State-Space Search, Problem Reduction, and Iterative Deepening: A Comparative Analysis

Dana S. Nau*
Computer Science Department, and Institute for Systems Research
University of Maryland
College Park, Maryland 20742
Email: nau@cs.umd.edu

Abstract

In previous work, Korf showed that by introducing one problem-reduction step into a state-space search, one could reduce the number of node generations from $O((2b)^{2d})$ to $O(b^d)$, where b and d are the branching factor and search depth. My results are as follows:

1. The $O(b^d)$ bound is tight, but the $O((2b)^{2d})$ bound is not: the A* procedure does only $\Theta(b^{2d})$ node generations. Thus, the improvement produced by one problem-reduction step is not always as great as the previous results might suggest.
2. In an AND/OR tree where multiple problem-reduction steps are possible, problem reduction produces a much more dramatic improvement: both the time complexity and the space complexity decrease from doubly exponential to singly exponential.
3. For iterative-deepening procedures like IDA* that only remember the nodes on the current path, the space complexity decreases but the time complexity increases—by exponential amounts in Korf's model, and doubly exponential amounts in the AND/OR-tree model. This is true even for IDAO*, a new procedure that improves IDA*'s performance by combining it with problem reduction.

These results lead to the following conclusions:

- In general, problem reduction can save huge amounts of both time and space.
- Whether to use a procedure that remembers every node it has visited, or instead use a limited-memory iterative-deepening procedure, depends on whether the primary objective is to save space or save time.

*This work was supported in part by NSF Grants IRI-8907890 and NSFD CDR-88003012.

1 Introduction

In his paper, "Planning as Search" [3], Korf showed that by introducing one problem-reduction step into a state-space search, one could reduce the number of node generations (which is polynomially related to the time complexity) from $O((2b)^{2d})$ to $O(b^d)$, where b was the branching factor of the search space, and d was the search depth. Yang, Nau, and Hendler [12] later extended Korf's result by showing that even if the subproblems produced by the problem-reduction step were not completely independent, in some cases one could still achieve the same big-O reductions.

Although interesting, these results were not completely conclusive, for two reasons:

1. It was not clear how tight the big-O bounds might be. For example, even if state-space search and problem-reduction search both did $\Theta(b^d)$ node generations, this would still be consistent with the big-O bounds.

2. Since these bounds were based on a model in which only one problem-reduction step was possible, this left it unclear how much savings one might achieve by doing multiple levels of problem reduction.

This paper presents an analysis of state-space search vs. problem reduction that addresses both of these issues. My results are as follows.

State-Space Search vs. Problem Reduction. If only one problem-reduction step is allowed, then the $O(b^d)$ bound for problem-reduction search is tight, but the $O((2b)^{2d})$ bound for state-space search is not: the actual figure depends on which state-space search procedure we use. For the IDA* procedure [2], the bound is nearly tight—IDA* does somewhere between $\Omega((2b)^{2d}/\sqrt{d})$ and $O((2b)^{2d})$ node generations. But the A* procedure [10] does much better: only $\Theta(b^{2d})$ node generations. Thus, even though problem-reduction search still produces an exponential-time improvement over state-space search, the improvement is not always as big as the previous results had suggested.

Rather than allowing only one problem-reduction step, suppose we search a binary AND/OR tree in which there are d problem-reduction steps: one before each state-space step. In this case, problem reduction produces a much more impressive improvement. Rather than merely reducing the time complexity from one exponential function to a smaller one (as happens above), problem reduction reduces both the time complexity and the space complexity from doubly exponential to singly exponential.

Limited-Memory Search vs. Full-Memory Search. It is well known that by using a limited-memory iterative-deepening search procedure such as IDA*, one can make a large reduction in the number of nodes stored (which is polynomially related to the space complexity). Furthermore, by combining limited-memory iterative deepening with problem reduction (in a new search procedure called IDAO*), we can achieve the same space complexity as IDA*, while reducing IDA*'s time complexity by an exponential amount (in Korf's model) or a doubly exponential amount (in the AND/OR-tree model).

However, there is a tradeoff: the limited-memory procedures (IDA* and IDAO*) take much more time than the corresponding full-memory procedures (A* and AO*, respectively). In Korf's model the time increase is exponential, and in the AND/OR tree model it is doubly exponential. The reason for this is that the limited-memory procedures do many re-explorations of paths or subtrees that the full-memory procedures explore only once.

This paper is organized as follows. Section 2 defines the basic terms, briefly discusses A* and AO*, and defines IDA* and IDAO*. Section 3 analyzes all four procedures in the case where only one level of problem reduction is possible. Section 4 analyzes them using the AND/OR-tree model, in which d levels of problem reduction are possible. Section 5 compares all of these results, and Section 6 contains concluding remarks.

