Depth-First Memory-Limited AND/OR Search and Unsolvability in Cyclic Search Spaces

Akihiro Kishimoto, Adi Botea and Radu Marinescu
IBM Research, Ireland
{akihirok, adibotea, radu.marinescu}@ie.ibm.com

Abstract
Computing cycle-free solutions in cyclic AND/OR search spaces is an important AI problem. Previous work on optimal depth-first search strongly assumes the use of consistent heuristics, the need to keep all examined states in a transposition table, and the existence of solutions. We give a new theoretical analysis under relaxed assumptions where previous results no longer hold. We then present a generic approach to proving unsolvability, and apply it to RBFAOO and BLDFS, two state-of-the-art algorithms. We demonstrate the performance in domain-independent nondeterministic planning.

1 Introduction
When cost-optimal heuristic search cannot memorize the examined search space effectively, depth-first search (DFS) is often preferred to best-first search (BFS), e.g., [Korf, 1985]. Depth-first memory-limited search (DFMLS) combines DFS with a transposition table (TT). DFMLS thus caches previous search results and reuses them for the nodes revisited during search [Plaat et al., 1996; Reinefeld and Marsland, 1994]. When the TT is full and new results need to be saved, existing TT entries are replaced with new entries [Nagai, 1999].

Bonet and Geffner’s Bounded Learning DFS (BLDFS) algorithm (2005a; 2005b) keeps revising lowerbounds of the optimal solution by a series of depth-first searches limited by a threshold that is explicitly set at the root, until finding the optimal solution. During search, the revised lowerbounds are saved in the TT. BLDFS often outperforms BFS algorithms such as AO* [Nilsson, 1980] and CFCREV [Jiménez and Torras, 2000]. Although BLDFS can be regarded as DFMLS, its theoretical properties assume monotonic lowerbounds in the TT [Bonet and Geffner, 2005b]; the TT requires unlimited capacity with consistent heuristics in use. This assumption does not hold when the search space is larger than the TT and TT entries need to be replaced. In this case, the effect is equivalent to using an inconsistent heuristic.

RBFAOO [Kishimoto and Marinescu, 2014] is a depth-first memory-limited AND/OR search algorithm based on ideas of Recursive Best-First Search (RBFS) [Korf, 1993] with the TT. It was shown to be complete (with respect to solvable or unsolvable instances) if the search space is a finite DAG.

We focus on optimal DFMLS on finite cyclic AND/OR search spaces where a solution must be cycle-free, assuming an additive state model as well as inconsistent heuristics and/or a limited size TT. First, we outline new challenging issues that arise under these relaxed assumptions, such as returning suboptimal solutions due to the TT and cycles. These issues are not considered in the previous work of Bonet and Geffner (2005a; 2005b) and Kishimoto and Marinescu (2014). Akagi et al. (2010) raise similar issues but analyzed them for OR search. We then develop a generic approach to prove unsolvability with cycles, and apply it to RBFAOO and a BLDFS variant. Eriksson et al. (2017; 2018a; 2018b) have studied proving unsolvability, but their approach is specific to OR search.

Furthermore, we present theoretical properties of our approach and show that, if a solution exists, then an optimal solution cost can eventually be found with limited memory. In addition, our approach can also prove the unsolvability of some instances in a cyclic search space but, in some other unsolvable instances, the technique may fail due to an infinite loop. While Akagi et al. (2010) consider the same assumptions, their theoretical results are for IDA*+TT, which is OR search. Our new theoretical results hold for two algorithms of different behaviors in more general scenarios than those of Akagi et al. (2010) and Bonet and Geffner (2005a; 2005b). Moreover, as a corollary, our analysis additionally includes RBFS+TT, which is a different OR search compared to IDA*+TT.

Finally, we validate our theoretical analysis with experiments in domain-independent nondeterministic planning, showing that there are instances that require our approach to prove unsolvability, and that the ability to find optimal solution costs is preserved for solvable instances.

2 Preliminaries
We consider a state model with fully-observable states and nondeterministic actions. A tuple \( \langle S, A, s_0, S_T \rangle \) defines the problem space, where \( S \) is a finite set of states, \( s_0 \in S \) is an initial state, \( A \) is a finite set of actions, and \( S_T \subseteq S \) is a set of terminal states. Unlike Bonet and Geffner (2005a), we allow the case where \( S_T \) is empty, for instances with no solution. \( A(s) \subseteq A \) denotes a set of applicable actions to a non-terminal state \( s \). Each action \( a \in A(s) \) returns a set of successor states denoted by \( F_{s,a} \subseteq S \). If \( F_{s,a} \) has more than
one state, action \( a \) is nondeterministic. Otherwise, \( a \) is deterministic. The cost of a transition from state \( s \) with action \( a \), denoted by \( c(s, a) \), is assumed \( c(s, a) > 0 \) and discretized to an integer for any \( a \) and \( s \). For simplicity, the costs for a terminal state (aka a goal in [Bonet and Geffner, 2005a]) and a deadend state are 0 and \( \infty \), respectively. Our results can easily be generalized for terminal states with non-negative costs.

Similarly to Bonet and Geffner (2005a), we map this state model into an AND/OR graph. An OR node is labeled by a state \( s \). An AND node is labeled by the set of states reached by an action \( a \), namely \( F_{s,a} \). The root of \( G \) is an OR node labeled by the initial state \( s_0 \). The children of an OR node \( s \) are AND nodes corresponding to the applicable actions \( A(s) \) in state \( s \), and the children of an AND node \( F_{s,a} \) are the OR nodes corresponding to the states in \( F_{s,a} \). The edge cost from OR node \( s \) to AND node \( F_{s,a} \) is \( c(a, s) \), while the edge cost from an AND node to an OR node is 0.

**Definition 1** (solution tree). *Given an AND/OR graph \( G \), a solution tree \( ST \) is a subtree of \( G \) such that: (1) the root node of \( ST \) is the initial state \( s_0 \), (2) for each internal OR node \( n \) in \( ST \), one of \( n \)'s children is in \( ST \), (3) for each internal AND node \( n \) in \( ST \), all of \( n \)'s children are in \( ST \), and (4) all tip nodes in \( ST \) are terminal nodes (goals).*

While Bonet and Geffner (2005a) introduce BLDFS for the max state model (MAX), we consider here the additive state model (ADD) of Bonet and Geffner (2005a) where the cost of a solution tree \( ST \) is defined as the sum of the edge costs in \( ST \). An optimal solution has a minimal cost.

