International Journal on Artificial Intelligence Tools
Vol. 23, No. 6 (2014) 1460028 (21 pages)
© World Scientific Publishing Company
DOI: 10.1142/S0218213014600288

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

Andres Calderon Jaramillo, Jicheng Fu
Computer Science Department, University of Central Oklahoma
100 North University Drive, Edmond, OK 73034, USA
acalderonjaramillo@uco.edu
jfu@uco.edu

Vincent Ng, Farokh B. Bastani, I-Ling Yen
Computer Science Department, University of Texas at Dallas
800 W. Campbell Road, Richardson, Texas 75080-3021, USA
vince@hlt.utdallas.edu
bastani@utdallas.edu
ilyen@utdallas.edu

Received 1 May 2014
Accepted 26 September 2014
Published 26 December 2014

Recently, the state-of-the-art AI planners have significantly improved planning efficiency on Fully Observable Nondeterministic planning (FOND) problems with strong cyclic solutions. These strong cyclic solutions are guaranteed to achieve the goal if they terminate, implying that there is a possibility that they may run into indefinite loops. In contrast, strong solutions are guaranteed to achieve the goal, but few planners can effectively handle FOND problems with strong solutions. In this study, we aim to address this difficult, yet under-investigated class of planning problems: FOND planning problems with strong solutions. We present a planner that employs a new data structure, MRDAG (multi-root directed acyclic graph), to define how the solution space should be expanded. Based on the characteristics of MRDAG, we develop heuristics to ensure planning towards the relevant search direction and design optimizations to prune the search space to further improve planning efficiency. We perform extensive experiments to evaluate MRDAG, the heuristics, and the optimizations for pruning the search space. 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 the state-of-the-art planners, MBP and Gamer, while demonstrating significantly better scalability.

Keywords: Fully observable nondeterministic (FOND) planning; strong cyclic planning, strong planning.

1. Introduction

Fully-observable nondeterministic (FOND) planning is an important and challenging research area.[1,2] To effectively address nondeterministic planning problems, Cimatti *et al.*[3]

classified planning solutions into three categories: *weak* solutions have a probability to achieve the goal; *strong* solutions are guaranteed to achieve the goal; and *strong-cyclic* solutions may terminate and if they do, they are guaranteed to achieve the goal.[3] Thus, strong solutions, if they exist, 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 best-known are arguably MBP and Gamer.[4] 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 larger problems.

The goal in this paper is to present 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[2] and PRP[5] so that they can 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 outperform 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 strong 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 a cycle is formed due to applying action a to state s. To break the cycle, state s should choose a different action to expand the search space if it has more than one applicable action. In case that state s only has one applicable action, 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 from 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 continues, 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 three 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. Third, we prune the search space with five optimizations based on the characteristics of MRDAG to further improve the planning efficiency.

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. Therefore, our planner has achieved the state-of-the-art performance on FOND planning problems with strong solutions.

2. Nondeterministic Planning

We introduce the definitions and notation in nondeterministic planning that will be used 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 nondeterministic 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.

3. Multi-Root Directed Acyclic Graph (MRDAG)

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. All the nodes in Fig. 1 represent states involved in 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 Fig. 1 are the states in the rootset of a MRDAG. Except for the initial state s_0, a state is in a rootset if and only if 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

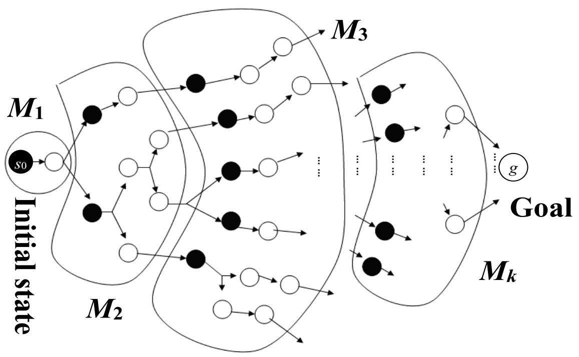

Fig. 1. Solution expansion with MRDAGs.

the states to be expanded in the same rootset? As we will see, heuristics will be used to impose these orderings.

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 there exists $(s', a') \in \pi_M$ such that $s \in \gamma(s', a')$ and one of the following two conditions is satisfied:

(1) s is a goal;
(2) s is not a goal, $|A(s)| > 1$ and s does not belong to M or 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 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 $\pi = \pi_{M1} \cup \pi_{M2} \cup \cdots \cup \pi_{Mn}$, where $\pi_{M1}, \pi_{M2}, \ldots, \pi_{Mn}$ are the policies of a sequence of feasible 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.