2 Preliminaries

2.1 Basic Definitions

Let S be any state space, and let s be its start node. If u is an arbitrary node of S, then the *depth* of u (denoted by depth(u)) is the length of the shortest path from s to u. The *height* of S (denoted by height(S)) is the depth of S's deepest node. A *complete path* of S is a path from s to a terminal node of S. A *solution path* from a node u is a path from u to a goal node. A *complete solution path* is a path from s to a goal node. nodes(S), leaves(S), and paths(S), respectively, denote the number of nodes, leaf nodes, and complete paths in S.

Suppose we search S using a search procedure $\mathcal{P}$. Then gen($\mathcal{P}, S$) denotes the total number of node generations done by $\mathcal{P}$, and storage($\mathcal{P}, S$) denotes the largest number of nodes $\mathcal{P}$ will need to store at any one time.

If S_1 and S_2 are any two state spaces, then their *composition* is the state space $R = S_1 \bullet S_2$ defined as follows. The node set of R is $\{\langle v_1, v_2\rangle : v_1, v_2$ are nodes of S_1, S_2, respectively$\}$. In particular, R's start node is $r = \langle s_1, s_2\rangle$, where s_1 and s_2 are S_1's and S_2's start nodes, respectively. If $v = \langle v_1, v_2\rangle$ is a node of R, then v's children are all nodes $\langle w_1, v_2\rangle$ such that w_1 is a child of v_1 in S_1, plus all nodes $\langle v_1, w_2\rangle$ such that w_2 is a child of v_2 in S_2. Alternatively, $S_1 \bullet S_2$ can be thought of as an AND/OR graph ao(R), consisting of a start node r, the state spaces S_1 and S_2, and an AND-branch from r to their start nodes s_1, s_2.

Example. Let S_1 and S_2 be complete binary trees of height 2, as shown in Figs. 1(a) and 1(b); and let $R = S_1 \bullet S_2$. Then R is the graph shown in Fig. 1(c), and ao(R) is the AND/OR tree shown in Fig. 1(d).

2.2 A* and IDA*

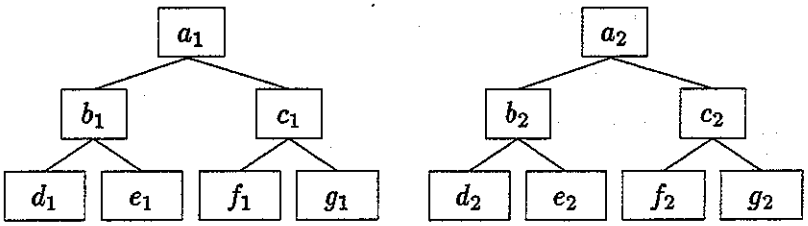
The best-known heuristic state-space search procedure is A* [10], which is guided by a heuristic function $h(u)$ that gives a lower bound on the cost of the least-cost path from the node u to a goal node. A* is basically a best-first branch-and-bound procedure [8, 4], so any solution path returned by it is guaranteed to be a least-cost solution path. Since A* is well known, I will not describe it further here.

Suppose the state space S is a complete b-ary tree of height d, such that every goal node has depth d, and every arc has cost 1. Suppose we search S using A* with the heuristic function $h \equiv 0$. Then A* will generate each node of S once, so

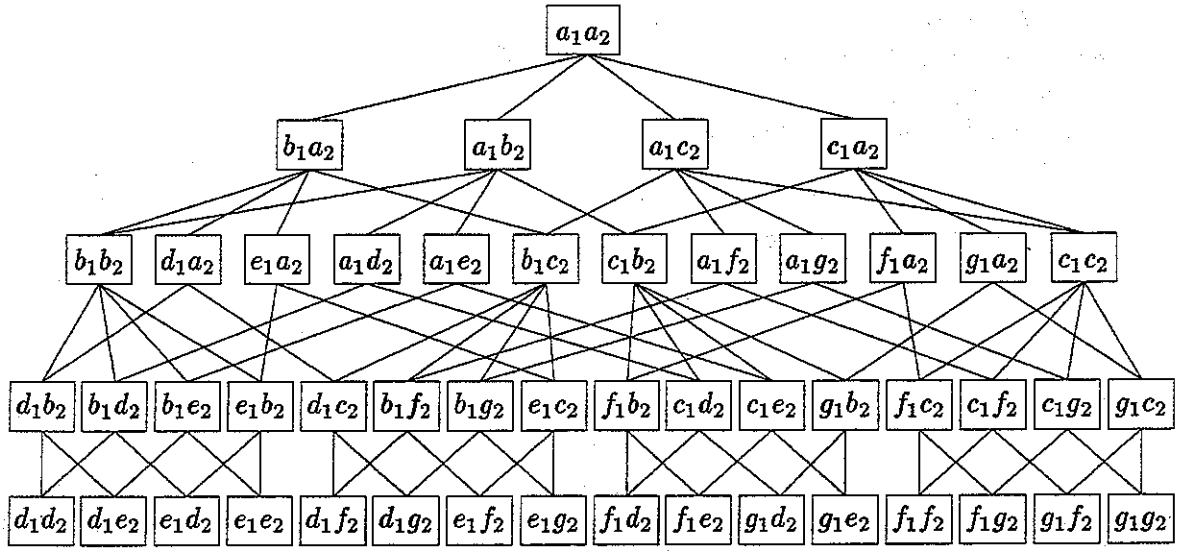
$$\text{gen}(\text{A}^*, S) = \Theta(b^d). \tag{1}$$