We can support the MAX with small changes. Yet, focusing on the ADD in our setting is essential, because RBFAOO and BLDFS face new challenges discussed in Section 3.

Each node \( n \) in \( ST \) is associated with a value \( V(n) \) capturing the optimal solution cost of the subtree rooted at \( n \). \( V(n) \) can be computed recursively based on the values of \( n \)'s successors: OR nodes by minimization, AND nodes by summation. The ADD model has applications such as the (dis)assembly domain [Jiménez and Torras, 2000] and web service composition [Liang and Su, 2005]. Chemical synthesis planning [Heifets and Jurisica, 2012] is a remarkable example, when the objective is minimizing the reaction cost.

Path \( p = s_0, a_0 \rightarrow \cdots \rightarrow s_{k-1}, a_{k-1} \rightarrow s_k \) leads to an AND node \( F_{s_k,a_k} \) from \( s_0 \) by a sequence of actions \( a_0, \cdots, a_{k-1} \). Path \( p = s_0, a_0 \rightarrow \cdots \rightarrow s_k, a_k \) leads to an AND node \( F_{s_k,a_k} \) from \( s_0 \) by a sequence of actions \( a_0, \cdots, a_k \). Path \( p + a \) indicates a path that extends \( p \) with action \( a \) leading to an AND node \( F_{n,a} \), where \( n \) is an OR node reached by path \( p \). Analogously, \( p + s \) extends path \( p \) by adding an OR child of an AND node reached by path \( p \).

A cyclic path has \( s_i = s_j \) for some \( i \neq j \). The search space can contain cyclic paths, but a solution tree has no cyclic path. If no cycle-free solution tree exists, the solution cost is \( \infty \).

Since we solve a minimization task, the algorithms we present attempt to improve a lowerbound \( L \) of the optimal solution cost \( C^* \).

When \( L = C^* \), the solution is proven optimal. To prove unsolvability, it is sufficient to ensure that \( L > U \), where \( U \) is an upperbound of the optimal solution. While computing good \( U \) values is an important topic of research, in this paper, we assume that such \( U \) cannot be calculated, and the algorithms need to explicitly prove \( L = \infty \).

We say that an algorithm has an infinite loop if it runs forever without finding \( C^* \) or proving \( L = \infty \). We assume that \( \infty - k < \infty \) holds, where \( k \) is a small number. In the actual implementation, a large finite number is often used to represent \( \infty \). We assume that \( V(n) < \infty - k \) holds if an optimal solution exists, and that no algorithm can gradually increase \( L \) to \( \infty - k \) in finite time if no solution exists.

**RBFAOO and BLDFS**

These algorithms maintain at each node \( n \) a lowerbound \( q(n) \) (called a q-value) on \( V(n) \). During search, they improve and cache in the TT \( q(n) \) which is calculated by backing up the q-values of \( n \)'s children, until proving \( q(r) = V(r) \) at the root \( r \) or \( q(r) = \infty \) (i.e., no solution). For an admissible heuristic \( h, q(n) \) is computed as:

1. \( q(n) = 0 \), if \( n \) is a terminal OR node.
2. \( q(n) = h(n) \), if \( n \) is a non-terminal tip node.
3. \( q(n) = \min_{a \in A(n)}(c(n, a) + q(F_{n,a})) \), if \( n \) is an internal OR node and \( F_{n,a} \) is an AND child of \( n \).
4. \( q(F_{n,a}) = \sum_{s \in F_{n,a}} q(s) \), if \( F_{n,a} \) is an internal AND node and \( s \) is an OR child of \( F_{n,a} \).

Let \( th(n) \) be a threshold at node \( n \). Both algorithms examine a subtree rooted at node \( n \) in a depth-first manner until \( th(n) < q(n) \) holds or \( n \) is solved optimally (termination condition). RBFAOO employs a local threshold controlling mechanism that initially sets \( th(r) = \infty \) at the root \( r \). It updates \( q(n) \) at node \( n \) using q-values of \( n \)'s children. If a termination condition holds, it backtracks to \( n \)'s parent. Otherwise, it selects a child to examine further. For \( q(F_{n,a}) = c(n, a) + q(F_{n,a}) \), let \( q(F_{n,a}) \) and \( q(F_{n,a}) \) be the smallest and second smallest values among the list of values of \( n \)'s children. If \( n \) has only one child, \( q(F_{n,a}) = \infty \).

RBFAOO selects a child as follows:

- At an OR node \( n \), select \( F_{n,a_1} \) with a new threshold \( th(F_{n,a_1}) = \min(th(n), q(F_{n,a_1})) - c(n,a_1) \).
- At an AND node \( F_{n,a} \), select any unsolved child \( s \) with a new threshold \( th(s) = th(F_{n,a}) - q(F_{n,a}) + q(s) \).

BLDFS employs a global threshold controlling scheme that gradually increases the threshold at the root. In each iteration, it sets \( th(r) = q(r) \) at the root \( r \). At an OR node \( n \), if a child \( F_{n,a} \) holds \( q(F_{n,a}) \leq th(n) \), it examines \( F_{n,a} \) with \( th(F_{n,a}) = th(n) - c(n,a) \). At an AND node, it updates the threshold in the same way as RBFAOO, which is also described in [Bonet and Geffner, 2005a].

3 Challenges under Relaxed Assumptions

In the ADD model with a consistent heuristic and unlimited TT capacity, BLDFS can always return an optimal solution in finite time if one exists in the finite cyclic search space. This property is based on the fact that BLDFS does not need to generate an action leading to a repeated state, since it never enters into a cycle [Bonet and Geffner, 2005b]. However, when using an inconsistent heuristic, even with an unlimited
TT, BLDFS may enter into a cycle. If the TT is updated without carefully considering that repeated state, BLDFS may lead to a suboptimal solution.

**Example with Inconsistent Heuristic**

In the example shown in Figure 1(left), both BLDFS and RBFAOO enter into a cycle when using an admissible, inconsistent heuristic whose values are shown in the figure. The optimal solution \( \text{opt} \) is \( A \rightarrow C \rightarrow D \rightarrow B \rightarrow E \rightarrow F \) with solution cost 54. BLDFS sets \( th(A) = h(A) = 50 \). If it does not detect a repeated state, \( th(A) \) is large enough to reach \( B \) again via path \( p_1 = A \rightarrow B \rightarrow D \rightarrow B \). In case of RBFAOO, \( th(B) = q'(C) - c(A, A \rightarrow B) = 40 \). As in the case of BLDFS, \( th(B) \) is large enough to revisit \( B \) via \( p_1 \).