4. Strong Planning Algorithm

4.1. *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

```
Global Variables: $\pi$, $\langle s_0, g, \Sigma\rangle$
Function STRONG_PLANNING
1.  $R \leftarrow \{s_0\}$; $\pi \leftarrow \phi$          /*$R$ is the rootset of the MRDAG*/
2.  while $R \neq \phi$ do
3.      $\pi_M \leftarrow$ GET-NEXT-SET-OF-ACTIONS($R$)
4.      if $\pi_M = \phi$ then
5.          if $R = \{s_0\}$ then return FAILURE else
6.              BACKTRACK($R$)
7.          endif
8.      else
9.          if BUILD-MRDAG($\pi_M$) <> FAILURE then
10.             $\pi \leftarrow \pi \cup \pi_M$
11.             if All-GOAL-OUTSIDERS($R$, $\pi_M$) then
12.                 return $\pi$
13.             else
14.                 $R \leftarrow$ GET-OUTSIDERS($R$, $\pi_M$)
15.             endif
16.         endif
17.     endif
18. endwhile
```

Fig. 2. Outline of the strong planning algorithm.

```
Function BUILD-MRDAG ($\pi_M$)
1. $\pi_{root} \leftarrow \pi_M$
2. foreach $(s, a) \in (\pi_{root})$ do
3.     if EXPAND-MRDAG($\pi_M$, $s$, $a$) = FAILURE then
4.         return FAILURE
5.     endif
6. endfor
7. return SUCCESS
```

Fig. 3. Algorithm for building a feasible MRDAG.

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 rootset 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.

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.

4.2. *Building a feasible MRDAG*

EXPAND-MRDAG is shown in Fig. 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

```
Function EXPAND-MRDAG ($\pi_M$, $s$, $a$)
1.  foreach $s' \in \gamma(s, a)$ & $NOT\text{-}GOAL(s')$ do
2.      if $s' \in S_\pi$ or $s' \in S_{\pi M}$ then
3.          if $DETECT\text{-}CYCLE(\pi \cup \pi_M)$ = TRUE then
4.              return FAILURE
5.          endif
6.      elseif $|A(s')| = 1$ then
7.          $\pi_M \leftarrow \pi_M \cup \{(s', a')\}$ with $a' \in A(s')$
8.          if $EXPAND\text{-}MRDAG$ ($\pi_M$, $s'$, $a'$) = 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
```

Fig. 4. Helper function for building a feasible MRDAG.

formed in the graph represented by the union of π and π_M as a result of applying a to s (line 3).[a] 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.

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 because it has more than one applicable action (see Definition 6).

4.3. *An illustrative example*

To better understand our strong planning algorithm, we apply it to the following simplified blocksworld problem (see Fig. 5), which will serve as our running example. In this problem, three actions are possible: the deterministic action *put-down*(X) and two nondeterministic actions, *pick-up*(X, Y) and *put-on*(X, Y). Here, X and Y are variables that represent block A, B, or C. *put-down*(X) puts block X onto the table. The nondeterministic action *pick-up*(X, Y) can pick up block X from the top of block Y. However, it may possibly drop block X onto the table due to the mechanical constraint. The nondeterministic

[a] 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.

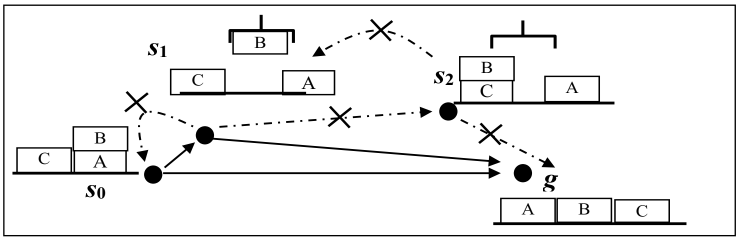

Fig. 5. Blocksworld example.

action *put-on*(X, Y) can put the held block X onto block Y. Similarly, the held block X may fall onto the table. The aim is to move the blocks so that goal g can be reached from the initial state s_0.

