policy π (lines 10–12 in Figure 2).

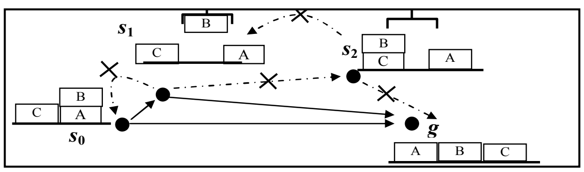

Figure 5. Blocksworld example

D. Heuristics

Two questions arise. First, which state in a given rootset should be expanded first if the rootset contains more than one state? Second, which action applicable to a state in a rootset should be applied first if the state contains more than one applicable action? We answer these questions by designing two heuristics, as described below.

To answer the first question, assume that the rootset of a MRDAG is $S_{Mr} = \{s_{r1}, s_{r2}, \ldots, s_{rk}\}$. Using the *most constrained state* (MCS) heuristic, we sort the states in S_{Mr} in increasing order of the number of actions applicable to a state. The MCS heuristic enables a simple and efficient way to implement *GET-NEXT-SET-OF-ACTIONS* (line 3 of Figure 2). Specifically, assume that $S_{Mr} = \{s_{r1}, s_{r2}, \ldots, s_{rk}\}$ is sorted by means of the MCS heuristic. For each state s_{ri} $(1 \leq i \leq k)$ in S_{Mr}, let $A_i = (a_{i1}, a_{i2}, \ldots, a_{i<mi>})$ be the list of applicable actions to s_{ri} and $<mi> = |A(s_{ri})|$ be the number of applicable actions. We assume that *GET-NEXT-SET-OF-ACTIONS* retrieves the actions in A_i in a fixed order, i.e., $a_{i1}, a_{i2}, \ldots, a_{i<mi>}$. Then, *GET-NEXT-SET-OF-ACTIONS* returns $\pi_M = \{(s_{r1}, \boldsymbol{a}_{\mathbf{11}}), (s_{r2}, a_{21}), \ldots, (s_{rk}, a_{k1})\}$ first. If it does not result in a feasible MRDAG, the function will try s_{r1}'s next action a_{12} and return $\{(s_{r1}, \boldsymbol{a}_{\mathbf{12}}), (s_{r2}, a_{21}), \ldots, (s_{rk}, a_{k1})\}$, then $\{(s_{r1}, \boldsymbol{a}_{\mathbf{13}}), (s_{r2}, a_{21}), \ldots, (s_{rk}, a_{k1})\}$, …, and finally $\{(s_{r1}, \boldsymbol{a}_{\mathbf{1<m1>}}), (s_{r2}, a_{21}), \ldots, (s_{rk}, a_{k1})\}$. If still no feasible MRDAG can be built, the function will try s_{r2}'s next action a_{22} and return $\{(s_{r1}, \boldsymbol{a}_{\mathbf{11}}), (s_{r2}, \boldsymbol{a}_{\mathbf{22}}), \ldots, (s_{rk}, a_{k1})\}$. If this combination does not lead to a feasible MRDAG, then the function will return $\{(s_{r1}, \boldsymbol{a}_{\mathbf{12}}), (s_{r2}, \boldsymbol{a}_{\mathbf{22}}), \ldots, (s_{rk}, a_{k1})\}$, $\{(s_{r1}, \boldsymbol{a}_{\mathbf{13}}), (s_{r2}, \boldsymbol{a}_{\mathbf{22}}), \ldots, (s_{rk}, a_{k1})\}$, …, and finally $\{(s_{r1}, a_{1<m1>}), (sr_2, a_{2<m2>}), \ldots, (s_{rk}, a_{k<mk>})\}$. Here is the rationale behind the MCS heuristic: as s_{r1} has the least number of applicable actions, *GET-NEXT-SET-OF-ACTIONS* can quickly enumerate its applicable actions and then start to consider the rest of the states in sorted order.

To answer the second question, we use the *least heuristic distance* (LHD) heuristic. For each state $s_{ri} \in S_{Mr} = \{s_{r1}, s_{r2}, \ldots, s_{rk}\}$ $(1 \leq i \leq k)$, we sort its applicable actions in increasing order of the heuristic distance to the goal. Specifically, applying an action a to s_{ri} may result in a set of states. Among these resulting states, the one that yields the shortest distance to the goal is used to define the heuristic distance of action a. In our implementation, we used the same heuristic as FF [9], i.e., relaxed plans, to estimate the heuristic distance. To break ties, actions with fewer effects are given higher priority. The rationale is that if an action has fewer effects, it is less nondeterministic and hence contains less unintended effects. If a tie still exists, then it is broken arbitrarily. The LHD heuristic is also implemented in *GET-NEXT-SET-OF-ACTIONS*. It should be easy to see that MCS and LHD can be applied in combination.