Now assume that BLDFS generates no action forming a cycle. Then, it incorrectly returns a suboptimal solution \( \text{soft} = A \rightarrow B \rightarrow E \rightarrow F \) with cost 62, since it searches in the TT for a cycle. The value of \( D \) is mistakenly saved in the TT for since a cycle is formed. The value of \( D \) reaches a dead-end. It reaches \( B \) again via path \( p_2 = A \rightarrow B \rightarrow D \rightarrow B \).

The pathmax heuristic does not resolve the issue in our example, since it is still an inconsistent heuristic [Holte, 2010].

In the ADD model, Bonet and Geffner (2005b) ensure an algorithmic equivalence between LDWS with no explicit threshold (i.e., use \( q(n) \) as a threshold for node \( n \)) and BLDFS with an explicit threshold. However, the equivalence no longer holds in our relaxed setting. For example, assume that an OR node \( n \) has one AND node \( m \) that has two successors \( s_1 \) and \( s_2 \), where \( h(n) > c(n, m) + h(s_1) + h(s_2) \) due to an inconsistent heuristic. Then, LDWS has an issue of setting up a threshold that is large enough to strictly improve \( q(n) \) or solve \( n \) after LDWS finishes examining \( s_1 \) or \( s_2 \). Hence, the explicit threshold of BLDFS is necessary even in the ADD.

**Example with Consistent Heuristic and Limited-Size TTs**

If existing TT entries are overwritten with the lowerbounds of new nodes, the heuristic estimates in the TT become inconsistent. This is equivalent to using an inconsistent heuristic.

We show an example only for RBFAOO, since an example for BLDFS can be constructed analogously. The heuristic shown in Figure 1(left) becomes consistent by replacing \( h(A) = h(C) = 50 \) with \( h(A) = h(C) = 0 \). In this case, RBFAOO first reaches \( D \) via \( p_3 = A \rightarrow C \rightarrow D \rightarrow B \), and only generates action \( B \rightarrow E \) (i.e. \( B \rightarrow D \) forms a cycle). Then, RBFAOO backtracks to \( A \) and stores improved \( q \)-values, i.e., \( q(B) = 51, q(D) = 52 \), and \( q(C) = 53 \), based on \( h(E) = 50 \). Next, it examines \( B \). Assume now that \( q(B) \) and \( q(D) \) are removed from the TT before \( B \) is examined. Then, it reaches \( D \) via \( p_2 \), since \( th(B) = q'(C) - c(A, A \rightarrow B) = 43 \) is large enough. It stores the value of \( \infty \) in the TT entry of \( D \) since \( D \rightarrow B \) forms a cycle. If the TT never removes \( D \), RBFAOO cannot reach a goal via \( \text{opt} \).

One straightforward way to be able to return an optimal solution with Figure 1 (left) is to perform so-called unfolding cycles (keep examining a node), even if it encounters a repeated state. While this approach returns an optimal solution if one exists, it cannot prove unsolvability of a problem instance that involves cycles such as Figure 1(right).

### 4 Boosting Ability to Prove Unsolvability

We introduce a generic approach to proving unsolvability, even for limited TTs and inconsistent heuristics, and combine it with RBFAOO and BLDFS with an ADD model. The heuristic \( h \) must be admissible, but can be inconsistent.

**Generic Principles**

The suboptimal solution returned in the example from Section 3 is caused by saving in the TT a value of \( \infty \) calculated by a cycle which holds only for that path. We bypass this incorrectness by extending TTs to separately save two types of values: \( q \)-values and \( r \)-values.

Since the \( q \)-value of a node \( n \), \( q(n) \), described in Section 2, is a path-independent lowerbound on the node’s cost, it is retrieved at \( n \) via any path. An \( r \)-value for a node \( n \) reached via a path \( p, r(n, p) \), is a path-specific lowerbound. The value \( r(n, p) \) is retrieved only when \( n \) is reached by path \( p \). Since \( r \)-values take into account that a repeated state is a deadend, \( r(n, p) \) is computed as follows:

1. \( r(n, p) = 0 \), if \( n \) is a terminal OR node and \( p \) does not form a cycle.
2. \( r(n, p) = \infty \), if \( n \) is an OR node and \( p \) forms a cycle.
3. \( r(n, h) = h(n) \), if \( n \) is a non-terminal tip node.
4. \( r(F_{n,a}, p) = \min_{a \in A(n)} (c(n, a) + r(F_{n,a}, p + a)) \), if \( n \) is an internal OR node and \( F_{n,a} \) is an AND child of \( n \).
5. \( r(F_{n,a}, p) = \sum_{s \in F_{n,a}} r(s, p + s) \), if \( F_{n,a} \) is an internal AND node and \( s \) is an OR child of \( F_{n,a} \).

When a search reaches a node \( n \) via a path \( p \), and further explores the space rooted at \( n \), it uses a threshold to control the search underneath. Both \( q(n) \) and \( r(n, p) \) get updated based on the search results. Because of the definition of the state model, a repeated node is always an OR node. If the search expands \( n \) and generates a child \( ch \) that belongs to path \( p \), then \( ch \) is a repeated node. It is a deadend along this path, and not expanded further along this path. When updating \( q(n) \) and \( r(n, p) \), we set \( r(ch, p + a + ch) = \infty \). However, computing \( q(n) \) uses \( q(ch) \), as shown in Section 2, as \( q \) is a path-independent estimation.

When \( n \) is reached via a new path \( p' \), \( q(n) \) is used to reduce duplicate search effort. In addition, since \( q(n) \) is more conservative than \( r(n, p') \), we set \( r(n, p') = q(n) \) if \( r(n, p') <
Algorithm 1 RBFAOO: RBFAOO with our approach

| Require: Root node node
| 1: root:th = \infty - 1
| 2: r = RBFAOOSearch(root)
| 3: return r

$q(n)$. This update is not uncommon, since $r(n, p') = h(n)$ when $n$ is reached via $p'$ for the first time.

Threshold for r-values. Typical DFMLS updates thresholds based on the q-value. In contrast, we adjust a threshold based on the r-value. The node to examine is selected by the smallest r-value at each OR node, and the threshold is updated to observe an improvement of the r-value as an indication of search progress. While Akagi et al. (2010) use this idea, they apply it only to IDA*+TT, and do not store r-values in the TT.