The algorithm begins by setting the rootset R of the first MRDAG to $\{s_0\}$ (line 1 in Fig. 2). Next, it computes the policy π_M based on R (line 3 in Fig. 2). Since *pick-up*(B, A) is the only applicable action to s_0, π_M is set to $\{(s_0, \textit{pick-up}(\text{B, A}))\}$. Since π_M is not empty, the algorithm attempts to build a feasible MRDAG by invoking *BUILD-MRDAG* (line 9 Fig. 2). Subsequently, *BUILD-MRDAG* invokes *EXPAND-MRDAG* to recursively expand the MRDAG (line 3 in Fig. 3). Applying *pick-up*(B, A) to s_0 results in two states (line 1 in Fig. 4). One is the goal since block B may fall onto the table. The other is state s_1 in Fig. 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 $\pi_{M1} = \{(s_0, \textit{pick-up}(\text{B, A}))\}$, and s_1 is the non-goal outsider. The control of the algorithm returns to line 9 of Fig. 2. The solution π is updated to be $\pi \leftarrow \pi \cup \pi_{M1} = \{(s_0, \textit{pick-up}(\text{B, A}))\}$ (line 10 of Fig. 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 Fig. 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 Fig. 2). It then invokes *BUILD-MRDAG* with the argument $\pi_M = \{(s_1, \textit{put-on}(\text{B, C}))\}$ (line 9 of Fig. 2). Subsequently, *BUILD-MRDAG* invokes *EXPAND-MRDAG* to recursively expand the MRDAG (lines 1–3 of Fig. 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 Fig. 5 (line 1 in Fig. 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 Fig. 4). Now, $\pi_M = \{(s_1, \textit{put-on}(\text{B, C})), (s_2, \textit{pick-up}(\text{B, C}))\}$. 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 Fig. 4) and then finds that a cycle has been formed between s_1 and s_2 (line 3 of Fig. 4). Hence, *EXPAND-MRDAG* returns failure (line 4 in Fig. 4). Subsequently, *BUILD-MRDAG* also returns failure (line 4 in Fig. 3). Hence, policy $\pi = (s_0, \textit{pick-up}(\text{B, A}))$ is not updated. Then, the strong planning algorithm (Fig. 2) selects another action, say *put-down*(B), for $R = \{s_1\}$, and invokes *BUILD-MRDAG* (line 9 in

Fig. 2) with $\pi_M = \{(s_1, put\text{-}down(\mathrm{B}))\}$. *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 Fig. 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 Fig. 4) to *BUILD-MRDAG* and then to the strong planning algorithm in Fig. 2. The current MRDAG M_2 includes the policy $\pi_M = \{(s_1, put\text{-}down(\mathrm{B}))\}$. Then, policy π is updated by $\pi \leftarrow \pi \cup \pi_M$, i.e., $\pi = \{(s_0, pick\text{-}up(\mathrm{B}, \mathrm{A}))\} \cup \{(s_1, put\text{-}down(\mathrm{B}))\} = \{(s_0, pick\text{-}up(\mathrm{B}, \mathrm{A})), (s_1, put\text{-}down(\mathrm{B}))\}$ (line 10 of Fig. 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).

5. Heuristics

When using MRDAG to expand the solution space, we need to answer two questions. 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 Fig. 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} $(1 \le i \le k)$ in S_{Mr}, let $A_i = (a_{i1}, a_{i2}, \ldots, a_{i\langle mi\rangle})$ be the list of applicable actions to s_{ri} and $\langle mi\rangle = |A(s_{ri})|$ 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., $a_{i1}, a_{i2}, \ldots, a_{i\langle mi\rangle}$. Then, *GET-NEXT-SET-OF-ACTIONS* returns $\pi_M = \{(s_{r1}, \boldsymbol{a}_{11}), (s_{r2}, a_{21}), \ldots, (s_{rk}, a_{k1})\}$ first. If it does not result in a feasible MRDAG, the function will try s_{r1}'s next action a_{12} and return $\{(s_{r1}, \boldsymbol{a}_{12}), (s_{r2}, a_{21}), \ldots, (s_{rk}, a_{k1})\}$, then $\{(s_{r1}, \boldsymbol{a}_{13}), (s_{r2}, a_{21}), \ldots, (s_{rk}, a_{k1})\}$, …, and finally $\{(s_{r1}, \boldsymbol{a}_{1\langle m1\rangle}), (s_{r2}, a_{21}) ,\ldots, (s_{rk}, a_{k1})\}$. If still no feasible MRDAG can be built, the function will try s_{r2}'s next action a_{22} and return $\{(s_{r1}, \boldsymbol{a}_{11}), (s_{r2}, \boldsymbol{a}_{22}), \ldots, (s_{rk}, a_{k1})\}$. If this combination does not lead to a feasible MRDAG, then the function will return $\{(s_{r1}, \boldsymbol{a}_{12}), (s_{r2}, \boldsymbol{a}_{22}), \ldots, (s_{rk}, a_{k1})\}$, $\{(s_{r1}, \boldsymbol{a}_{13}), (s_{r2}, \boldsymbol{a}_{22}), \ldots, (s_{rk}, a_{k1})\}$, …, and finally $\{(s_{r1}, a_{1\langle m1\rangle}), (sr_2, a_{2\langle m2\rangle}), \ldots, (s_{rk}, a_{k\langle mk\rangle})\}$. 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 $s_{ri} \in S_{Mr} = \{s_{r1}, s_{r2}, \ldots, s_{rk}\}$ $(1 \le i \le k)$, 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 fewer unintended effects. If a tie still exists, then it is broken arbitrarily. 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. 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. 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 Fig. 4) or cycles (see lines 3 and 4 of Fig. 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 Fig. 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 Fig. 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 $i = 1, 2, 3, \ldots, n-1$ 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 rootset to expand a MRDAG. Hence, M_1 will be created based on the root s_0 and action a. By induction, it will create $M_2, \ldots, M_n$, which is a solution to the planning problem. Hence, we obtain a contradiction. □

6. Pruning the Search Space

During the planning process, some state-action pairs always lead to cycles or dead-ends and hence result in infeasible MRDAGs. If such state-action pairs are not marked up, the planning algorithm may repeatedly try to use them to expand MRDAGs, which may negatively impact efficiency. It is therefore desirable to permanently disable such state-action pairs so that our planning algorithm will not attempt to use them to expand the search space. For our purpose, disabling a state-action pair (s, a) amounts to modifying the state-transition function γ in Definition 1, i.e., making action a inapplicable to state s. Note that when disabling some state-action pairs, we need to ensure that the resulting planning problem is strongly-equivalent to the original planning problem as defined in Definition 11.