Theorem 1: A MRDAG $M = \{S_{Mr}, \pi_M\}$ can be uniquely identified by S_{Mr} and the set of actions applied to S_{Mr}, A_r.

Proof sketch. According to Definition 5, except the states in S_{Mr}, all other states in M only have a single applicable action. Hence, after applying A_r to S_{Mr}, the expansion of M has no variations. If a generated state is not already in s_π, then it either is an outsider of M if it has more than one applicable action, or can continue to expand M by applying its only applicable action. Therefore, with S_{Mr} and A_r, we cannot obtain two different MRDAGs.

Theorem 2: The proposed strong planning algorithm in Figure 2 is sound and complete.

Proof sketch. To prove soundness, assume that the algorithm returns a solution consisting of a sequence of MRDAGs $M_1, M_2, \ldots, M_n$, where M_i is the parent of M_{i+1} for $i = 1, 2, 3, \ldots, n-1$. Note that each MRDAG in the sequence is feasible, as our algorithm maintains the feasibility of MRDAGs, i.e., there are no dead-ends (see lines 11 and 12 of Figure 4) or cycles (see lines 3 and 4 of Figure 4) in the MRDAGs. In addition, the possible non-goal leaf states can only exist in a MRDAG's outsiders (see lines 6–13 of Figure 4), which form the root of its child. Hence, we only need to check the last MRDAG, M_n. The algorithm terminates with success if and only if the outsiders of M_n are all goal states (lines 11 and 12 in Figure 2). Hence, no non-goal leaf states are possible in the solution, i.e., there is a path leading to the goal from any non-goal state in the solution without going through any cycles.

Completeness can be proved by contradiction. Suppose that there is a solution to the given planning problem but our planning algorithm terminates in failure. According to Definition 10, we can represent the solution by a sequence of MRDAGs, M_1, M_2, …, and M_n,. Here, M_i is the parent of M_{i+1} for i = 1, 2, 3, …, n - 1 and M_1's rootset contains only the initial state s_0 of the planning problem. According to Theorem 1, M_1 is determined by $\{s_0\}$ and an action a. Our algorithm should be able to try action a because our algorithm exhaustively tries all the possible combinations of actions applicable to the root to expand a MRDAG. Hence, M_1 will be created based on the root s_0 and action a. By induction, it will create M_2, …, M_n, which is a solution to the planning problem. Hence, we obtain a contradiction.

V. EVALUATION

To ensure fairness in our experiments, we used the benchmark planning domains from the IPC 2008 FOND track [6]. Since no problem in IPC 2008 has strong solutions, we created problems with strong solutions by revising four benchmark domains in the FOND track, namely faults [ft], tireworld [tw], blocksworld [bw], and first-responders [fr]. To test how fast a strong algorithm can report failure to find a strong solution, we also used the strong cyclic blocksworld domain [scbw].

We revised the four aforementioned benchmark domains so that they contain strong solutions as follows.

Faults: Here, the goal is to complete a set of operations. We relax the requirements to allow operations to complete even with faults. With this relaxation, it is possible to generate strong solutions with problem instances p_*x*_*x*, where the first *x* represents the number of operations and the second *x* represents the maximum number of allowable faults.

Tireworld: Here, the goal is to drive a car from the initial location to the goal location through a series of intermediate stops. Of the three possible actions, move-car and change-tire are nondeterministic. Move-car may or may not have a flat tire when moving from one location to another. Change-tire may or may not change the tire successfully. The original tireworld domain only has strong cyclic solutions because change-tire, if failed, will do nothing. We modified change-tire so that it is deterministic (i.e., no failure is possible), keeping everything else unchanged.

Blocksworld: We enhanced blocksworld by combining it with the faults domain. The *pick-up* action may become faulty and need a repair. The goal condition of each problem is the configuration where all the blocks are on the table. Solving blocksworld problems is by no means trivial: Gamer can only solve 10 out of 30 problems while MBP can solve none.

First-responders: We revised the first-responders by changing three nondeterministic actions. In the original domain, the fire may or may not be put out by unloading the fire unit. In addition, victims hurt by fire can be treated on the scene at a fire unit or a medical unit. The treat action either heals the victims or does nothing. We change the "unload-fire-unit" to be deterministic, i.e., fire can always be put out. We change the two "treat-victim-on-scene" actions to generate the effects of healing the victim or the victim becoming dying. In the latter case, the victim must be sent to the hospital using a vehicle.

A. Planners

As baselines, we use MBP and Gamer. In addition, to determine the contributions made by the two heuristics, we evaluate four versions of our planners: SP uses both heuristics, MCS uses only the MCS heuristic, LHD uses only the LHD heuristic, and NOH uses none of the heuristics. In NOH and LHD, the states in the rootset of a MRDAG are expanded in the order in which they are added to the rootset when BUILD-MRDAG or EXPAND-MRDAG is called. In MCS and NOH, if a state has more than one applicable action, one of the actions will be randomly chosen to expand the MRDAG.