Reuse of proof trees. If a node $n$ is optimally solved, then typical DFMLS marks $n$ as solved and saves it in the TT. When $n$ is reached via any path, $n$ is regarded as solved without re-examining the search space rooted at $n$. However, care must be taken in this approach to ensure that no implicit cycles are created [Campbell, 1985; Kishimoto and Müller, 2004]. For example, assume a node $n$ is saved in the TT as solved, and $n$’s solution tree has a node $c$. If $c$ is removed from the TT, $n$ is reached by a path $p$, and $p$ contains $c$, then regarding $n$ via $p$ as solved is not correct due to a cyclic path $c \cdots \rightarrow n \cdots \rightarrow c$. We avoid implicit cycles by checking both q-values and r-values when marking a node as solved. Assume that a node $n$ via path $p$ is solved or proven a deadend. If $q(n) = r(n, p)$, the TT entry for $q(n)$ marks the flag as solved. This is used as a path-independent solution. If $q(n) \neq r(n, p)$, the TT entry for $r(n, p)$ marks the flag as solved, which is reused only for $n$ via path $p$. Theorem 4 in Section 5 ensures the correctness.

Space saving in the TT. The TT is typically implemented as a hash table, and has a fixed number of entries to store information on nodes. Compared to the TTs which take only q-values, our approach manages q- and r-values of each node. To save the TT entries, if $r(n, p) = q(n)$, only $q(n)$ is saved. Otherwise, in addition to the TT entry for $q(n)$, another TT entry is used to save $r(n, p)$. When $r(n, p)$ is retrieved but no information is available except $q(n)$, $q(n)$ is used for $r(n, p)$.

Combination with RBFAOO

Algorithms 1-4 show a combination of our approach with RBFAOO, denoted by RBFAOO. We omit the overestimation technique [Kishimoto and Marinescu, 2014], as automatically setting an efficient overestimation parameter in cyclic search spaces remains open. The essential differences from original RBFAOO, using the r-values instead of q-values, are highlighted in bold. Algorithm 2 focuses on searching for a solution. Method FromTT (Algorithms 3 and 4) retrieves from TT: $r(n, p)$ where $p$ is a path to reach $n$, $q(n)$, and a flag whether $n$ is solved. If $p$ forms a cycle due to $n$, then set $r(n, p) = \infty$. If the TT has no entry for $q(n)$ or $r(n, p)$, use $h(n)$. Also, FromTT returns $q(n)$ for $r(n, p)$ if $r(n, p) < q(n)$.

Node $n$ is a 3-field tuple: a threshold $th$, a flag $slvd$ and a path $path$ to reach $n$. The flag is true if $n$ is solved or proven unsolvable. A node $n$ also has state information, to enable to retrieve $q(n)$ from the TT. SaveInCache saves in the TT an $r(n, p)$, $q(n)$, and whether $n$ via $p$ is solved.

Examining a node $n$ includes a check whether $n$ has children or not. If not (lines 1–4 in Algorithm 2), the Evaluate method sets $n.slvd = \top$ and checks if $n$ is a terminal or a path-independent deadend (i.e., deadend unrelated to cycles). The q-value is set to 0 for a terminal node and to $r(n)$ if proven a deadend. If $n.th < r(n, p)$ or $n.slvd = \top$, the TT is typically implemented as a hash table, and has a fixed number of entries to store information on nodes. Compared to the TTs which take only q-values, our approach manages q- and r-values of each node. To save the TT entries, if $r(n, p) = q(n)$, only $q(n)$ is saved. Otherwise, in addition to the TT entry for $q(n)$, another TT entry is used to save $r(n, p)$. When $r(n, p)$ is retrieved but no information is available except $q(n)$, $q(n)$ is used for $r(n, p)$.

Combination with RBFAOO

Algorithms 1-4 show a combination of our approach with RBFAOO, denoted by RBFAOO. We omit the overestimation technique [Kishimoto and Marinescu, 2014], as automatically setting an efficient overestimation parameter in cyclic search spaces remains open. The essential differences from original RBFAOO, using the r-values instead of q-values, are highlighted in bold. Algorithm 2 focuses on searching for a solution. Method FromTT (Algorithms 3 and 4) retrieves from TT: $r(n, p)$ where $p$ is a path to reach $n$, $q(n)$, and a flag whether $n$ is solved. If $p$ forms a cycle due to $n$, then set $r(n, p) = \infty$. If the TT has no entry for $q(n)$ or $r(n, p)$, use $h(n)$. Also, FromTT returns $q(n)$ for $r(n, p)$ if $r(n, p) < q(n)$.

Node $n$ is a 3-field tuple: a threshold $th$, a flag $slvd$ and a path $path$ to reach $n$. The flag is true if $n$ is solved or proven unsolvable. A node $n$ also has state information, to enable to retrieve $q(n)$ from the TT. SaveInCache saves in the TT an $r(n, p)$, $q(n)$, and whether $n$ via $p$ is solved.

Examining a node $n$ includes a check whether $n$ has children or not. If not (lines 1–4 in Algorithm 2), the Evaluate method sets $n.slvd = \top$ and checks if $n$ is a terminal or a path-independent deadend (i.e., deadend unrelated to cycles). The q-value is set to 0 for a terminal node and to $r(n)$ if proven a deadend. If $n.th < r(n, p)$ or $n.slvd = \top$, the TT is typically implemented as a hash table, and has a fixed number of entries to store information on nodes. Compared to the TTs which take only q-values, our approach manages q- and r-values of each node. To save the TT entries, if $r(n, p) = q(n)$, only $q(n)$ is saved. Otherwise, in addition to the TT entry for $q(n)$, another TT entry is used to save $r(n, p)$. When $r(n, p)$ is retrieved but no information is available except $q(n)$, $q(n)$ is used for $r(n, p)$.

Combination with RBFAOO

Algorithms 1-4 show a combination of our approach with RBFAOO, denoted by RBFAOO. We omit the overestimation technique [Kishimoto and Marinescu, 2014], as automatically setting an efficient overestimation parameter in cyclic search spaces remains open. The essential differences from original RBFAOO, using the r-values instead of q-values, are highlighted in bold. Algorithm 2 focuses on searching for a solution. Method FromTT (Algorithms 3 and 4) retrieves from TT: $r(n, p)$ where $p$ is a path to reach $n$, $q(n)$, and a flag whether $n$ is solved. If $p$ forms a cycle due to $n$, then set $r(n, p) = \infty$. If the TT has no entry for $q(n)$ or $r(n, p)$, use $h(n)$. Also, FromTT returns $q(n)$ for $r(n, p)$ if $r(n, p) < q(n)$.

Node $n$ is a 3-field tuple: a threshold $th$, a flag $slvd$ and a path $path$ to reach $n$. The flag is true if $n$ is solved or proven
choose any unsolved child s. Then, update \( th(s) \) (lines 12 and 21), and recursively examine \( s \) (lines 13 and 22).