Since A* stores every node it generates,

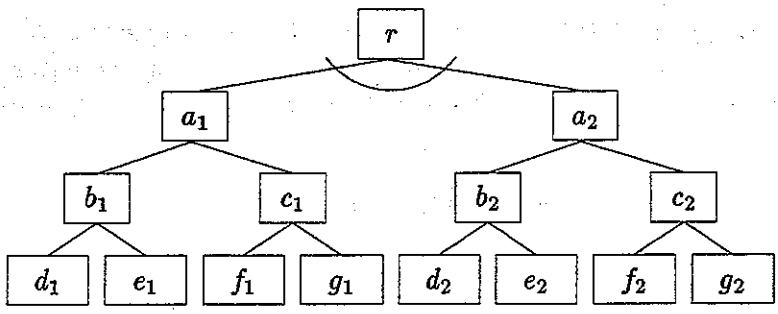
$$\text{storage}(\text{A}^*, S) = \Theta(b^d). \tag{2}$$



(a): the state space S_1. (b): the state space S_2.



(c): the state space $R = S_1 \bullet S_2$.



(d): the AND/OR graph ao(R).

Figure 1: Two binary trees, and the state space and AND/OR tree produced by composing them.

```
procedure IDA*
    global k, k'
    k := h(s), where s is the start node
    k' := ∞
    loop
        P := the path containing only s
        depth-first(P)
        k := k'
    repeat
end IDA*

procedure depth-first(P)
    c := the sum of P's arc costs and the h-value of P's last node
    if c > k, then k' := min(k', c)
    else if P's last node is a goal node, then
        exit from IDA*, returning P
    else for every child v of P's last node, do
        P' := the path formed by appending v to P
        depth-first(P')
    end
end depth-first
```

Figure 2: Pseudocode for IDA*.

Korf [2] developed the IDA* procedure in order to improve on A*'s space complexity. Basically, IDA* performs a depth-first search, backtracking whenever it finds a path whose costs exceeds a cutoff value k. It repeats this search for larger and larger values of k, until it finds a solution. Fig. 2 shows a pseudocode version of IDA*.

There are worst-cases in which IDA*'s time complexity is much worse than A*'s [6, 5]. However, there are cases in which it has the same time complexity as A*, and much better space complexity than A*. In particular, it is easy to show [2] that if we use IDA* with $h \equiv 0$ to search the state space S defined above, then

$$\text{gen}(\text{IDA}^*, S) = \Theta(b^d); \tag{3}$$

$$\text{storage}(\text{IDA}^*, S) = \Theta(d). \tag{4}$$

2.3 AO* and IDAO*

AO* [10] (and other similar procedures [9, 7]) are analogues of A* that do problem-reduction search on an AND/OR graph G. In an AND/OR graph, the concept of a path from a node u is replaced by that of an *AND-tree* rooted at u, which is formed by starting at u, and recursively following one edge at each OR-branch and all edges at each AND-branch. A *solution tree* T rooted at u is an AND-tree rooted at u such that every leaf node of T is a goal node. A solution for G is any solution tree rooted at G's start node.

```
procedure IDAO*
    global k, k'
    k := h(s), where s is the start node
    k' := ∞
    loop
        T := the AND-tree containing only s
        depth-first(T)
        k := k'
    end
end IDAO*

procedure depth-first(T)
    c := the sum of T's arc costs and the h-values of T's tip nodes
    if c > k, then k' := min(k', c)
    else if every tip node of T is a goal node, then
        exit from IDAO*, returning T
    else do
        let u be T's leftmost non-goal tip node
        if the branch B from u is an AND-branch, then
            T' := the AND-tree formed by attaching B to T
            depth-first(T')
        else for every child v of u, do
            T' := the AND-tree formed by attaching (u, v) to T
            depth-first(T')
        end
    end depth-first
```

Figure 3: Pseudocode for IDAO*.