Definition 11. A planning problem $\langle s_0', g', \Sigma' \rangle$ is *strongly-equivalent* to problem $\langle s_0, g, \Sigma \rangle$ if $s_0' = s_0$, $g' = g$, and the set of strong solutions for $\langle s_0', g', \Sigma' \rangle$ is equal to the set of strong solutions for $\langle s_0, g, \Sigma \rangle$.

Theorem 3. Our planning algorithm remains sound and complete if the resulting planning problem is strongly-equivalent to the original planning problem by disabling some state-action pairs.

Proof. The definition of MRDAG and its expansion property ensure that if a solution is found, it will be connected, acyclic, and dead-end free. Hence, soundness is maintained. In addition, since the resulting planning problem is strongly-equivalent to the original planning problem, only those state-action pairs that are irrelevant to strong solutions are disabled. Hence, the expansion of MRDAGs will be the same as it does for the original planning problem except that the expansion is conducted in a smaller, pruned search space. Hence, completeness is maintained. □

To prune the search space, we modify our algorithm to allow state-action pairs to be disabled in the function *EXPAND-MRDAG* before it returns FAILURE (see Fig. 4). We employ two constraints for disabling state-action pairs:

(1) We only disable state-action pairs that belong to the current MRDAG under expansion. This is to ensure that state-action pairs in the ancestry MRDAGs are not affected.
(2) The resulting planning problem $\langle s_0, g, \Sigma' \rangle$ must be strongly-equivalent to the original planning problem $\langle s_0, g, \Sigma \rangle$, where Σ' is the same as domain Σ except that the state-transition function is modified to account for the disabled state-action pairs.

Given the above discussion, we propose five optimizations that can help prune the search space.

Optimization 1. Suppose that a MRDAG $M = \{S_{Mr}, \pi_M\}$ is currently under expansion. During the execution of *EXPAND-MRDAG* (see Fig. 4), a dead-end s is found (see line 11 in Fig. 4). Let $S_{ps} = \{s_p \mid s \in \gamma(s_p, \pi_M(s_p))\}$ be the set of parent states of s under the current policy π_M for the MRDAG M. For each s_p in S_{ps}, we permanently disable the state-action pair $(s_p, \pi_M(s_p))$. Then, for each s_p, we recursively apply the above procedure until it reaches a state s_r in the rootset S_{Mr} such that $(s_r, \pi_M(s_r))$ is also disabled.

Theorem 4. Our planning algorithm remains sound and complete with Optimization 1.

Proof. Optimization 1 only disables state-action pairs that lead to a dead-end. Since no strong solutions can include dead-ends, the planning problem that resulted from disabling such state-action pairs is strongly-equivalent to the original planning problem. According to Theorem 3, our planning algorithm is sound and complete. □

Optimization 2. Suppose that a MRDAG $M = \{S_{Mr}, \pi_M\}$ is currently under expansion. During the execution of *EXPAND-MRDAG* (see Fig. 4), if the addition of a state-action pair (s, a) would create a cycle consisting of states that are not part of any rootsets, then we can treat s as a dead-end and apply the procedure specified in Optimization 1.

Theorem 5. Our planning algorithm remains sound and complete with Optimization 2.

Proof. Without loss of generality, Fig. 6 illustrates a dead-end cycle discussed in Optimization 2. The rootset consists of states s_{r1}, s_{r2}, …, s_{rk}. Applying action a to state s creates a cycle in which no states belong to any rootsets. According to Definition 5, states that are not in the rootset only have one applicable action. Hence, all the states involved in the cycle have only one applicable action. It implies that these states will inevitably get involved in the cycle since they do not have other actions leading to other paths. Hence, any states on the cycle cannot be in any strong solution to the original planning problem. Therefore, the planning problem that resulted from the use of Optimization 2 is strongly-equivalent to the original planning problem. Hence, according to Theorem 3, our planning algorithm is sound and complete. □

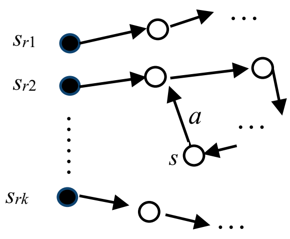

Fig. 6. Illustration of Optimization 2.