B. Problem Coverage

Table I shows the problem coverage (i.e., the number of problems for which a strong plan was found or not found). Note that these results were obtained using a desktop computer with Intel Pentium-4 CPU 3GHz and 1 GB memory. We set the cutoff time to 1,200 seconds to prevent a planner from running indefinitely. As we can see, the four versions of our planner (SP, LHD, MCS, and NOH) demonstrate outstanding scalability by solving a significantly larger number of problems than Gamer and MBP. Specifically, LHD achieves the highest problem coverage followed by SP. Although NOH and MCS solve fewer problems than LHD and SP, they have a high coverage on the strong cyclic blocksworld (scbw) and first-responders (fr) domains. MCS performs slightly better than NOH. These results suggest that MCS does not contribute significantly to the problem coverage.

TABLE I. PROBLEM COVERAGE

Domain	Gamer	MBP	SP	LHD	MCS	NOH
***scbw* (30)**	10	10	29	30	30	30
bw(30)	10	0	30	30	10	10
ft (10)	6	4	10	10	3	3
tw (12)	11	0	12	12	5	4
fr (50)	20	10	49	49	46	45
Total (132)	57	24	130	131	94	92

C. CPU Time and Plan Size

Next, we evaluate the planners with respect to CPU time and plan size. The CPU time refers to the time required by a planner to find a strong solution (if one exists) or report that a strong solution does not exist. Note that to ensure a fair comparison, we only compare the pure search time, as the preprocessing time of Gamer and MBP is lengthy. The plan size is associated with the quality of the plans: the smaller the better. Table II shows the evaluation results. Only the difficult problems (i.e., problems for which at least one planner timed out or took > 50 seconds to find a solution) are listed. The column labeled with "*t*" shows the CPU time measured in seconds and the column labeled with "*s*" shows the size of the plans. "---" indicates that the planner timed out on that problem. As randomness is involved in SP, LHD, MCS, and NOH, we ran each problem three times and calculated average results. If a planner failed to find a solution on any of the three trials, we marked it as "---".

While Gamer performed much better than MBP on all the domains, our planners performed significantly better than Gamer. On average, SP and LHD are about 4 orders of magnitude faster than Gamer on strong blocksworld, first-responders, and tiresworld, about 3 orders of magnitude faster than Gamer on faults, and 2 orders of magnitude faster on strong cyclic blocksworld. In addition, SP and LHD generate plans with size comparable to those generated by Gamer in most of the cases.

In terms of the contributions made by the two heuristics, LHD is on average 5 times faster on first-responders, and up to 2 orders of magnitude faster on tireworld and 3 orders of magnitude faster on faults than MCS. On the other hand, MCS is about 3 times faster than LHD on strong and strong cyclic blocksworld domains. In terms of plan size, LHD consistently generates much compacter plans than MCS. On average MCS's plans are about 35 times larger than those of LHD.

In addition, MCS is only marginally faster than NOH on average. When MCS and LHD are used in combination in SP, SP's performance is only marginally better than LHD. Hence, we can conclude that MCS does not contribute as significantly as LHD to the planning performance. In other words, the order in which the states within the rootset of a MRDAG are expanded does not seem to make a big difference in performance.