Like A*, AO* is guided by a heuristic function $h(u)$; in this case, $h(u)$ gives a lower bound on the cost of the least-cost solution tree rooted at u. Like A*, AO* is basically a best-first branch-and-bound procedure [8], and thus the first time that it finds a solution for G, this guaranteed to be the least-cost solution. Since AO* is well known, I will not describe it further in this paper.

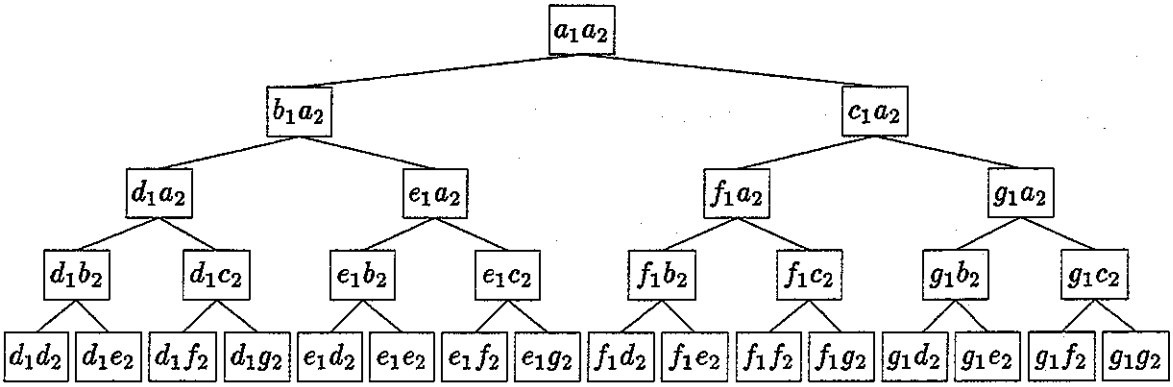
Suppose the AND/OR graph G is a complete b-ary tree of height d, such that every goal node has depth d, and every arc has cost 1. Suppose we search G using AO* with the heuristic function $h \equiv 0$. Then AO* will generate every node of G, so

$$\text{gen}(\text{AO*}, G) = \Theta(b^d). \tag{5}$$

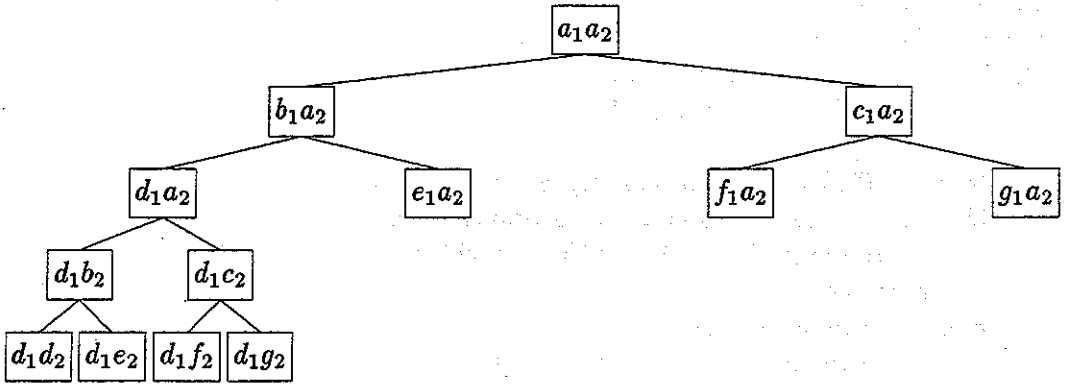
Since AO* stores every node it visits,

$$\text{storage}(\text{AO*}, G) = \Theta(b^d). \tag{6}$$

AO*'s space complexity can be improved by defining a new search procedure that is analogous to IDA* but also does problem reduction. This procedure, which I call IDAO* (Iterative Deepening



(a): the case where d_1, e_1, f_1, g_1 are all goals.



(b): the case where d_1 is a goal, but e_1, f_1, g_1 are not.

Figure 4: Two different possibilities for serialize(R), where R is as shown in Fig. 1.

AO*), is defined in Fig. 3. Like IDA*, IDAO* performs a depth-first search, backtracking whenever the accumulated cost exceeds a cutoff value k; and it repeats this search for larger and larger values of k until it finds a solution. However, the entities that it generates during this search are not paths as in IDA*, but instead are AND-trees.

