# Exploiting Belief State Structure in Graph Search

**Jason Wolfe and Stuart Russell**
Computer Science Division
University of California, Berkeley, CA 94720
jawolfe@cs.berkeley.edu, russell@cs.berkeley.edu

## Abstract

It is well-known that eliminating repeated states is essential for efficient search of state-space AND-OR graphs. The same technique has been found useful for searching belief-state AND-OR graphs, which arise in nondeterministic partially observable planning problems and in partially observable games. Whereas physical states are viewed by search algorithms as atomic and admit only equality tests, belief states, which are sets of possible physical states, have additional structure: one belief state can subsume or be subsumed by another. This paper presents new algorithms that exploit this property to achieve substantial speedups. The algorithms are demonstrated on Kriegspiel checkmate problems and on a partially observable vacuum world domain.

## Introduction

For a wide variety of AI search problems, the identification of previously encountered subproblems is essential for achieving good performance. Successful applications of this idea to date include A* for single-agent graph search, $\alpha$-$\beta$ with transposition tables for adversarial graph search, forward and backward subsumption in theorem proving, and the caching of "nogoods" in CSP- and SAT-solving (Bayardo & Schrag 1997) and planning (Hoffmann & Koehler 1999). In this paper, we present a novel application of this idea for sharing solutions between sets of possible worlds in partially observable planning problems.

In general, partially observable planning is concerned with devising strategies for achieving goals even when the true state of the world is unknown (*partial observability*), the effects of actions may be uncertain (*nondeterminism*), and future actions may depend on the observations received to date (*contingency*). In this paper, we consider finding *fixed-depth*, *acyclic* plans that are guaranteed to reach a goal within a given number of steps, despite present and future uncertainty. As we will see, this problem is isomorphic to that of finding *guaranteed win* strategies in partially observable games, which ensure that a player will achieve the optimal payoff within some finite horizon. One well-known approach for constructing such plans (see, e.g., Chapters 3 and 12 of (Russell & Norvig 2003)), which we adopt, is to search an AND–OR tree whose nodes correspond to *belief states* (which are *sets* of possible *physical states*).

Copyright © 2007, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

The idea of detecting *exactly repeated belief states* during AND–OR search has already been successfully exploited in the literature: Sakuta and Iida (2000) consider several hashing schemes, and Bertoli *et al*. (2001) utilize ordered binary decision diagrams (OBDDs), both of which allow for the detection of repeated belief states in constant time. Once detected, repeated belief-state nodes are exploited as follows. First, if a previously evaluated belief state is encountered again, its value is re-used without additional effort. Second, cyclic plans are avoided, by detecting when a belief state is repeated earlier in the current plan. Both works use variants of a standard DFS (depth-first search) algorithm for searching ordinary AND/OR graphs, which treats belief states as black boxes that admit only equality tests.

A key observation, which to our knowledge has not been fully exploited, is that belief states have additional structure that is relevant for search. In particular, even if two belief states are not identical, one may be a subset or superset of the other. This is relevant, because a successful plan for a belief state is just a plan that works in all of its physical states. Thus, this plan must also work on any subset of that belief state; conversely, if no plan exists for some belief state, all of its supersets must be unsolvable as well.

To our knowledge, only two previous works have explored this direction. First, Kurien *et al*. (2002) created the *fragPlan* algorithm, which builds up a conformant plan for a belief state one physical state at a time. Second, Russell and Wolfe (2005) proposed a general *incremental* framework for belief-state AND–OR search, which recognizes uncertainty over physical states as a new possible search dimension in addition to depth and breadth. Both works share the same basic idea: if we build up a plan for a belief state incrementally, we can fail early if we find one of its subsets to be unsolvable. However, neither of these works attempts to exploit belief-state structure much beyond this special case.

This paper's primary contribution is a set of techniques for fully exploiting the presence of *related belief-states* in belief-state AND–OR trees during search. In particular, our algorithms will efficiently detect previously *proved supersets* and *disproved subsets* of a current belief state, as well as *generalized cycles* in which a subset of the current belief state appears earlier in the current plan.

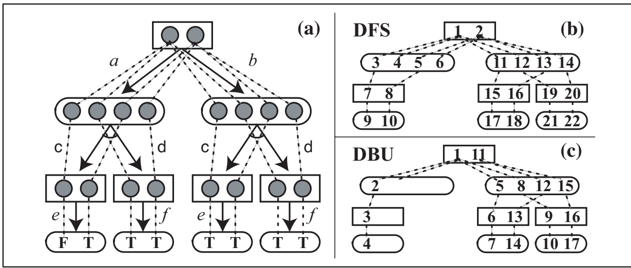



Figure 1: (a) A simple belief-state tree for a nondeterministic domain. Circles are nonterminal physical states, T/F are terminal physical states, rectangles are belief-state OR-nodes, and ovals are belief-state AND-nodes. (b/c) The order in which search algorithm DFS/DBU would examine the physical states in this tree.

We first show how to extend DFS to this setting, by supplementing a standard algorithm for AND/OR graph search with a data structure that supports sub/superset queries. In a sense, this is an obvious extension to the previous work, and it is somewhat surprising that it has remained unexplored. One possible explanation is that since sub/superset queries come at a potentially hefty cost — unlike equality queries, they cannot be done in constant time — it is not at all obvious that the benefits of this approach would outweigh the costs. A key result of this paper is that the extra effort often *does* pay off; in two unrelated domains, we demonstrate order-of-magnitude improvement over previous algorithms.