A small yet efficient change to the original RBFAOO is to store \( q(n) \) in the TT, immediately after it is updated (lines 9 and 17). This is necessary to quickly back-propagate a larger \( q \)-value involving repeated states. Bonet and Geffner (2006) use a similar idea for an MDP version of LDFS.

RBFAOO corrects the example shown in Figure 1 left. RBFAOO reaches \( B \), via \( A \to B \), updates \( q(B) = \min(1 + h(E), 1 + h(D)) = 1 \), then reaches \( D \) via \( p_1 = A \to B \to D \). It generates a cycle \( p_1 \to B \), and stores \( r(D, p_1) = \infty \) due to the cycle. On the other hand, it sets \( q(D) = 1 + q(B) = 2 \). It updates \( r(B, A \to B) = \min(1 + r(D, p_1), 1 + r(E, A \to B \to E)) = 51 \), and \( q(B) = \min(1 + q(D), 1 + q(E)) = 3 \). It then selects \( C \) and reaches \( D \) via \( p_2 = A \to C \to D \). Since \( r(D, p_2) = 4 \), which is different from \( r(D, p_1) = \infty \), it does not regard \( D \) via this path as unsolved. Therefore, it reaches \( B \) via \( p_2 \to B \), eventually leading to optimal solution \( p_2 = B + E + F \).

Combination with Bounded LDFS Variant

BLDFS with our approach, denoted \( \text{BLDFS}_r \), controls its threshold by \( r \)-values, based on the approach of Bonet and Geffner (2005a) for the ADD model. While RBFAOO and BLDFS share the same principles, BLDFS examines the space with a global threshold controlling scheme.

Let \( \text{th}(n, p) \) be a threshold for a node \( n \) reached via path \( p \), and \( rt \) be the root. In each iteration, BLDFS starts examining the search space with \( \text{th}(rt, rt) = r(rt, rt) \), and examines \( n \) via \( p \) as long as \( r(n, p) \leq \text{th}(n, p) \). At the start of search, \( \text{th}(rt, rt) = h(rt) \), since \( r(rt, rt) = h(rt) \).

At each internal node, BLDFS selects a child in the same way as described in Section 2 except that it uses the \( r \)-values:

- At an OR node \( n \) via path \( p \), for a child \( F_{n,a} \) via path \( p_F = p + a \), if \( c(n, a) + r(F_{n,a}, p_F) \leq \text{th}(n, p) \) holds, examine \( F_{n,a} \) with \( \text{th}(F_{n,a}, p_F) = \text{th}(n, p) - c(n, a) \).

- At an AND node \( F_{n,a} \) via path \( p_F \), select any unsolved child \( s \) with a new threshold \( \text{th}(s, p_s) = \text{th}(F_{n,a}, p_F) - r(F_{n,a}, p_F) + r(s, p_s) \), where \( p_s = p_F + s \).

The \( q \)- and \( r \)-values are back-propagated to the root in the same way as RBFAOO. For theoretical analysis, we modify slightly the original BLDFS. If a solution is found at node \( n \), our BLDFS variant delays to back up the solved flag of \( n \)’s parent, until the solution cost at \( n \) is proven optimal within the threshold. It is when BLDFS marks the solved flag at the root as solved that the optimal solution cost is proven at the root. Theoretical analysis with the original BLDFS under our relaxed assumptions is left as future work.

Algorithms 5–6 show the pseudocode of \( \text{BLDFS}_r \), which is a combination of BLDFS with our approach. We use notations that are familiar in the heuristic search community. To explain in detail how \( r \)-values and \( q \)-values are managed, our pseudocode is represented in a more detail than in the work of Bonet and Geffner (2005a; 2005b). The essential differences from original BLDFS are highlighted in bold. Node \( n \) includes the threshold \( \text{th} \), called the \( \text{bound} \) in [Bonet and Geffner, 2005a]. In addition, \( n \) contains \( r \) and \( q \), an \( r \)-value and a \( q \)-value stored after \( n \) is examined, and a path \( \text{path} \) to \( n \).
from the root. The Sum method in Algorithm 4 is extended to receive a child $s_i$ and sets $s_i:(slvd$, indicating if $s_i$ is marked as solved in the TT. Sum also updates $n.r, n.q$ and $n.slvd$.

5 Theoretical Properties

We define the notion of memory requirement linear in the search depth and the branching factor along the path.

**Definition 2.** Given a search problem and an amount of memory available, we say that the reasonable memory requirement is satisfied if, for any path that could be explored in the search, the memory is sufficient to store all nodes along the current paths, together with all their siblings.

If $b$ is the maximum branching factor, and $d$ is the maximum depth of a path explored in the problem at hand, we need at least $O(bd)$ memory, as discussed in [Korf, 1993].

We prove theoretical properties about our approach combined with RBFAOO and our BLDFS variant that require only reasonable memory, denoted by RBFAOO$_r$ and BLDFS$_r$, respectively. Our results ensure that our approach can be enhanced with a TT that has any strategies to replace the TT entries of any nodes previously examined, unless these TT entries are for the nodes/their siblings of the path currently examined. For example, assume that a search path $p_1$ changes to a new one $p_2$. Let $n$ be a node on $p_1$ that is neither on $p_2$ nor any of the siblings of the nodes on $p_2$. Then, the q- and r-values for $n$ can be safely removed from the TT.

The reasonable memory requirement is close to the theoretically minimum memory requirement to return an optimal solution cost. Note that the size of the solution tree is needed as a memory requirement to return a solution tree.

Let $C(n, p)$ be an optimal solution cost for node $n$ via path $p$, $T_q(n)$ be $q(n)$ in the TT, and $T_r(n, p)$ be $r(n, p)$ in the TT. If no result is saved, $T_q(n) = h(n)$ where $h$ is an admissible heuristic. So is $T_r(n, p)$, but if $p$ forms a cycle, then $T_r(n, p) = \infty$. Since $T_r(n, p)$ is increased to $T_q(n)$ as discussed in Section 4, clearly $T_q(n) \leq T_r(n, p)$ holds.

**Lemma 1.** $T_r(n, p)$ always has admissible values for any path $p$ leading to node $n$, i.e., $T_r(n, p) \leq C(n, p)$.

**Theorem 2 (Correctness).** The solution cost returned by RBFAOO$/BLDFS_r$ is always optimal.

**Lemma 3.** Assume that RBFAOO$_r$/BLDFS$_r$ marks a node $n$ as always solved, i.e., $n$ via path $p$ is solved and $T_q(n) = T_r(n, p)$ holds. Then, $T_q(n) = C(n, n)$ holds, where $C(n, n)$ is the optimal solution cost of the subtree rooted at $n$.