Optimization 3. Suppose that a MRDAG $M = \{S_{Mr}, \pi_M\}$ is currently under expansion. During the execution of *EXPAND-MRDAG* (see Fig. 4), if the addition of a state-action pair (s, a) forms a cycle in which only one of its states s_r belongs to a rootset, then we can permanently disable the state-action pair (s_r, $\pi_M(s_r)$).

Theorem 6. Our planning algorithm remains sound and complete with Optimization 3.

Proof. Without loss of generality, Fig. 7 shows the situation specified in Optimization 3. State s_r is the only state that belongs to a rootset. In fact, state s_r must belong to the rootset of the MRDAG currently being expanded, i.e., $M = \{S_{Mr}, \pi_M\}$. Otherwise, suppose that state s_r belongs to a rootset in an ancestry MRDAG M'. The MRDAG expansion property specifies that the expansion of M' stops when it reaches the rootset of its child MRDAG. It

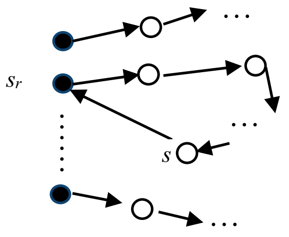

Fig. 7. Illustration of Optimization 3.

implies that the portion of the cycle that belongs to M' will lead to a state in its child MRDAG's rootset. Hence, it contradicts the condition that there is only one state in the cycle belonging to a rootset.

In addition, all other states in the cycle only have a single applicable action as they do not belong to any rootsets (see Definition 5). This implies that after applying action $\pi_M(s_r)$ to s_r, the further expansion will inevitably form the cycle as the rest of the states on the cycle only have one applicable action. Hence, $(s_r, \pi_M(s_r))$ cannot be part of any strong solution to the original planning problem. The planning problem that resulted from disabling $(s_r, \pi_M(s_r))$ is strongly-equivalent to the original planning problem. According to Theorem 3, our planning algorithm is sound and complete. □

Optimization 4. Suppose that a MRDAG $M = \{S_{Mr}, \pi_M\}$ is currently under expansion. During the execution of *EXPAND-MRDAG*, it is found that the addition of a state-action pair (s, a) forms a cycle. Furthermore, this cycle contains more than one state belonging to some rootsets. Among such states, exactly one state s_r belongs to the rootset of the current MRDAG M. Then, we can temporarily disable the action a_r currently assigned to s_r in π_M. Temporarily disabling (s_r, a_r) means that *GET-NEXT-SET-OF-ACTIONS* (see line 3 in Fig. 2) will not assign action a_r to state s_r again. Later, if backtracking occurs (line 6 in Fig. 2), the state-action pair (s_r, a_r) is re-enabled.

Theorem 7. Our planning algorithm remains sound and complete with Optimization 4.

Proof. Without loss of generality, Fig. 8 illustrates a situation specified in Optimization 4. In Fig. 8, the solid dots represent the states in the rootsets. We show that whenever the state-action pair (s_r, a_r) is applied, a cycle will be formed. Note that when expanding the current MRDAG $M = \{S_{Mr}, \pi_M\}$ by applying action a_r to state s_r, the rest of the expansion inside MRDAG M only includes states with one applicable action. Hence, there are no variations. Then, a state transition leads the cycle to another state s_r' in the rootset of MRDAG $M' = \{S_{Mr}', \pi_M'\}$. Since the policy $\pi_{M'}$ remains unchanged when MRDAG M expands (see Constraint 1 discussed previously), the portion of the cycle belonging to M' remains unchanged. Hence, the cycle will be formed whenever the state-action pair (s_r, a_r) is applied. Temporally disabling (s_r, a_r) can avoid forming the cycle. Note that it is possible that it is the state s_r' that chooses a wrong action, which in turn leads to this cycle. Hence, it is unsafe to permanently disable (s_r, a_r). □

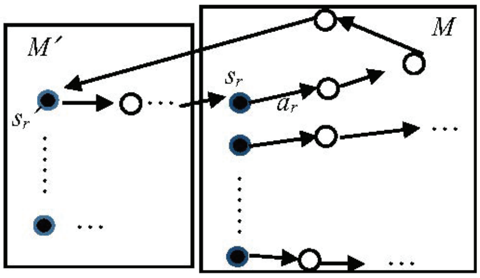

Fig. 8. Illustration of Optimization 4.

Optimization 5. If a state-action pair (s, a) leads to a self-loop (one-state cycle), it is safe to permanently disable (s, a).

Proof. Without loss of generality, Fig. 9 illustrates an example of a self-loop. Obviously, such a state-action pair violates the property of strong solutions and will not be included in any strong solutions to the original planning problem. Hence, the planning problem resulted from disabling (s, a) is strongly-equivalent to the original planning problem. According to Theorem 3, our planning algorithm is sound and complete. □

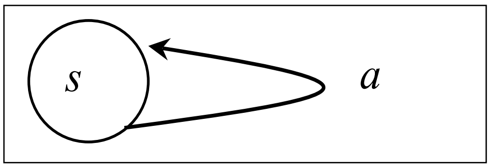

Fig. 9. Illustration of Optimization 5.