We also apply this idea to search algorithm DBU, a member of the incremental family mentioned above, which was previously found to be more effective than DFS in some domains (Russell & Wolfe 2005). In the process we develop an extension of DBU for graph search, which did not exist previously. This requires some additional insights, since (unlike DFS) the relationships between belief-state nodes can change during DBU's searches. In our experiments, these new DBU variants provided up to an additional order-of-magnitude speedup.

In total, we present pseudocode for four search strategies, tree- and graph-search variants of both DFS and DBU. Each graph variant can be used with either equality or sub/superset queries, leading to a total of six implemented algorithms, all of which have been proven sound and complete. Due to lack of space, these proofs are omitted, and pseudocode has been simplified substantially; details of our actual implementations are given at the end of each section.

## Belief-State AND–OR Trees

A belief-state AND–OR tree naturally captures the space of potential plans for a nondeterministic, partially observable planning problem. In such a tree, nodes correspond to belief states (sets of physical states indistinguishable given the available information), OR-nodes represent action choices by the planning agent, and AND-nodes represent the arrival of percepts (new information, which may allow the agent to discriminate between previously indistinguishable states).

Figure 1(a) shows a simple, abstract example of a belief-state AND–OR tree. In the root belief-state node there are two possible physical states (shown as circles), which correspond to different possible world states. The agent knows it will start in one of these states, but will not know which one; thus, it must find a (possibly conditional) plan of action that can achieve the goal from either state. As a concrete example, suppose that you and a coauthor are traveling to Providence to present a paper. Your coauthor has already caught his flight, and you realize that you are not sure whether or not he has brought a copy of the slides for the talk.

The root of Figure 1(a) is an *OR-node* (rectangle), from which the agent can choose to take action *a* or *b*. Each action leads to an AND-node (oval), which contains the results of applying that action in every possible physical state. If an action has nondeterministic effects, it may take a single physical state to multiple possible successors. For example. perhaps the actions correspond to boarding the plane with your slides either (*a*) in your checked bag, or (*b*) in your carry-on. Moreover, this action is nondeterministic, as your checked bag may not make it onto the plane with you.

*AND-nodes* correspond to the arrival of percepts, which *partition* the belief state into disjoint subsets based on the observation expected in each physical state (percepts do not change the underlying physical states). For example, after arriving at the Providence airport, you get to observe whether (c) your bag was lost, or (d) it arrived successfully. From this point, you can either try to (*e*) present with your coauthor's laptop, or (*f*) use your own (not all of the possible options are shown in the figure).

Finally, nodes can be assigned truth values. Physical states are assigned values first: each one may be nonterminal, $true$ (i.e., a goal state), or $false$. Then, AND-nodes are proven ($true$) iff *none* of their physical states are $false$, and *all* of their children are proven (since the agent must plan for all possibilities). Likewise, OR-nodes are proven iff *all* of their physical states are $true$, or *any* of their children are proven (since the agent can choose which actions to take). In the example, only the bottom left physical state (where your coauthor forgot his slides, and your copy was lost with your bag) is $false$; all other possibilities result in a successful presentation. As a consequence, the three belief-state nodes on the path *a*, c, *e* are $false$, and all other nodes are $true$.

A *proof tree* is a subset of the entire AND–OR tree for a problem, in which exactly one action is retained at each OR-node, and the root is proven by the above rules. Such a proof tree corresponds directly to an executable plan, in which the agent does the specified action at each OR-node. For example, the right subtree of the example is a proof tree for the instance, in which you first do action *b*, then do action *e* or *f* depending on whether you observe percept c or d. This plan guarantees that you will reach a $true$ physical state, regardless of which state you start in and the outcome of the nondeterministic action *b*.

An important issue, which we have not yet discussed, is how belief states should be represented. Symbolic techniques such as the ordered binary decision diagram (OBDD) methods developed by Bertoli *et al.* (2001) are popular in the planning community, since they can sometimes represent large sets of states compactly. However, such symbolic methods impose an overhead of several orders of magnitude for complex games such as chess (Russell & Wolfe 2005), and except for certain tractable cases (Amir & Russell 2003)

they provide no guarantees of compactness. An alternative popular in the partially observable games community (Ciancarini, Libera, & Maran 1997; Sakuta & Iida 2000) is to represent belief states as *explicit* sets of atomic physical states. While this approach is simpler, it can be inefficient for large but "simple" belief states.[1] In this work we choose this latter option, primarily because it allows us to utilize previous work on subset queries; however, the ideas presented should apply to symbolic representations as well.

## Searching Belief-State AND–OR Trees

This section describes two algorithms—DFS and DBU—for searching belief-state AND–OR trees. Figure 1(b+c) depicts the order in which these algorithms would examine the physical states when searching the tree in Figure 1(a).

The first algorithm, DFS, is just standard depth-first AND–OR search on the belief-state tree. As can be seen in Figure 1(b), DFS visits each belief-state node in turn, constructing all of its physical states before moving on to the next node. Figure 2 provides pseudocode for a recursive implementation of DFS, which can be invoked by DFS-AND(*belief state*, *depth limit*). At OR-nodes, it tries each move in turn, returning $true$ iff some successor belief state (corresponding to an AND-node) is solvable. At AND-nodes, it tries each percept in turn, returning $true$ iff all of the partitions (corresponding to OR-nodes) are goal states, or are recursively solvable within the provided depth limit.