Basically, what IDAO* does is to serialize the subgoals in each AND-tree, and then do an IDA*-style search of them. For example, suppose IDAO* is examining an AND-tree that contains an AND-branch from some node u to two other nodes u_1 and u_2. If u_1 is not a goal node, then IDAO* will not even try to solve u_2 until it has found a solution tree for u_1. Thus, IDAO* generates nodes of the AND/OR graph G in the same order that IDA* generates nodes of the state-space graph serialize(ss(G)), which can be described as follows.

Since each AND-branch of G corresponds to a composition operation, this means that G corresponds to the state-space graph ss(G) produced by doing one composition operation for each AND-branch of G. For example, if G is the AND/OR graph shown in Fig. 1(d), then ss(G) is the state space shown in Fig. 1(c). Thus, some or all of the nodes of ss(G) may be composed of "subnodes" from other, smaller state spaces (for example, the node $\langle a_1, a_2 \rangle$ of Fig. 1(c) has subnodes a_1 and a_2). From the definition of the composition operation, it follows that if u is such a composite

node, then u's children in ss(G) are produced by replacing subnodes of u with the children of those subnodes. Of these children of u, the only ones that correspond to serializing G's subgoals (and hence the only ones that we include in serialize(ss(G))) are the ones we get by replacing the first subnode that is not a goal node. For example, in Fig. 1(c), the first nongoal subnode of $\langle a_1, a_2 \rangle$ is a_1, so the tree serialize(R) (see Fig 4) includes $\langle b_1, a_2 \rangle$ and $\langle c_1, a_2 \rangle$ but not $\langle a_1, b_2 \rangle$ and $\langle a_1, c_2 \rangle$.

IDAO* generates nodes of G in the same order that IDA* generates nodes of serialize(ss(G)), so

$$\text{gen}(\text{IDAO}^*, G) = \Theta(\text{gen}(\text{IDA}^*, \text{serialize}(\text{ss}(G)))); \quad (7)$$

$$\text{storage}(\text{IDAO}^*, G) = \Theta(\text{storage}(\text{IDA}^*, \text{serialize}(\text{ss}(G)))). \quad (8)$$

The above equations will be useful for analyzing IDAO*'s performance.

3 State-Space Search vs. One-Step Problem Reduction

Let the state space S be a complete b-ary tree of height d such that every arc has cost 1 and goal node has depth d.[1] Then every complete solution path has length d. Let S_1 and S_2 be two disjoint copies of S, and let $R = S_1 \bullet S_2$ (for example, see Fig. 1). Then R is a state space in which one level of problem reduction is possible. R has the following properties, which will be useful in analyzing how search procedures behave on it.

$$\text{height}(R) = \text{height}(S_1) + \text{height}(S_2) = 2d; \quad (9)$$

$$\text{leaves}(R) = \text{leaves}(S_1)\text{leaves}(S_2) = (b^d)^2 = b^{2d}; \quad (10)$$

$$\text{nodes}(R) = \text{nodes}(S_1)\text{nodes}(S_2) = \left(\frac{b^{d+1}-1}{b-1}\right)^2 = \Theta(b^{2d}). \quad (11)$$

Every complete path P of R is produced by interleaving a complete path P_1 of S_1 and a complete path P_2 of S_2. Since P_1 and P_2 each have length d, the number of possible ways to interleave them is $\binom{2d}{d}$. Using Stirling's approximation [1], it is easy to show that this is $\Theta(2^{2d}/\sqrt{d})$. Thus,

$$\text{paths}(R) = \text{leaves}(R)\binom{2d}{d} = b^{2d}\Theta\left(\frac{2^{2d}}{\sqrt{d}}\right) = \Theta\left(\frac{(2b)^{2d}}{\sqrt{d}}\right). \quad (12)$$

Now, suppose we do a heuristic search of R, using the heuristic function $h \equiv 0$. Below, we consider four cases:

Case 1. Suppose we do a state-space search, using a procedure like A* that keeps track of every node that it generates. A* will expand each non-leaf node of R once, so

$$\text{gen}(\text{A}^*, R) = \Theta(\text{nodes}(R)) = \Theta(b^{2d}). \quad (13)$$

Since A* stores every node that it generates,

$$\text{storage}(\text{A}^*, R) = \Theta(\text{nodes}(R)) = \Theta(b^{2d}). \quad (14)$$

[1] Thus every goal node is a leaf node, and this simplifies the derivation of some of the complexity bounds in Eqs. 13–23. However, it is straightforward to show that those complexity bounds are still correct if we generalize the model to allow height(S) $\geq d$, where d is the depth of the shallowest goal node.