7. 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]. In addition, to test how fast a strong algorithm can detect that no strong solutions exist, we also used the strong cyclic blocksworld domain [scbw].

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

Faults: 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: 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 the enhanced 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 domain by changing three non-deterministic 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*" action 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.

7.1. *Planners*

We use MBP and Gamer as baselines. In addition, to determine the contributions made by the two heuristics and five optimizations, we evaluate ten versions of our planners: SP uses both heuristics and all five optimizations, NHO uses none of the heuristics or optimizations, MCS uses only the MCS heuristic without any optimizations, LHD uses only the LHD heuristic without any optimizations, BHU uses both heuristics without any optimizations, OP1 uses optimization 1 with both heuristics, OP2 uses optimization 2 with both heuristics, and so do for OP3, OP4, and OP5. In NHO 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 NHO, if a state has more than one applicable action, one of the actions will be randomly picked to expand the MRDAG. The reason that OP1 to OP5 utilize both heuristics is that the heuristics attempt to minimize the use of randomness and hence the experimental results are repeatable. As a result, the comparison results will be more reliable in reflecting the effectiveness of the optimizations.

7.2. *Problem coverage*

Problem coverage refers to the sum of the number of problems that a planner can solve and the number of problems for which it can detect the non-existence of strong solutions on a particular domain. We set a cutoff time, 1200 seconds, to prevent a planner from running indefinitely when attempting to solve a planning problem. Table 1 shows the experimental results on problem coverage. Note that these results were obtained using a desktop computer with Intel Pentium-4 CPU 3GHz and 1 GB memory. As we can see, when no heuristics or optimizations were employed, the planner NHO performed slightly better than MBP but worse than Gamer. When the heuristic MCS was used, the planner MCS solved slightly more problems than NHO. In comparison, the use of heuristic LHD significantly improved planning performance, i.e. the planner LHD significantly outperformed Gamer and MBP. Besides, the other seven versions of our planner also demonstrated outstanding scalability by solving a significantly larger number of problems than Gamer and MBP. Specifically, when all the heuristics and optimizations were used, the planner, SP, achieved the highest problem coverage, followed by OP4 and OP5. The planner BHU had the same problem coverage as LHD, which suggests that the heuristic LHD played a more critical role than MCS. The use of Optimizations 4 and 5 further improved the problem coverage.

Table 1. Problem coverage.

Domain	Gamer	MBP	SP	NHO	MCS	LHD	BHU	OP1	OP2	OP3	OP4	OP5
scbw (30)	10	10	29	10	10	10	10	10	10	10	10	14
bw (30)	10	0	30	4	7	30	30	30	30	30	30	30
ft (10)	6	4	10	3	3	10	10	10	10	10	10	10
tw (14)	13	0	14	2	2	14	14	14	14	14	14	14
fr (75)	43	12	73	40	40	65	65	65	66	65	73	67
Total (159)	82	26	156	59	62	129	129	129	130	129	137	135

7.3. *CPU time and plan size*

Besides the problem coverage, we also evaluated 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. Table 2 shows the evaluation results on CPU time measured in seconds. 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. "--" indicates that the planner timed out on that problem. As randomness is involved in our planners, 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 "--".

Experimental results indicated that Gamer performed much better than MBP on all the domains. Nevertheless, our SP planner performed significantly better than Gamer. On average, SP was about four orders of magnitude faster than Gamer on strong blocksworld and tireworld, about three orders of magnitude faster than Gamer on faults and first-responders, and two orders of magnitude faster on strong cyclic blocksworld. When comparing the two heuristics, LHD is on average four orders of magnitude faster on faults and three orders of magnitude faster on first-responders than MCS. On the other hand, MCS is about eight times faster than LHD on strong blocksworld domain and two times faster on strong cyclic blocksworld domain. For the optimizations, OP5 performed the best on the strong blocksworld domain, OP1 performed the best on the tireworld domain, and OP4 performed the best on the first-responders and strong cyclic blocksworld domains. In addition, the five optimization planners achieved similar performance on the faults domain.

Table 2. CPU time comparison.