The second algorithm, DBU (for depth-then-breadth-then-uncertainty), is a simple instance of the *incremental* search framework described in the introduction. Whenever DBU encounters a belief state, it first constructs a proof for a single physical state, and then extends the proof to cover additional states one-by-one. More specifically, the algorithm performs successive depth-first searches for solutions in the *physical-state trees*, with the constraint that the solutions should make up a single belief-state proof tree (which is continually extended and refined in the process). Unlike DFS, DBU can find *minimal disproofs* that consider only a single element of each belief state (compare, e.g., the left branches of Figure 1(b) and (c)). In the best case, this leads to a speedup equal to the average belief-state size in the tree.

The implementation of DBU, which can be invoked by DBU-AND(*new AND-node*, *belief state*, *depth*), maintains state in the form of an explicitly represented belief-state proof tree, which is necessary because the same belief-state node may be visited multiple times while constructing and extending a proof. DBU-OR adds a single physical state to its current proving child at each operation, unless a disproof of that move is found; in that case, it must repair the plan, re-examining all previous states on a new possible move.

As written, both DFS and DBU return $true$ or *false* to indicate whether a problem is solvable or not; they could easily be extended to return proof trees (i.e., plans) for solvable instances instead. Each algorithm can be applied directly to find solutions up to some fixed length $d$, or be combined with iterative deepening to find optimal (shortest) solutions.

```
/* states is a set of physical states, d is remaining depth.   */

function DFS-OR(states, d) returns true/false
  for each m ∈ ACT(states) do
    successors ← REMOVE-DUPS( ∪_{s∈states} SUCC(s, m))
    if DFS-AND(successors, d) then return true
  return false

function DFS-AND(states, d) returns true/false
  if false ∈ PERCEPTS(states) then return false
  for each p ∈ PERCEPTS(states) do
    if p = true then continue
    if d = 0 then return false
    r ← DFS-OR({s | s∈states ∧ PERCEPT(s)=p}, d - 1)
    if r ≠ true then return r
  return true

/* b is a belief-state OR-node or AND-node.
 * All nodes have mappings CHILD(b)[k], for move/percepts k.
 * OR-nodes have a list of states STATES(b),
 * and an ordered list MOVES(b), both initially empty.
 * AND-nodes have a set of states VISITED(b).              */

function DBU-OR(b, state, d) returns true/false
  add state to STATES(b)
  new ← {state}
  for each m ∈ MOVES(b) do
    if NULL(CHILD(b)[m]) then
      CHILD(b)[m] ← a new AND-node
    if (∀s∈new)DBU-AND(CHILD(b)[m],SUCC(s,m),d) then
      return true
    delete m from MOVES(b) and delete CHILD(b)[m]
    new ← STATES(b)      /* start new proof from scratch */
  return false

function DBU-AND(b, states, d) returns true/false
  if false ∈ PERCEPTS(states) then return false
  for each s ∈ states do
    p ← PERCEPT(s)
    if p = true ∨ s ∈ VISITED(b) then continue
    if d = 0 then return false
    add s to VISITED(b)
    if NULL(CHILD(b)[p]) then
      CHILD(b)[p] ← a new OR-node with MOVES(·)=ACT(s)
    r ← DBU-OR(CHILD(b)[p], s, d - 1)
    if r ≠ true then return r
  return true
```

Figure 2: DFS and DBU, depth-first belief-state AND–OR tree search algorithms. ACT(·) returns the applicable actions from some state(s), SUCC($s$, $m$) returns the successors of physical state $s$ on action $m$, and PERCEPT($s$) returns the last percept in state $s$. For simplicity, we assume that the terminal relation is observable via the special percepts *true*/*false*, and that action preconditions depend only upon observable information; both can be relaxed at the cost of some additional code complexity.

## Belief-State AND–OR Graphs

This section describes an extension of our belief-state AND–OR tree model to incorporate information provided by re-

[1] For example, in a propositional domain with $n$ fluents and no state constraints, the simple formula $true$ represents a belief state containing all $2^n$ possible states. It is an open question to what extent such belief states tend to arise arise in realistic domains.

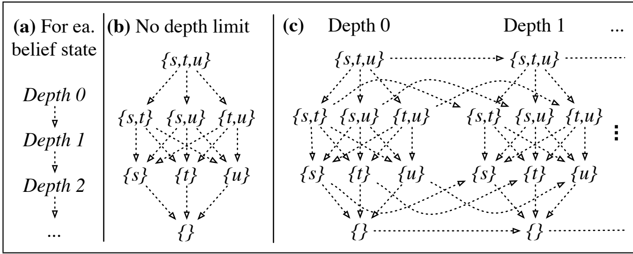



Figure 3: Subsumption lattices relevant for solving belief-state AND–OR search problems. Each dotted arc stipulates that if its source is solvable, then its target is solvable too. (a) If a belief state is solvable at a given remaining depth, it is solvable at all greater depths as well. (b) If a belief state is solvable, all of its subsets are solvable as well. (c) The most general lattice for our problem.