Theorem 4 guarantees that an optimal solution cost at $n$ can be transposed from one path to another (see Section 4).

**Theorem 4.** Assume that node $n$ via path $p$ is solved and $T_q(n) = T_r(n, p)$. Then, marking $n$ as solved via any path does not affect the optimal solution cost.

Proof. Let $s_0$ be the root. We prove the theorem by showing that a node $n$ marked as always solved is not a part of the optimal solution cost at $s_0$, if $T_q(n)$ causes an implicit cycle. In this case, $T_q(n)$ merely prunes away duplicate search rooted at $n$, even if marked as solved.

Assume that $n$ via $p$ is marked as always solved. Let $ST$ be the (optimal) solution tree rooted at $n$ and $T_q(n) = C(n, n)$.

By Lemma 3 ST has no cyclic path, since the solution cost at each internal OR node $m$ in $ST$ is $C(m, m)$ and $c(m, a) > 0$.

Let $Q(n, p)$ be the set of OR nodes on the non-cyclic path $p$ from $s_0$ to node $n$, and $P(n, p) = Q(n, p) \setminus \{n\}$. The correctness of the theorem is ensured as follows: (1) If $s \not\in ST$ for every $s \in P(n, p)$, $C(p, n) = C(n, n)$. No cycle is created from $s_0$ to reach any node in $ST$. (2) Otherwise, a node $s \in P(n, p)$ may be in $ST$. That is, $ST$ cannot be used to calculate $C(n, p)$, since $s$ creates a cycle. Let $m \in P(n, p)$ be the closest node to $s_0$ on $p$ which creates a cycle in $ST$. Let $m = \text{node from } P(n, p) \text{ that creates a cycle in } ST$. Then, $C(m, m) = T_q(n) = C(n, n)$ holds. In addition, nodes $n \in P(m, p_m)$ creates a cycle in $ST$. Therefore, $C(m, m) = T_q(n)$, indicating that $T_q(n)$ is never a part of the optimal solution at $s_0$.

We prove the admissibility of BLDFS, with more relaxed assumptions than [Bonet and Geffner, 2005b].

**Lemma 5.** If Algorithm 6 (BLDFS$_\text{Driver}$) starts with $s_0.th = b \ll \infty$ at the root $s_0$, then it terminates in finite time with either $s_0.r > b$ or $s_0.slvd = \top$.

**Theorem 6 (Admissibility).** BLDFS$_r$ finds an optimal solution cost in finite time, if one exists in the finite search space.

Proof sketch. If $s_0.th \geq C(s_0, s_0)$, BLDFS$_\text{Driver}$ returns an optimal solution cost $C(s_0, s_0)$ in finite time, since: (1) Theorems 2 and 4 ensure solution optimality. (2) Each node $n$ in the optimal solution tree is examined, which is easily proven by induction on the number of chains of the recursive calls.

If $s.th < C(s_0, s_0)$, by Lemma 5, BLDFS$_\text{Driver}$ strictly increases $s.th$ in finite time, since BLDFS$_\text{Driver}$ cannot mark $s_0$ as solved (see line 12 in BLDFS$_\text{Driver}$). Since $C(s_0, s_0) < \infty$, in a finite number of iterations, BLDFS$_r$ sets $s_0.th = b > C(s_0, s_0)$, enabling to optimally solve any solvable instance in finite time.

The results in [Kishimoto and Marinescu, 2014] hold only for finite DAGs (i.e., no cycles involved). On the other hand, RBFAOO$_r$ focuses on finite cyclic spaces.

**Lemma 7.** Assume that RBFAOO$_r$ examines a node $n$ via path $p$ with $th(n) = b \ll \infty$. Then, RBFAOO$_r$ backtracks to $n$ in finite time with either $T_r(n, p) > b$ and/or $n$ is marked as solved.

**Theorem 8 (Admissibility).** RBFAOO$_r$ finds an optimal solution cost in finite time, if one exists in the finite search space.

Proof sketch. Let $s_0$ be the root and $ST$ be an optimal solution tree rooted at $s_0$. If no path exists which makes RBFAOO$_r$ deviate from ST (i.e., $ST$ is the whole search space), since $C(s_0, s_0) < \infty$ and $k = \infty$, RBFAOO$_r$ finds an optimal solution cost in finite time. This is easily verified by the fact that (1) RBFAOO$_r$ requires only reasonable memory and (2) $T_r(n, p) \leq C(n, p) < th(n) = \infty - k$ for each $n \in ST$, where $k$ is a small number. This $th(n)$ is large enough to cover all terminal nodes in $ST$. Otherwise, let $n$ via path $p$ be the first OR node in $ST$ at which RBFAOO$_r$ generates two children $F_{n,a_1}$ and $F_{n,a_2}$, where $F_{n,a_1} \in ST$ but $F_{n,a_2} \not\in ST$. Here, we show only the case with two children, since more general cases are proven in an
analogous way. \( F(n, a_2) \) may or may not lead to another optimal solution, but because of the optimality of \( ST \), \( C(n, p) = c(n, a_1) + C(F(n, a_1, p_{a_1}) \leq c(n, a_2) + C(F(n, a_2, p_{a_2})) \) holds, where \( p_a = p + a \). RBFAOO \( \tau \) reaches \( n \) with \( th(n) = \infty - k \) where \( k \) is a small number.

\[ T_r(F(n, a_2)) = \infty \] means that RBFAOO \( \tau \) cannot deviate from \( ST \). Hence, we consider the case where \( T_r(F(n, a_2)) < \infty - k \), i.e., \( c(n, a_2) + T_r(F(n, a_2)) < \infty - k \). Since \( F(n, a_1) \) has an optimal solution, \( c(n, a_1) + T_r(F(n, a_1, p_{a_1}) \leq C(n, p) < \infty - k \) holds (see Lemma 1 as well). Therefore, the threshold \( t \) set to examine \( F(n, a_1) \) and \( F(n, a_2) \) always satisfies \( t < \infty - k \).