Problem	Gamer	MBP[b]	SP	NHO	MCS	LHD	BHU	OP1	OP2	OP3	OP4	OP5
scbw-1	0.749	153.624	0.002	0.029	0.029	0.044	0.049	0.049	0.049	0.049	0.022	0.002
scbw-2	1.212	227.759	0.001	0.002	0.002	0.009	0.013	0.013	0.013	0.013	0.011	0.001
scbw-3	0.964	172.042	0.003	0.027	0.027	0.046	0.055	0.056	0.056	0.056	0.023	0.003
scbw-6	0.656	73.632	0.002	0.001	0.002	0.010	0.012	0.012	0.012	0.012	0.010	0.002
scbw-8	0.621	61.395	0.003	0.013	0.012	0.031	0.032	0.032	0.031	0.032	0.018	0.003
scbw-9	0.997	230.296	0.002	0.009	0.012	0.024	0.029	0.029	0.029	0.029	0.016	0.002
scbw-10	0.893	231.405	0.003	0.028	0.028	0.050	0.058	0.058	0.057	0.058	0.026	0.003
scbw-20	--	--	0.108	--	--	--	--	--	--	--	--	--
scbw-30	--	--	0.343	--	--	--	--	--	--	--	--	0.336
bw-1	88.331	--	0.003	--	0.003	0.006	0.006	0.007	0.007	0.007	0.006	0.003
bw-2	86.329	--	0.002	0.001	0.001	0.002	0.002	0.003	0.002	0.002	0.002	0.002
bw-3	85.864	--	0.004	--	0.002	0.006	0.007	0.007	0.007	0.007	0.006	0.004
bw-5	87.773	--	0.003	0.001	0.001	0.006	0.006	0.007	0.007	0.007	0.006	0.004
bw-6	86.704	--	0.002	0.002	0.002	0.004	0.004	0.004	0.004	0.004	0.004	0.002
bw-7	87.682	--	0.005	--	0.002	0.011	0.011	0.011	0.011	0.011	0.011	0.005
bw-8	85.861	--	0.004	0.001	--	0.010	0.010	0.011	0.011	0.011	0.011	0.005
bw-9	87.803	--	0.005	--	--	0.010	0.011	0.011	0.011	0.011	0.011	0.004
bw-10	87.958	--	0.003	--	0.002	0.006	0.007	0.007	0.007	0.007	0.006	0.003
bw-20	--	--	0.063	--	--	0.338	0.411	0.414	0.415	0.416	0.338	0.059
bw-30	--	--	0.600	--	--	5.602	6.964	7.608	7.016	7.173	5.493	0.542
ft-6-6	292.925	--	0.011	--	--	0.011	0.011	0.012	0.012	0.012	0.012	0.012
ft-8-8	--	--	0.086	--	--	0.087	0.086	0.088	0.087	0.088	0.088	0.088
ft-9-9	--	--	0.231	--	--	0.234	0.231	0.235	0.235	0.235	0.234	0.236
ft-10-10	--	--	0.611	--	--	0.617	0.609	0.619	0.618	0.619	0.618	0.620
tw-10	231.159	--	0.001	--	--	0.001	0.001	0.001	0.001	0.001	0.001	0.001
tw-11	244.766	--	0.001	--	--	0.001	0.001	0.001	0.001	0.001	0.001	0.001
tw-12	242.073	--	0.001	--	--	0.001	0.001	0.001	0.001	0.001	0.001	0.001
tw-14	94.501	--	0.010	--	--	0.027	0.357	0.014	0.054	0.038	1.506	1.933
fr-1-8	9.954	55.367	0.002	--	10.056	0.003	0.002	0.003	0.003	0.002	0.003	0.002
fr-1-9	53.167	295.584	0.003	--	--	0.003	0.004	0.004	0.004	0.003	0.003	0.003
fr-1-10	690.423	--	0.004	--	--	0.005	0.004	0.004	0.005	0.004	0.005	0.004
fr-10-1	0.755	--	0.011	0.046	0.133	0.010	0.011	0.010	0.012	0.012	0.012	0.012
fr-10-2	--	--	0.012	0.059	0.007	0.018	0.010	0.012	0.011	0.016	0.011	0.011

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

Table 3. Plan size comparison.

Problem	Gamer	SP	NHO	MCS	LHD	BHU	OP1	OP2	OP3	OP4	OP5
bw-1	21	21	--	40	21	21	21	21	21	21	21
bw-2	14	14	29	26	14	14	14	14	14	14	14
bw-3	21	21	--	33	21	21	21	21	21	21	21
bw-5	21	21	26	30	21	21	21	21	21	21	21
bw-6	14	14	28	37	14	14	14	14	14	14	14
bw-7	28	28	--	48	28	28	28	28	28	28	28
bw-8	28	28	37	--	28	28	28	28	28	28	28
bw-9	28	28	--	--	28	28	28	28	28	28	28
bw-10	21	21	--	30	21	21	21	21	21	21	21
bw-20	--	40	--	--	40	40	40	40	40	40	40
bw-30	--	65	--	--	65	65	65	65	65	65	65
ft-6-6	127	127	--	--	127	127	127	127	127	127	127
ft-8-8	--	511	--	--	511	511	511	511	511	511	511
ft-9-9	--	1023	--	--	1023	1023	1023	1023	1023	1023	1023
ft-10-10	--	2047	--	--	2047	2047	2047	2047	2047	2047	2047
tw-10	1	1	--	--	1	1	1	1	1	1	1
tw-11	5	5	--	--	5	5	5	5	5	5	5
tw-12	1	1	--	--	1	1	1	1	1	1	1
tw-14	21	24	--	--	26	37	26	32	34	26	37
fr-1-8	10	10	--	226	10	10	10	10	10	10	10
fr-1-9	11	11	--	--	11	11	11	11	11	11	11
fr-1-10	12	12	--	--	12	12	12	12	12	12	12
fr-10-1	3	3	365	309	3	3	3	3	3	3	3
fr-10-2	--	11	441	87	17	8	10	9	14	9	9