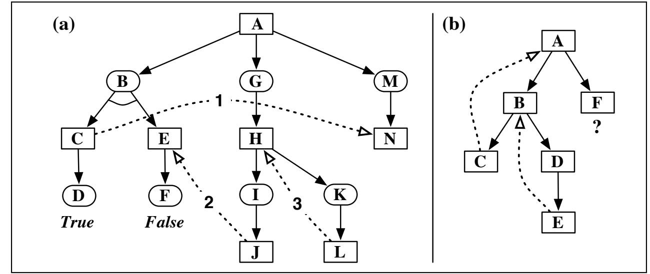



Figure 4: (a) A belief-state AND/OR graph, which includes some relevant subsumption arcs. For readability, physical states are not shown. Arcs *1* and *2* are proving and disproving cross-edges (resp.), whereas arc *3* is a cyclic back-edge. (b) A portion of a tree that might arise when searching up to node **F**. Only OR-nodes and back-edges are shown; we assume that all other moves at OR-nodes have been disproved. AND-nodes are omitted, as each had only one relevant branch, which points to the OR-node (shown) that caused failure by returning $stack_d$.

peated and/or related belief states. In particular, we consider adding new *subsumption arcs* to our trees, which indicate that the solvability of a target belief-state node is *entailed* by the solvability of a source node. This formulation differs from the standard approach to state-space graph search, in that it allows our algorithms to consider relationships between non-identical nodes. We will begin by looking at the *subsumption lattices* underlying the presence of repeated and related belief states, and then show how this information can be incorporated into search in the next sections.

In the case of exactly repeated belief states, the subsumption lattice has the simple form shown in Figure 3(a). This captures the fact that, e.g., the solvability of a belief-state node $\{s, t\}$ at remaining depth 5 is entailed by the solvability of a belief-state node $\{s, t\}$ at depth 3. In the more general setting of finding related belief states, we also consider subsumption relationships between belief-states and their subsets, as shown in Figure 3(b). In combination with depth generalization, this leads to the fully general lattice of Figure 3(c), which is the one that our algorithms that exploit related belief states will use.

## DFS for Belief-State AND–OR Graphs

This section describes changes to DFS needed to utilize information about repeated or related belief states. Functionality is added to DFS-OR to maintain a cache of previous results, and discover *useful* subsumption relationships (drawn from the reflexive, transitive closure of the relevant lattice) at each step. In particular, the algorithm will find previous proofs that *subsume* the current node, and previous disproofs or nodes on the stack that are *subsumed by* the current node.

Figure 4(a) depicts a belief-state graph with subsumption arcs, which might be constructed by DFS. The first move from the root leads to AND-node **B**, which has two children corresponding to different possible percepts. While the first child is easily proved, the other child is disproved and so the algorithm backtracks to try the second move. It proceeds until it hits node **J**, at which point it discovers a *subsumption cross-edge* to **E** (e.g., perhaps **J** has belief state $\{s, t, u\}$ at remaining depth 4, and **E** has $\{s, u\}$ at depth 6). Because **J** subsumes **E** and **E** is already known to be $false$, the algorithm concludes that **J** is also unsolvable and backtracks.

The next interesting point is at node **L**, where the algorithm discovers a *subsumption back-edge* to **H**. This indicates that any plan that works from **L** would also work at **H** directly, and so there is no purpose in considering further plans from **L**. This is a generalization of the usual notion of cyclic plan avoidance, because belief-state **L** may be a superset of **H** rather than being strictly identical to it.

Finally, the algorithm proceeds to examine the third move, and finds a subsumption cross-edge *from* node **C** to node **N**. Because node **C** is already known to be $true$, this implies the truth of node **N** and the algorithm returns success.

**function** DFS-OR($states$, $d$) **returns** $true/false/stack_{d'}$
  **if** $(\supseteq) states$ proven at depth $\leq d$ **then return** $true$
  **if** $(\subseteq) states$ disproven at depth $\geq d$ **then return** $false$
  **if** $(\subseteq) states$ on the stack at depth $d'$ **then return** $stack_{d'}$
  set $states$ as on the stack at depth $d$
  $maxstack \leftarrow 0$
  **for each** $m \in$ ACT($states$) **do**
    $r \leftarrow$ DFS-AND(REMOVE-DUPS($\cup_{s \in states}$SUCC($s$,$m$)), $d$)
    **if** $r$ = $true$ **then**
      set $states$ as off the stack and proved at depth $d$
      **return** $true$
    **else if** $r$ = $stack_{d'}$ **then**
      $maxstack \leftarrow$ MAX($maxstack$, $d'$)
  set $states$ as off the stack
  **if** $maxstack >$ d **then return** $stack_{maxstack}$
  set $states$ as disproved at depth $d$ and **return** $false$

Figure 5: DFS-OR for *graph* search (DFS-AND is unchanged).

Figure 5 shows a modified DFS-OR function, which can detect and exploit subsumption relationships between nodes during search. It is essentially just a memoized version of DFS-OR from Figure 2, with two exceptions. First, the cache lookups generalize to previously unseen inputs, by drawing on the relevant subsumption lattice. Second, extra logic has been added to avoid considering cyclic plans. The basic idea behind cycle avoidance is simple: while in the process of solving a belief state, mark it as being on the stack; then, if an identical belief state (or superset) is encountered while searching deeper, immediately fail without considering further extensions.

However, combining cycle avoidance with memoization

requires additional care, to avoid the so-called graph history interaction (GHI) problem (Campbell 1985). Consider the simplified belief-state tree (showing only OR-nodes) in Figure 4(b), corresponding to a partial DFS search of **A** up to node **F**. The search of child **B** failed due to detected cycles on all branches, and so one might consider associating **B** with value $false$ in the cache. However, this would be incorrect: if **F** is found to be $true$, then a successful plan may exist for **B** as well (through node **C**). While this plan would not be relevant in this context, it might be if **B** was encountered later in a new context in which **A** was not on the stack. Thus, at the point when a node is exited, if there were detected back edges from its descendants to its ancestors, no value is stored for it in the cache.