If RBFAOO \( \tau \) examines \( F(n, a_1) \) with \( t < C(n, p) - c(n, a_1) \), by Lemma 7, RBFAOO \( \tau \) backtracks to \( n \) in finite time. It holds that if \( T_r(F(n, a_1, p_{a_1}) \) is strictly increased, or/and \( F(n, a_1) \) is solved. If \( F(n, a_1) \) is solved but \( v = c(n, a_1) + C(F(n, a_1, p_{a_1}) > c(n, a_2) + T_r(F(n, a_2, p_{a_2}) \) still holds, \( F(n, a_2) \) is examined with \( th(F(n, a_2)) = v - c(n, a_2) \). By Lemma 7, this step completes in finite time, proving that \( a_1 \) leads to an optimal solution cost.

Analogously, if RBFAOO \( \tau \) examines \( F(n, a_2) \) with \( t < C(n, p) - c(n, a_2) \), by Lemma 7, RBFAOO \( \tau \) backtracks to \( n \) in finite time and \( T_r(F(n, a_2, p_{a_2}) \) is strictly increased, or/and \( F(n, a_2) \) is solved. If \( F(n, a_2) \) is solved, \( T_r(F(n, a_2, p_{a_2}) \geq C(n, p) - c(n, a_2) \).

Since RBFAOO \( \tau \) requires only reasonable memory, it remembers \( T_r(F(n, a_1, p_{a_1}) \) and \( T_r(F(n, a_2, p_{a_2}) \). By Lemmata 1 and 7, even if \( F(n, a_1) \) remains unsolved, RBFAOO \( \tau \) strictly increases \( T_r(F(n, a_1, p_{a_1}) \) and \( T_r(F(n, a_2, p_{a_2}) \), eventually leading to \( c(n, a_1) + T_r(F(n, a_1, p_{a_1}) \leq C(n, p) \leq c(n, a_2) + T_r(F(n, a_2, p_{a_2}) = b \) in finite time. When this happens, RBFAOO \( \tau \) examines \( F(n, a_1) \) with \( th(F(n, a_1)) = th(F(n, a_1)) - c(n, a_1) \). If \( T_r(F(n, a_2, p_{a_2}) = \infty \) (i.e., \( F(n, a_2) \) is proven unsolvable), RBFAOO \( \tau \) cannot deviate from \( ST \). If \( b < \infty \), \( th(F(n, a_1)) = b - c(n, a_1) \), since \( th(n) = \infty - k \). By Lemma 7, RBFAOO \( \tau \) returns to \( F(n, a_1) \) in finite time.

Since \( T_r(F(n, a_2, p_{a_2}) \leq C(h(n), p_{a_1}) \leq b - c(n, a_1) \), \( T_r(F(n, a_1, p_{a_1}) \) cannot be strictly increased. Therefore, RBFAOO \( \tau \) must mark \( F(n, a_1) \) as solved, resulting in solving \( n \) in finite time. If \( a_2 \) also leads to another optimal solution, RBFAOO \( \tau \) may choose \( F(n, a_2) \) rather than \( F(n, a_1) \). However, the same discussion holds, resulting in RBFAOO \( \tau \) making \( F(n, a_2) \) as solved in finite time.

Theorem 2 ensures the solution optimality, once RBFAOO \( \tau \) finds it. Hence, RBFAOO \( \tau \) returns an optimal solution cost for \( s_0 \) in finite time if a solution exists.

Theorem 8 leads to a new corollary for RBFS [Korf, 1993] with the TT and our approach, denoted by RBFS \( \tau \). While the admissibility of RBFS assumes that the search space is a tree, RBFS \( \tau \) further reduces duplicate search on cyclic spaces.

**Corollary 9.** If a solution exists in the finite search space, RBFAOO \( \tau \) finds an optimal solution in finite time.

RBFAOO \( \tau \) and BLDFS \( \tau \) can prove unsolvability for the graph in Figure 1(right), but have limitations in other cases:

**Theorem 10.** RBFAOO \( \tau \) and BLDFS \( \tau \) are incomplete when proving unsolvability in the cyclic search space.

**Proof sketch.** The example shown in Figure 2 cannot be solved. An explicit cycle cannot be detected due to large values of \( r(B, A \rightarrow B) \) and \( r(C, A \rightarrow C) \). This is easily proven by induction in a similar way to [Akagi et al., 2010].

### 6 Experimental Results

We use domain-independent nondeterministic planning with unit edge costs and the ADD model to evaluate RBFAOO \( \tau \) and BLDFS \( \tau \). Baselines RBFAOO \( \tau \) and BLDFS unfold repeated states, ensuring solution optimality with the limited TT size or inconsistent heuristics. We modified slightly the original BLDFS [Bonet and Geffner, 2005a] to delay saving a solution cost in the TT until it is proven optimal, as discussed in Section 4. To the best of our knowledge, this change has not empirically impacted the performance, since a first solution our BLDFS returned is optimal in all cases we observed. We coded all algorithms in C++ (64-bit) and ran experiments on an Intel Xeon CPU X5690 processor at 3.47GHz. The TT is limited to 1GB. The time per instance is 30 minutes. When the TT is full, SmallTreeGC [Nagai, 1999] discards \( \% \) TT entries with small subtrees. As in [Akagi et al., 2010; Kishimoto and Marinescu, 2014], we set \( \% \) to 30.

We extended the \( h \)\textsuperscript{max} heuristics [Bonet and Geffner, 2001; Haslum and Geffner, 2000] to nondeterministic planning, which is consistent. In constructing an inconsistent heuristic from \( h \)\textsuperscript{max}, we perform the perturbation as follows: We represent a state as a bit vector of true or false values of grounded predicates. Among the algorithms compared, the same heuristic values are returned when the same states are evaluated. A set of grounded predicates are randomly selected with the same seed among the algorithms, before search is performed. Whether a small value is subtracted or not is determined by whether one of the selected predicates is true or false. We select 5% of the grounded predicates.

**Unsolvable Instances and Inconsistent Heuristics.** We use instances from the Uncertainty Track in the 6th International Planning Competition in 2008, consisting of BWD (blocksworld, 30 instances), FLTS (faults, 55 instances) and FRES (first-responders, 100 instances). All these instances admit solutions with cycles. They are unsolvable under the cycle-free solution requirement. Table 1(top) shows the number of instances proven unsolvable. Without any search, the inconsistent heuristic proves that 26 instances in BWD, and all 100 instances in FRES are unsolvable. That is, no further improvement can be achieved on these instances even with a better search algorithm. For the remaining four instances in BWD, and all 55 instances in FLTS, the heuristic cannot prove unsolvability at their initial states. In these 59 instances, RBFAOO \( \tau \) and BLDFS time out, since all they can do is to unfold repeated states until exceeding their thresholds. On the other hand, both RBFAOO \( \tau \) and BLDFS \( \tau \) successfully prove unsolvability of all instances in BWD and 14 instances in FLTS (i.e., a total of 18 additional instances proven successfully).
clearly demonstrating the effectiveness of our approach. The p7,2 instance in FLTS is the most difficult that RBFAOO
and BLDFS, could prove as unsolvable: RBFAOO, expands 288,862,627 nodes (589 seconds), while BLDFS expands 184,513,049 nodes (351 seconds).