Case 2. Suppose we do a state-space search, using a limited-memory iterative-deepening procedure a procedure like IDA* that only keeps track of the nodes on the path it is currently exploring. During each iteration of its loop, IDA* will generate each node u once for each path that it finds to u, so

$$\text{gen}(\text{IDA}^*, R) = \Omega(\text{paths}(R)) = \Omega\left(\frac{(2b)^{2d}}{\sqrt{d}}\right). \tag{15}$$

Every node of R has $2b$ children, and IDA* behaves as if R were a tree, so IDA* does no more node generations than it would do on a complete $2b$-ary tree of depth $2d$. Thus from Eq. 3,

$$\text{gen}(\text{IDA}^*, R) = O((2b)^{2d}). \tag{16}$$

Like a depth-first search, IDA* stores only the nodes on the current path. Thus since every complete path of R has length $2d$,

$$\text{storage}(\text{IDA}^*, R) = \Theta(d). \tag{17}$$

Case 3. Suppose we do a problem-reduction search, using a procedure like AO* that keeps track of every node that it generates. AO* will generate each node of $\text{ao}(R)$ once, so

$$\text{gen}(\text{AO}^*, R) = \Theta(\text{nodes}(S_1) + \text{nodes}(S_2)) = \Theta(b^d). \tag{18}$$

Since AO* stores every node that it generates,

$$\text{storage}(\text{AO}^*, R) = \Theta(\text{nodes}(S_1) + \text{nodes}(S_2)) = \Theta(b^d). \tag{19}$$

Case 4. Suppose we do a problem-reduction search, using a limited-memory iterative-deepening procedure a procedure like IDAO* that only keeps track of the nodes on the AND-tree that it is currently exploring. As discussed in Section 2.3, IDAO* will generate nodes in $\text{ao}(R)$ in the same order that IDA* would generate nodes in the state-space graph $\text{serialize}(R)$. The worst case is if every leaf node of S_1 is a goal node. In this case, $\text{serialize}(R)$ is a b-ary tree of height $2d$, as illustrated in Fig. 4(a). Thus from Eqs. 3–4 and 7–8,

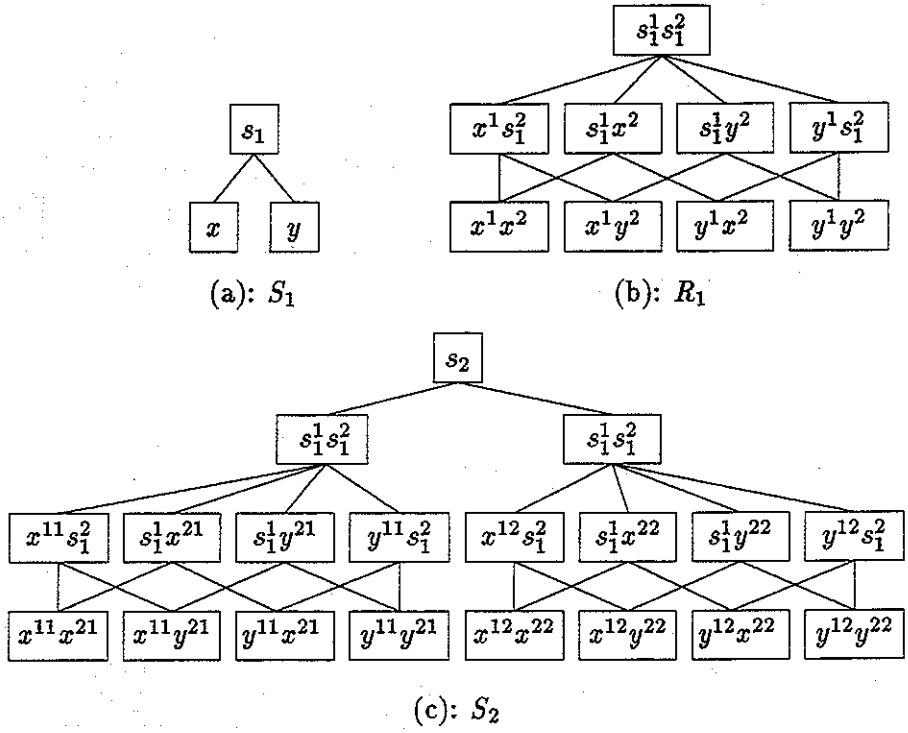
$$\text{gen}(\text{IDAO}^*, R) = O(\text{gen}(\text{IDA}^*, \text{serialize}(R))) = O(b^{2d}); \tag{20}$$

$$\text{storage}(\text{IDAO}^*, R) = O(\text{storage}(\text{IDA}^*, \text{serialize}(R))) = O(d). \tag{21}$$