If, on the other hand, **F** was disproved, then it would be correct to cache **A** as being $false$. More generally, if all cycles have been "closed" without finding a solution, then *all* involved nodes are unsolvable. In Figure 5, this logic is implemented by adding a new return value $stack_d$, distinct from $true$ or $false$, which indicates that the search for a proof was halted due to back edges, the shallowest target of which was a node on the stack at depth $d$. This signal is treated the same way as $false$, except that it is not cached.[2]

Our actual implementation includes several improvements, which were omitted in the pseudocode for simplicity. First, it computes and caches the *actual* depths of proofs and disproofs, rather than just using the current remaining depth. Second, whereas the pseudocode leaves the values of nodes such as **B** in Figure 4(b) as "unknown", our implementation goes back and assigns them a value whenever possible (e.g., if node **F** was disproved, **B**-**E** would all be assigned $false$).

## The DFS= Algorithm

The simplest belief-state graph algorithm we consider is "DFS=", which uses our modified DFS procedure in combination with a hash table for detecting exactly repeated belief states in constant time.[3] We generalize to new depths by hashing each belief state to a status object [$maxDisproof$ $stack$ $minProof$], initially [-1 $nil$ $\infty$], which stores the maximum depth at which the belief state is known to be unsolvable, the current depth at which it is on the stack (or $nil$ if none), and the minimum depth at which it is known to be solvable. Queries are executed by retrieving the object associated with a belief state and comparing its remaining depth with the stored values; then, updates are executed by simply replacing the relevant values.

[2]This solution has some similarities to Breuker's Base-Twin Algorithm (BTA) for best-first search (1998). However, whereas the BTA considers *all identity* relationships between nodes, our approach draws on a subsumption lattice and considers only *relevant subsumption* relationships. Moreover, our use of depth-first search simplifies things considerably; since each node is visited exactly once, there is no need to regenerate "possible draw" markings or modify a "base node" through its "twin" like in the BTA.

[3]Sakuta and Iida (2000) concluded that when physical-state hash values are Zobrist keys, the best combiner is a simple sum. While the sum is convenient since it is symmetric, we found that it leads to a relatively large number of systematic collisions; thus, our DFS= algorithm uses a simple linear hash function instead, computed on a sorted list of the hash values of the physical states.

## The DFS⊆ Algorithm

Our second graph algorithm, DFS⊆, uses the above search procedure with a more complex data structure for finding related belief states. This data structure must maintain a database of all belief states encountered thus far, and support finding both proved supersets and disproved or stack-resident subsets of a query. Although constant-time lookups and updates don't seem to be possible in this setting, we can hope for a data structure that performs reasonably well in practice. To this end, we draw on an extensive empirical evaluation of four different subset/superset query algorithms performed by Helmer and Moerkotte (1999), who concluded that for both subset and superset lookups, a scheme called *inverted files* performs best.

Thus, our implementation of DFS⊆ uses inverted files, maintaining a hash table that maps from each physical state to the set of belief states encountered thus far that contain it. Various lookup schemes are possible within this framework; in our current implementation, we do an optimized version of the following. When a belief state is added to the data structure, a new status object consisting of [$maxDisproof$ $stack$ $minProof$] and the belief-state size is added to lists hashed under each of its constituent physical states. Then, upon receiving a query set at a given depth, we retrieve the lists corresponding to its elements, and count the number of lists in which each status object appears. If a belief state proved at lower remaining depth appears in all of the lists, then it must be a superset of the query and we return it as a proof; likewise, if a belief state on the stack or disproved at greater remaining depth appears in the same number of lists as its size, then it must be a subset of the query and we return it as a disproof or cycle as appropriate.

# DBU for Belief-State AND–OR Graphs

This section describes a graph version of DBU, along with appropriate data structures for finding repeated or related belief states in this setting.

Because DBU works at the level of individual physical states rather than entire belief states, it can gather more information than DFS. For instance, suppose that DBU is called on a belief state with 10 elements, and it incrementally finds a proof covering the first 5 states before discovering a disproof upon adding the sixth one. In the process, it learns that the belief state with five states is solvable, whereas the one with six is unsolvable. In the same situation, DFS would only learn that the entire 10-state belief state is unsolvable. Thus, during its searches DBU learns about both *more proofs* and *more general disproofs* than DFS.

However, this greater potential comes at a cost. First, since DBU-OR is called once for each *physical state* generated, the overhead of doing a belief-state lookup at each step will be substantially higher than for the corresponding versions of DFS. Second, there are algorithmic complications arising from the fact that the subsumption relationships in DBU's trees can change during search, when new physical states are added to existing belief-state nodes. In particular, when a new state is added to a belief-state node, it may become equal to or a superset of some previous nodes, and cease being equal to or a subset of other previous nodes. The

```
/* OR-nodes are now augmented with list of states NEW(b),
 * and a list of moves LOOPY(b), both initially empty.        */

function DBU-OR(b, state, d) returns true/false/stack_d'
   move all moves from LOOPY(b) to the end of MOVES(b)
   add state to NEW(b)
   states ← NEW(b) ∪ STATES(b)
   if (⊇)states proven at depth ≤ d then return true
   if (⊆)states disproven at depth ≥ d then return false
   if (⊆)states on the stack at depth d' then return stack_d'
   set states as on the stack at depth d
   maxstack ← 0
   for each m ∈ MOVES(b) do
      if NULL(CHILD(b)[m]) then
         CHILD(b)[m] ← a new AND-node
      r ←(∀s∈NEW(b))DBU-AND(CHILD(b)[m],SUCC(s,m),d)
      if r = true then
         move all states from NEW(b) to STATES(b)
         set states as off the stack and proved at depth d
         return true
      else if r = stack_d' then
         add m to LOOPY(b)
         maxstack ← MAX(maxstack, d')
      delete m from MOVES(b) and delete CHILD(b, m)
      move all states from STATES(b) to NEW(b)
   set states as off the stack
   if maxstack > d then return stack_maxstack
   set states as disproved at depth d and return false
```