**Solvable Instances and Inconsistent Heuristics.** Since the instances considered in the previous subsection are unsolvable, we modified them to admit cycle-free solutions, as described in [Fu et al., 2013]. Specifically, in BWD, we make only the pick-up action nondeterministic to ensure that the optimal solution is a tree. In FRES, we additionally allow hospitals to accept victims at status dying. This change allows to have cases where a solution is not always a sequence of actions but a tree. However, 27 instances are proven to be unsolvable, resulting in 73 solvable instances. These three domains have different characteristics. Nondeterministic actions tend to be selected much more frequently in FRES than in BWD. While the search spaces of BWD and FRES contain cycles, the search space of FLTS is a DAG. For DAGs, since \( r(n, p) = q(n) \) always holds for any node \( n \) and path \( p \), RBFAOO, and BLDFS, have no advantage over RBFAOO and BLDFS but incur extra overhead due to calculating both \( r \)-values and \( q \)-values. Table 1 (bottom) shows the number of instances solved optimally by each algorithm, where the number inside the parentheses is the total search time of the solved instances in seconds. When finding an optimal solution cost, our approach preserves the solving ability of RBFAOO and BLDFS without sacrificing their running time. RBFAOO, and RBFAOO solve the same set of instances, and so do BLDFS, and BLDFS. Search time differences range from 1 to 7%, which is a small overhead, compared to benefits pointed out earlier. For example, in FLTS where the search space is a DAG, RBFAOO, and RBFAOO examine the same portions of the search space, resulting in RBFAOO, running 2% slower than RBFAOO due to the extra computational overhead. We observed a similar phenomenon for BLDFS, and BLDFS.

**Limited TT and Consistent Heuristics.** We selected 5 difficult instances (four have optimal solutions and one does not), and ran an experiment using the consistent \( h_{\text{max}} \), a TT limited to 30 MB, and a 2-hour time limit per instance. Table 2 shows the runtimes in seconds. Both RBFAOO, and BLDFS, solve three instances including the one (p7,2) which has no solution in the cyclic search space. On the other hand, RBFAOO and BLDFS solve only two instances which have optimal solutions due to a lack of the cycle detection scheme. We also see that our approach does not sacrifice the ability to solve solvable instances.

### Table 1: Performance on unsolvable instances (top) and solvable instances (bottom).

<table>
<thead>
<tr>
<th>Domain</th>
<th>RBFAOO,</th>
<th>RBFAOO</th>
<th>BLDFS,</th>
<th>BLDFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BWD (30)</td>
<td>30</td>
<td>26</td>
<td>30</td>
<td>26</td>
</tr>
<tr>
<td>FLTS (55)</td>
<td>14</td>
<td>0</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>FRES (100)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Domain</th>
<th>RBFAOO,</th>
<th>RBFAOO</th>
<th>BLDFS,</th>
<th>BLDFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BWD (30)</td>
<td>10 (2.83)</td>
<td>10 (3.05)</td>
<td>10 (2.68)</td>
<td>10 (2.67)</td>
</tr>
<tr>
<td>FLTS (55)</td>
<td>48 (4344)</td>
<td>48 (4233)</td>
<td>47 (2326)</td>
<td>47 (2247)</td>
</tr>
<tr>
<td>FRES (73)</td>
<td>26 (4452)</td>
<td>26 (4497)</td>
<td>27 (5156)</td>
<td>27 (5072)</td>
</tr>
</tbody>
</table>

### Table 2: Runtime with consistent heuristic and limited TT. The instances not shown here are not solved by any method.

<table>
<thead>
<tr>
<th>Domain</th>
<th>RBFAOO,</th>
<th>RBFAOO</th>
<th>BLDFS,</th>
<th>BLDFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLTS p4,2</td>
<td>153</td>
<td>153</td>
<td>113</td>
<td>111</td>
</tr>
<tr>
<td>FLTS p8,7</td>
<td>834</td>
<td>818</td>
<td>724</td>
<td>712</td>
</tr>
<tr>
<td>FLTS p7,2</td>
<td>1786</td>
<td>≥ 2h</td>
<td>1608</td>
<td>≥ 2h</td>
</tr>
</tbody>
</table>

### 7 Related Work

Value Iteration (VI) can optimally solve search problems by solving the Bellman equation using dynamic programming [Bellman, 1957]. However, it is limited to solving problems with relatively small search spaces that fit into memory, and may not be able to prove unsolvability due to cycles.

CFC_{REV} [Jiménez and Torras, 2000] is a best-first search like AO* [Nilsson, 1980] that can handle cyclic search spaces to find a cycle-free optimal solution but it needs to keep the examined search space in memory. Liang and Su (2005) presents an algorithm that assumes all terminal nodes are known beforehand and available in memory, which does not hold in domains such as nondeterministic planning.

MAO* [Chakrabarti et al., 1989] is an AO* variant that can operate with limited memory but assumes the search space is a DAG and uses a specific strategy to remove nodes from memory. Our approach makes no specific assumption about the replacement strategies of the TT, and the search space can be cyclic as long as an optimal solution is a tree.

LAO* [Hansen and Zilberstein, 2001] combines AO* with dynamic programming (value or policy iteration) to handle optimal solutions involving cycles. LAO* does not operate with limited memory. In addition, in our setting, cycles need to be regarded as unsolvable, and optimal solutions may not contain cycles if they exist. LDFS(MDP) [Bonet and Geffner, 2006] extends LDFS to solve MDP, allowing for cyclic optimal solutions as LAO* does. Extensions of LAO* and LDFS(MDP) to our scenario remain open.

### 8 Conclusions

In this paper, we focused on finding optimal acyclic solutions in cyclic AND/OR search spaces. We showed in detail how to integrate our ideas into two state-of-the-art algorithms. Our new theoretical analysis addresses important gaps for cases of insufficient cache memory or inconsistent heuristics. Experiments in domain-independent planning demonstrate a boost in the ability to prove unsolvable difficult instances.

In future work we plan to investigate complete algorithms across unsolvable instances. As previously mentioned, an analysis of the original BLDFS technique in combination with our enhancements is another interesting direction.

### Acknowledgments

We would like to thank Masatomo Asai for his constructive comments on the paper.
References