The best case is if only one leaf node of S_1 is a goal node. In this case, $\text{serialize}(R)$ is the concatenation of two complete b-ary trees of height d, as illustrated in Fig. 4(b). If we explore this tree using IDA*, there will be $2d$ complete iterations of IDA*'s loop, plus one more iteration during which IDA* finds an answer and exits. During the first d iterations, IDA* only generates nodes in the top half of the tree. During each subsequent iteration, IDA* will generate all of the $\Theta(b^d)$ nodes in the top half of the tree, plus one or more nodes in the bottom half of the tree. Thus,

$$\text{gen}(\text{IDAO}^*, \text{serialize}(R)) = \Omega(db^{2d}); \tag{22}$$

$$\text{storage}(\text{IDAO}^*, R) = \Omega(d). \tag{23}$$



(a): S_1 (b): R_1

(c): S_2

Figure 5: The state spaces S_1, R_1, and S_2. The state space R_2 (not shown here) is a graph of height 6, with 64 leaf nodes.

4 State-Space Search vs. Multi-Step Problem Reduction

The previous section discussed what happens when only one problem-reduction step is possible. This section discusses what happens if it is possible to do multiple problem-reduction steps—one between each application of a state-space search operator. For $d = 1, 2, \ldots$, let the state spaces R_d and S_d be defined recursively as follows (see Fig. 5):

- S_1 is a binary tree of height 1 in which every arc has cost 1.
- Let S_d^1, S_d^2 be two disjoint copies of S_d. Then $R_d = S_d^1 \bullet S_d^2$.
- Let R_d^1, R_d^2 be two disjoint copies of R_d, and r_d^1, r_d^2 be their start nodes. Then S_{d+1} contains all nodes and edges of R_d^1 and R_d^2, together with a start note s_{d+1} and two new edges (s_{d+1}, r_d^1) and (s_{d+1}, r_d^2), each of cost 1.

Then for each d, ao(S_d) and ao(R_d) are AND/OR trees of height $2d-1$ and $2d$, respectively, as illustrated in Fig. 6. In addition, S_d and R_d have the following properties, which will be useful in analyzing how search procedures behave on them.

$$\text{height}(S_1) = 1; \tag{24}$$

$$\text{height}(R_d) = 2(\text{height}(S_d)); \tag{25}$$

$$\text{height}(S_{d+1}) = \text{height}(R_d)) + 1 = 2\text{height}(S_{d-1}) + 1; \tag{26}$$