Figure 6: DBU-OR for *graph* search (DBU-AND is unchanged).

first case is not problematic, since the algorithm will automatically discover the new relationships; the second case, on the other hand, will require extra care to ensure correctness.

Figure 6 shows a modified DBU-OR function for graph search. The modifications to DBU-OR are the same as for DFS-OR, with two changes needed to handle the above issue.

The first problem case arises when a new state that is added to an OR-node breaks a previous proving cross-edge. Then, our algorithm must go back to searching for a solution, being careful to start up where it left off. In Figure 6, the list NEW($b$) is used to keep track of the set of states that have not yet been integrated into an explicit proof at node $b$.

The second problem case is a bit more subtle, so we will illustrate it with a concrete example. Suppose that the tree in Figure 4(b) is a portion of a larger tree for some problem instance, which was constructed on a first visit to node **A**. Then, further search at **F** succeeded in finding a plan for its current belief state, and so the algorithm returned to the parent of **A** (not shown). Here it found another physical state that had not yet been considered at **A**. Finally, the algorithm attempted to incorporate this new state into the plan at **A**, and by extension **F**, but failed. At this point, we might incorrectly conclude that **A** is unsolvable with the new physical state added. However, it might be the case that the new physical state broke one of the cycles in **B**'s subtree, so that new acyclic plans are possible at **C** and/or **E**. The change to deal with this case is simple: when a recursive call on a move fails with $stack_d$, that move is pushed onto a special list LOOPY($b$). Then, whenever new states are added to node $b$, these LOOPY moves are returned to MOVES($b$) so that they will be attempted again in this new context.

Our actual implementation includes several improvements in addition to those described for DFS, which for simplicity are omitted in the pseudocode, First, when a proving cross-edge is broken, our implementation searches first on the move that was associated with the previous proof, as a domain-independent heuristic; to support this, our cache data structures are augmented to store a "proving move" along with each proof. Second, rather than discarding "loopy" subtrees and potentially regenerating them every time a belief-state node changes, we keep them around and simply re-check whether the added states have broken any of the cycles. A further optimization keeps track of simple conditions under which a subtree need not be re-checked.

## The DBU= Algorithm

The hash-table data structure described for DFS= can be used unmodified with the new DBU search procedure to yield a "DBU=" algorithm. However, we note that the space requirements for DBU= are linear in the number of *physical states* constructed, rather than belief states (like DFS=).

## The DBU⊆ Algorithm

The inverted file data structure used in DFS⊆ can also be used unmodified to produce a DBU⊆ algorithm. However, unlike in the strict equality case, here there are substantial improvements that can be made, and we have done so.

In particular, we note that whereas the space requirements for DFS⊆ were linear in the total number of physical states constructed, the requirements for DBU⊆ with the same data structure would be much worse. This is because when proving a belief state with $n$ physical states, DBU will first prove a subset of size 1, then a subset of size 2, and so on up to $n$. Naively storing each of these subsets as separate objects would impose quadratic space requirements, as well as slow down queries, since the number of candidate belief states to search through would grow accordingly.

Fortunately, there is a simple solution to this problem, which exploits the nested structure of the belief states considered by DBU. At each step, DBU adds a physical state into a pre-existing belief-state node, which must already correspond to an object in the cache. Rather than constructing a new cache entry for the augmented belief state, we can simply extend the old entry to cover the new state.

This can be implemented by extending the status objects to have forms such as $((s_1, 3, m_1), (s_2, 5, m_2), (s_3, 3, disproof))$, which compactly conveys that belief state $\{s_1\}$ is solvable at depth 3 by move $m_1$, $\{s_1, s_2\}$ is solvable at depth 5 by move $m_2$, and $\{s_1, s_2, s_3\}$ is unsolvable at depth 3 (the special value *stack* is allowed in this last position as well). Each of the tuples $(s, d, m)$ has a pointer to the entire status object, and is hashed on a list keyed by $s$. This "telescoped" representation is efficient to store and update; most cache operations simply require adding a new tuple to an existing status object, and then pushing this tuple onto the appropriate index list. Finally, by a fairly simple extension of the procedure described for DFS⊆, this data structure can be efficiently queried to find relevant subsets and supersets during search.

## Experiments

This section evaluates our six search algorithms (DFS and DBU, each in baseline, "=", and "⊆" varieties) on two different tasks: nondeterministic vacuum world planning problems and Kriegspiel (partially observable chess) checkmate problems. For simplicity, the search algorithms are compared without heuristics; however, we expect that observed speedups should generalize to other settings as well. To prevent systematic biases in our results, all arbitrary choices (e.g., move orderings) are randomized.