TABLE II. CPU TIME AND PLAN SIZE COMPARISON

Problem	Gamer		MBP[2]	SP		LHD		MCS		NOH	
	t	*s*	*t*	*t*	*s*	*t*	*s*	*t*	*s*	*t*	*s*
scbw-1	0.760	NA	148.346	0.003	NA	0.002	NA	0.001	NA	0.001	NA
scbw-2	1.244	NA	221.011	0.001	NA	0.001	NA	0.001	NA	0.001	NA
scbw-3	0.961	NA	167.435	0.003	NA	0.003	NA	0.001	NA	0.001	NA
scbw-6	0.658	NA	70.287	0.002	NA	0.002	NA	0.001	NA	0.001	NA
scbw-8	0.633	NA	57.433	0.003	NA	0.002	NA	0.001	NA	0.001	NA
scbw-9	1.001	NA	228.980	0.001	NA	0.002	NA	0.001	NA	0.001	NA
scbw-10	0.911	NA	232.064	0.003	NA	0.003	NA	0.001	NA	0.001	NA
scbw-20	---	---	---	0.119	NA	0.141	NA	0.041	NA	0.049	NA
scbw-30	---	---	---	0.326	NA	0.344	NA	0.057	NA	0.057	NA
bw-1	89.462	21	---	0.003	21	0.003	21	0.001	33	0.001	21
bw-2	86.071	14	---	0.002	14	0.001	14	0.001	23	0.001	39
bw-3	86.888	21	---	0.003	21	0.003	21	0.001	38	0.001	33
bw-5	88.048	21	---	0.003	21	0.003	21	0.001	33	0.001	68
bw-6	87.177	14	---	0.002	14	0.002	14	0.001	14	0.001	21
bw-7	87.738	28	---	0.004	28	0.005	28	0.002	47	0.001	52
bw-8	85.607	28	---	0.004	28	0.004	28	0.001	45	0.002	47
bw-9	87.953	28	---	0.004	28	0.004	28	0.003	104	0.002	100
bw-10	88.974	21	---	0.003	21	0.003	21	0.001	31	0.001	38
bw-20	---	---	---	0.059	40	0.056	40	---	---	---	---
bw-30	---	---	---	0.557	65	1.157	65	---	---	---	---
ft-6-6	291.790	127	---	0.012	127	0.012	127	---	---	---	---
ft-8-8	---	---	---	0.088	511	0.089	511	---	---	---	---
ft-9-9	---	---	---	0.237	1023	0.235	1023	---	---	---	---
ft-10-10	---	---	---	0.620	2047	0.619	2047	---	---	---	---
tw-10	234.021	1	---	0.001	1	0.001	1	---	---	0.770	868
tw-11	241.141	5	---	0.001	5	0.001	5	---	---	---	---
tw-12	242.036	1	---	0.001	1	0.001	1	---	---	---	---
tw-14	95.095	21	---	0.009	34	0.009	32	---	---	---	---
fr-1-8	10.046	10	55.377	0.002	10	0.003	10	0.006	172	0.010	328
fr-1-9	52.265	11	296.332	0.003	11	0.003	11	---	---	0.016	448
fr-1-10	721.715	12	---	0.004	12	0.004	12	0.044	1037	0.036	857
fr-10-1	0.754	3	---	0.012	3	0.011	3	0.022	95	0.070	289
fr-10-2	---	---	---	0.013	12	0.012	11	0.081	505	0.030	197

[2] MBP often outputs too much information to count policy size.

VI. CONCLUSIONS

In this paper, we presented a strong algorithm for FOND planning problems that exploited a novel data structure, MRDAG (multi-root directed acyclic graph). We conducted extensive experiments to evaluate how planning performance is affected by (1) the order in which the actions applicable to a state are chosen and (2) the order in which the states in the rootset of a MRDAG are expanded via the proposal of two heuristics, MCS and LHD. Experimental results on four domains showed that the use of MRDAG indeed made cycle handling easier and more efficient and the use of the LHD heuristic significantly improved planning performance. Most importantly, our planner significantly outperformed two state-of-the-art planners, Gamer and MBP, by solving more problems in less time.

ACKNOWLEDGMENT

This work was supported in part by the Oklahoma Center for the Advancement of Science and Technology (OCAST), HR12-036.

REFERENCES

[1] U. Kuter, D. Nau, E. Reisner, and R. P. Goldman, "Using classical planners to solve nondeterministic planning problems," *in 18th International Conference on Automated Planning and Scheduling (ICAPS)*, 2008.

[2] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso, "Weak, strong, and strong cyclic planning via symbolic model checking," *Artif. Intell.*, vol. 147, pp. 35-84, 2003.

[3] P. Kissmann and S. Edelkamp, "Solving Fully-Observable Non-deterministic Planning Problems via Translation into a General Game," *in KI 2009: Advances in Artificial Intelligence*. vol. 5803, B. Mertsching, et al., Eds., ed: Springer Berlin Heidelberg, 2009, pp. 1-8.

[4] J. Fu, V. Ng, F. B. Bastani, and I.-L. Yen, "Simple and fast strong cyclic planning for fully-observable nondeterministic planning problems," *in Proceedings of the Twenty-Second international joint conference on Artificial Intelligence* - Volume Three, Barcelona, Catalonia, Spain, 2011.

[5] C. J. Muise, S. A. McIlraith, and J. C. Beck, "Improved Non-Deterministic Planning by Exploiting State Relevance," *in ICAPS*, 2012.

[6] D. Bryce and O. Buffet, "International Planning Competition Uncertainty Part: Benchmarks and Results," *in Proceedings of International Planning Competition*, 2008.

[7] D. Bryce and O. Buffet, "6th International Planning Competition: Uncertainty Part," *in Proceedings of International Planning Competition*, 2008.

[8] R. Tarjan, "Depth-first search and linear graph algorithms," *in 12th Annual Symposium on Switching and Automata Theory*, 1971, pp. 114-121.

[9] J. Hoffmann and B. Nebel, "The FF Planning System: Fast Plan Generation Through Heuristic Search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253-302, 2001.