The plan size is associated with the quality of the plans: the smaller the better. Table 3 shows the evaluation results on plan size. SP generated plans with the same size as Gamer on the blocksworld and faults domains. Its plan sizes were also comparable to those generated by Gamer on the tireworld and first-responders domains. On average, it was 1.02 times larger on the tireworld domain and 1.16 times larger on the first-responders domain. In terms of the contributions made by the two heuristics, MCS generated plans with size 2 to 20 times larger than LHD. For the ten versions of our planners, LHD got involved in eight. All eight planners had plan size similar to those of SP. Hence, we can conclude that MCS does not contribute as significantly as LHD to 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.

7.4. *Discussion*

It is not difficult to see that if we remove the constraints that enforce the acyclicity of MRDAG and ensure that no dead-end cycles exist, our planning algorithm will be able to search for strong cyclic solutions if strong solutions do not exist. In fact, the ability to quickly determine the non-existence of strong solutions is important. The planner can be configured to look for strong solutions first, and if it detects that no strong solution exists, it will search for strong cyclic solutions. Our experimental results on the strong cyclic blocksworld domain showed that our strong planning algorithm could quickly determine the non-existence of strong solutions. In contrast, Gamer and MBP could only determine the non-existence of strong solutions for the first 10 planning problems within the cutoff time. As the discussion of strong cyclic planning is beyond the scope of this paper, we make our planner as well as the experimental results on strong cyclic planning publicly available (see http://cs2.uco.edu/~fu/research/MRDAG for details) so that interested researchers can benefit from our algorithm.

8. Conclusion

In this paper, we presented a strong algorithm for FOND planning problems based on a novel data structure, MRDAG (multi-root directed acyclic graph). A MRDAG defines how the solution space is expanded while maintaining acyclicity throughout the planning process. We equip a MRDAG with two heuristics, MCS and LHD, which define

(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.

To further improve planning efficiency, we prune the search space by using five optimizations. We conducted extensive experiments to evaluate the contributions made by the heuristics and optimizations. Experimental results on four domains showed that (1) the use of MRDAG indeed made cycle handling easier and more efficient; (2) the use of the LHD heuristic significantly improved planning performance; and (3) different optimizations are suitable under different situations, and when all the heuristics and optimizations were combined together, our planning algorithm achieved the best performance. Most importantly, our planner significantly outperformed two state-of-the-art planners, Gamer and MBP: on average it ran more than three orders of magnitude faster than Gamer and MBP and solved a significantly larger number of problems.

Acknowledgments

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

References

1. U. Kuter *et al.*, Using classical planners to solve nondeterministic planning problems, in *18th Int. Conf. on Automated Planning and Scheduling* (*ICAPS*) (2008).
2. J. Fu *et al.*, Simple and fast strong cyclic planning for fully-observable nondeterministic planning problems, in *Proc. of the 22nd Int. Joint Conf. on Artificial Intelligence*, Vol. 3 (AAAI Press: Barcelona, Catalonia, Spain, 2011), pp. 1949–1954.
3. A. Cimatti *et al.*, Weak, strong, and strong cyclic planning via symbolic model checking, *Artif. Intell.* **147**(1–2) (2003) 35–84.
4. 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*, B. Mertsching, M. Hund and Z. Aziz (eds.) (Springer Berlin Heidelberg, 2009), pp. 1–8.

5. C. J. Muise, S. A. McIlraith and J. C. Beck, Improved non-deterministic planning by exploiting state relevance, in *ICAPS*, L. McCluskey *et al*. (eds.) (AAAI, 2012).
6. D. Bryce and O. Buffet, International planning competition uncertainty part: Benchmarks and results, in *Proc. of Int. Planning Competition* (2008).
7. D. Bryce and O. Buffet, 6th international planning competition: Uncertainty part, in *Proc. of Int. Planning Competition* (2008).
8. R. Tarjan, Switching and automata theory, in *12th Ann. Symp. on Depth-First Search and Linear Graph Algorithms* (1971).
9. J. Hoffmann and B. Nebel, The FF planning system: Fast plan generation through heuristic search, *Journal of Artificial Intelligence Research* **14** (2001) 253–302.