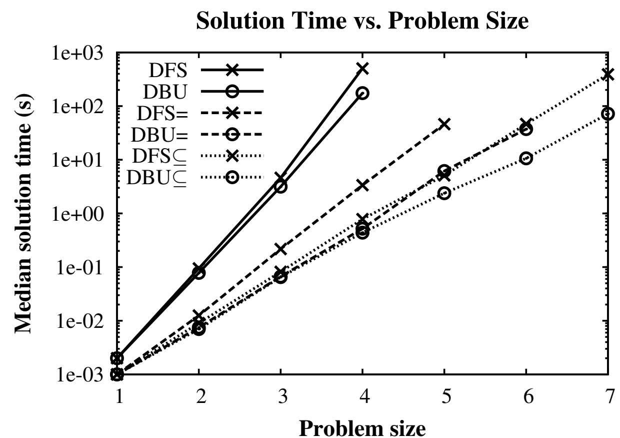
### Vacuum-World Planning Problems

Our first set of test problems are drawn from a partially observable, nondeterministic vacuum world domain (a variant of that in (Russell & Norvig 2003)). An agent is situated in a $w \times h$ grid in which some squares may be dirty, only its current square is observable, and the goal is to make all of the squares clean. The available actions are *left*, *right*, *up*, *down*, and *suck*. *suck* unconditionally makes the current square clean, whereas the move actions always succeed but are only applicable when they lead to valid squares. The catch is that the agent is malfunctioning, so that when it goes *down* or *right* it may cause the source square to become dirty, without knowing whether or not this has occurred. This is a good test domain because its problems instances are simple, involve nontrivial nondeterminism and partial observability while still admitting guaranteed plans, and vary widely in difficulty (primarily with the size of the world).

Our first experiments considered boards of size $2 \times h$ for heights $h$ ranging from 1 to 7, where the goal was always for the agent to leave the board clean, starting from a singleton initial belief state with the agent in the upper-left square and only the bottom-right square dirty. The depth limit was always set to the length of the optimal solution, $3*h+1$. Each algorithm was tested 20 times on problems of each size, and the run-time and number of physical states constructed were recorded for each run. Trials taking more than $10^4$ seconds or 400 MB of memory were halted early. The median results of these runs are depicted in Figure 7. The graph shows median run time (in log scale) as a function of problem size, for each algorithm. The table shows the same results, limited to problem sizes 4-6 but also including the median number of physical states constructed.

The performance of the algorithms varies widely, spanning 3-4 orders of magnitude even on the fairly small $2 \times 4$ problem instances. Some kind of repeated state detection is clearly necessary for good performance: the baseline algorithms both run out of time on problems of size 5 and greater. DFS= and DBU= perform better, until they run out memory at sizes 6 and 7 (resp.). Finally, DFS⊆ and DBU⊆ are able to scale successfully to $2 \times 7$ problems (and beyond).

We are most interested in comparing our 3 new algorithms, DFS⊆, DBU=, and DBU⊆, to the best previously existing algorithm, DFS=. Thus, we will contrast these four algorithms' quantitative performance on $2 \times 5$ problems, the largest that all four could solve. A first thing to note is that all else being equal, subset testing and DBU are preferable to equality testing and DFS; indeed, DBU⊆ is about 20 times faster than DFS=, and constructs nearly 80 times




| Size | 4 | | 5 | | 6 | |
|---|---|---|---|---|---|---|
| | Time | States | Time | States | Time | States |
| **DFS** | 502.3 | 49036K | * | * | * | * |
| **DFS=** | 3.4 | 257K | 46.1 | 3961K | ** | ** |
| **DFS⊆** | 0.8 | 36K | 5.1 | 309K | 46.4 | 3023K |
| **DBU** | 174.6 | 5892K | * | * | * | * |
| **DBU=** | 0.5 | 11K | 6.2 | 117K | 37.3 | 631K |
| **DBU⊆** | 0.4 | 10K | 2.4 | 52K | 10.6 | 217K |

Figure 7: Performance of 6 algorithms on $2 \times h$ vacuum-world problems, for various problem sizes $h$, with a depth limit equal to the optimal solution length. All values are medians over 20 runs. Runtimes are in seconds. Top: solution time (in log scale) as a function of problem size, for each of the six algorithms. Bottom: table of results, for a subset of problem sizes. "States" is the total number of physical states constructed. (*) and (**) indicate that a run exceeded the 10,000s time limit or 400MB memory limit, resp.

fewer physical states (which might be a concern if physical state operations were expensive, unlike in this domain).

Delving a bit deeper, we see that simply switching from DFS= to DFS⊆ improves performance by about a factor of 10, in terms of both runtime and physical states. This in itself might be surprising; even more so is the fact that subset testing seems to carry only about 25% greater overhead than equality testing, measured by comparing the ratios of *physical states examined / runtime*. DBU= performs nearly as well as DFS⊆ in terms of runtime, and nearly 3 times better in terms of physical states examined; DBU⊆ improves upon this by a further factor of 2.5.

Finally, we briefly review some additional statistics. Across the board, about 30% of lookups with equality testing returned results, whereas for subset testing this number was closer to 60%. In both cases, less than 0.5% of the hits were proofs, 50-80% were disproofs, and the rest were cycles. These proportions are at least partially explained by the depth limit, which causes there to be many more $false$ physical states than $true$ ones in the belief-state trees.

### Kriegspiel Checkmate Problems

Our second set of experiments used checkmate problems drawn from the game of Kriegspiel, a partially observable variant of chess in which each player cannot see the pieces or moves of her opponent, but instead receives limited per-

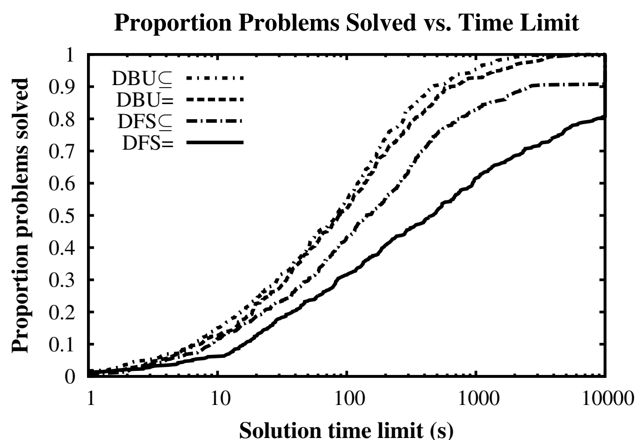



Figure 8: Results on solvable 7-ply Kriegspiel checkmate problems. The $y$-axis shows the fraction of problems solved within a given amount of CPU time (in log scale). The algorithms are listed in *decreasing* order of efficiency.

cepts about captures, checks, and illegal move attempts.[4]

In particular, we used a database of 500 7-ply Kriegspiel checkmate problems, which span a wide range of difficulty levels.[5] While 7-ply problems might sound trivial, they are actually quite difficult due to the *try-until-feasible* property: in Kriegspiel, a player must keep attempting moves on her turn until one succeeds. Thus, the worst-case branching factor for *a single ply* is *factorial* in the number of potentially legal actions, which is typically between 10 and 30.

Figure 8 shows the performance of our algorithms on this database, graphed as the proportion of problems solved within a given run-time limit. DFS and DBU are not shown, as they could only solve a few percent of the problems within the time limit. DBU= and DBU⊆ solved all 500 problems, whereas DFS= failed to solve about 20% solely due to the $10^4$ second time restriction, and DFS⊆ failed to solve about 10% solely due to the 400 MB memory restriction.

The results are qualitatively quite similar to the previous set; again, all else being equal, subset testing and DBU are better, and the differences between the algorithms are more pronounced on more difficult problems. For problems in the 50th percentile of difficulty we see about a factor of 10 speedup over the best previously existing algorithm, and by the 80th percentile this factor has grown to about 50.

## Conclusions and Future Work

We have proposed several new sound and complete algorithms for belief-state AND–OR search, and shown them to yield more than an order of magnitude performance improvement on two very different test domains. However, there is still much work to be done.

A first possibility is to investigate the individual contributions of cached proofs, cached disproofs, and cycle checking to the performance improvements observed, and use this data to develop alternative subset lookup schemes. Because memory is a primary limitation in scaling these algorithms to larger problems, one might also consider removing subsumed entries from the cache data structures, and/or simply taking a memory-bounded approach and forgetting as necessary when memory becomes scarce.

Another direction is to consider utilizing related belief states in a *domain-independent heuristic* for move ordering. For example, a move that led to a proof on a *subset* (or intersecting set) of the current belief state seems more promising than a move that led to a disproof on a *superset*, and should perhaps be tried first.

A final possibility is to investigate applications to symbolic belief state representations (e.g., BDDs) and other search algorithms (e.g., Proof-Number Search (Allis 1994)), as well as extensions for probabilistic planning.

## Acknowledgments

This research was supported in part by the DARPA REAL program, award FA8750-04-2-0222. We would also like to thank the anonymous reviewers for their helpful comments.

## References

Allis, L. V. 1994. *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. Dissertation, University of Limburg.

Amir, E., and Russell, S. 2003. Logical filtering. In *IJCAI*, 75–82.

Bayardo, Jr., R. J., and Schrag, R. C. 1997. Using CSP look-back techniques to solve real-world SAT instances. In *AAAI*, 203–208.

Bertoli, P.; Cimatti, A.; Roveri, M.; and Traverso, P. 2001. Planning in nondeterministic domains under partial observability via symbolic model checking. In *IJCAI*, 473–478.

Breuker, D. 1998. *Memory versus Search in Games*. Ph.D. Dissertation, Universiteit Maastricht.

Campbell, M. 1985. The graph-history interaction: on ignoring position history. In *Proc. ACM*, 278–280.

Ciancarini, P.; Libera, F. D.; and Maran, F. 1997. Decision Making under Uncertainty: A Rational Approach to Kriegspiel. In *Advances in Computer Chess 8*, 277–298.

Helmer, S., and Moerkotte, G. 1999. A study of four index structures for set-valued attributes of low cardinality. Technical Report TR-1999-002, University of Mannheim.

Hoffmann, J., and Koehler, J. 1999. A new method to index and query sets. In *IJCAI*, 462–467.

Kurien, J.; Nayak, P.; and Smith, D. 2002. Fragment-based conformant planning. In *AIPS*, 153–162.

Russell, S., and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition.

Russell, S., and Wolfe, J. 2005. Efficient belief-state AND-OR search, with application to Kriegspiel. In *IJCAI*, 278–285.

Sakuta, M., and Iida, H. 2000. Solving Kriegspiel-like problems: Exploiting a transposition table. *ICGA Journal* 23(4):218–229.

[4]These problems are modeled as single-agent planning problems by treating the opponent's turns as situations in which our agent has only one available "no-op" action, which has nondeterministic effects corresponding to the opponent's possible moves.

[5]More information on Kriegspiel and this database is online at `http://cs.berkeley.edu/~jawolfe/kriegspiel/`.