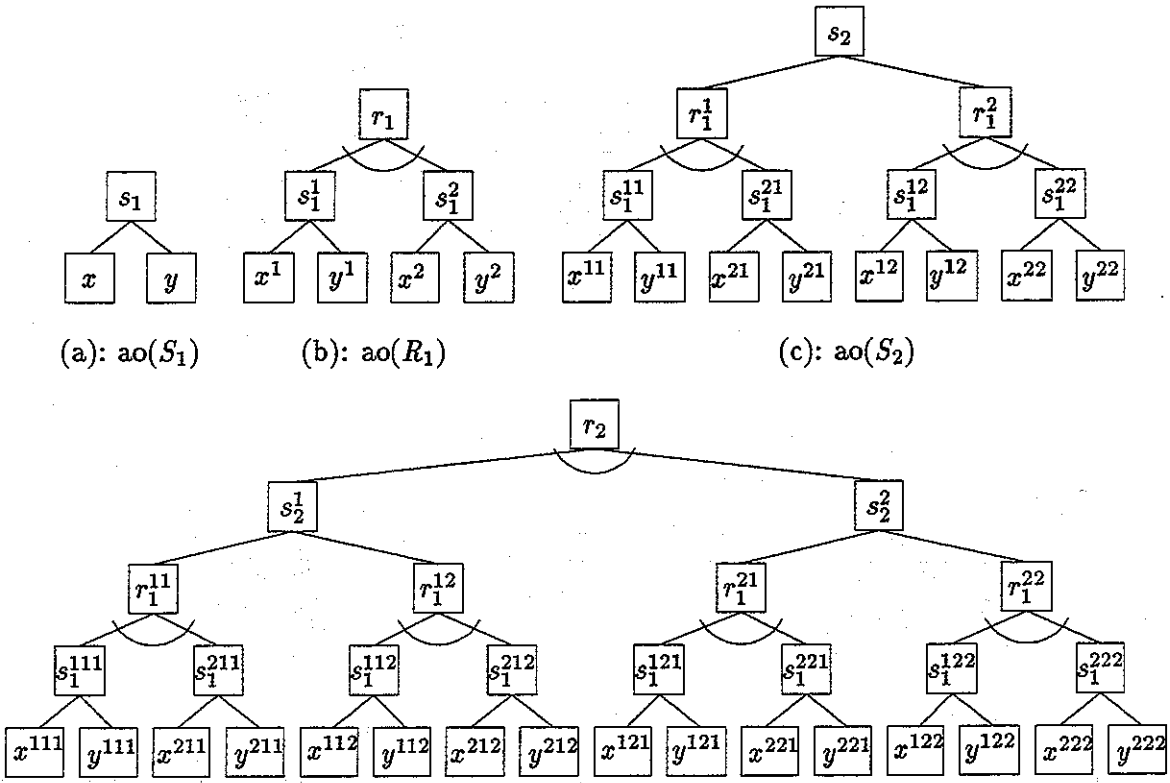


Figure 6: The AND/OR trees ao(S_1), ao(R_1), ao(S_2), and ao(R_2).

$$\begin{aligned}
\text{leaves}(S_1) &= 2; && (27)\\
\text{leaves}(R_d) &= (\text{leaves}(S_d))^2; && (28)\\
\text{leaves}(S_{d+1}) &= 2\text{leaves}(R_d); && (29)
\end{aligned}$$

$$\begin{aligned}
\text{nodes}(S_1) &= 3; && (30)\\
\text{nodes}(R_d) &= (\text{nodes}(S_d))^2; && (31)\\
\text{nodes}(S_{d+1}) &= 2(\text{nodes}(R_d)) + 1 = 2(\text{nodes}(R_d))^2 + 1. && (32)
\end{aligned}$$

Solving these recurrence relationships, we get

$$\begin{aligned}
\text{height}(S_d) &= 2^d - 1; && (33)\\
\text{height}(R_d) &= 2^{d+1} - 2; && (34)
\end{aligned}$$

$$\begin{aligned}
\text{leaves}(S_d) &= 2^{2^d-1} = \Theta(2^{2^d}); && (35)\\
\text{leaves}(R_d) &= 2^{2^{d+1}-2} = \Theta(4^{2^d}); && (36)
\end{aligned}$$

$$\text{nodes}(S_d) \approx \Theta(2.48^{2^d}); \tag{37}$$

$$\text{nodes}(R_d) \approx \Theta(6.17^{2^d}). \tag{38}$$

For every i, every complete path of S_d or R_d has a length of height(S_d) or height(R_d), respectively. Every complete path of R_d is produced by interleaving a complete path P_d^1 of S_d^1 and a complete path P_d^2 of S_d^2. Since the length of each of these paths is height(S_d), the number of possible ways to interleave them is

$$\binom{2\text{height}(S_d)}{\text{height}(S_d)} = \Theta\left(\frac{2^{2\text{height}(S_d)}}{\sqrt{\text{height}(S_d)}}\right) = \Theta\left(\frac{2^{2^{d+1}-2}}{\sqrt{2^d-1}}\right) = \Theta(2^{2^{d+1}-d/2}). \tag{39}$$

Thus,

$$\begin{aligned} \text{paths}(R_d) &= \text{leaves}(R_d)\Theta(2^{2^{d+1}-d/2}) \\ &= \Theta(2^{2^{d+1}})\Theta(2^{2^{d+1}-d/2}) \\ &= \Theta(2^{2^{d+2}-d/2}). \end{aligned} \tag{40}$$

Now, suppose we do a heuristic search of R_d, using the heuristic function $h \equiv 0$. Below, we consider four cases:

Case 1. Suppose we do a state-space search, using a procedure like A* that keeps track of every node that it generates. A* will expand each non-leaf node of R_d once, so the total number of node generations will be

$$\text{gen}(\text{A*}, R_d) = \Theta(\text{nodes}(R_d)) \approx \Theta(6.17^{2^d}). \tag{41}$$

Since A* stores every node that it generates,

$$\text{storage}(\text{A*}, R_d) = \Theta(\text{nodes}(R_d)) \approx \Theta(6.17^{2^d}). \tag{42}$$

Case 2. Suppose we do a state-space search, using a limited-memory iterative-deepening procedure a procedure like IDA* that only keeps track of the nodes on the path it is currently exploring. During each iteration of its loop, IDA* will generate each node u once for each path that it finds to u, so the number of generations will be at least as much as the number of complete paths in R_d. Thus,

$$\text{gen}(\text{IDA*}, R_d) = \Omega(\text{paths}(R_d)) = \Omega(2^{2^{d+2}-d/2}). \tag{43}$$

For $0 \le k \le \text{height}(R_d)$, IDA* will find paths to the nodes of depth height$(R_d) - k$ during at most $k+2$ of its loop iterations; and for each k, the total number of paths to nodes at this depth is no greater than paths(R_d). Therefore,

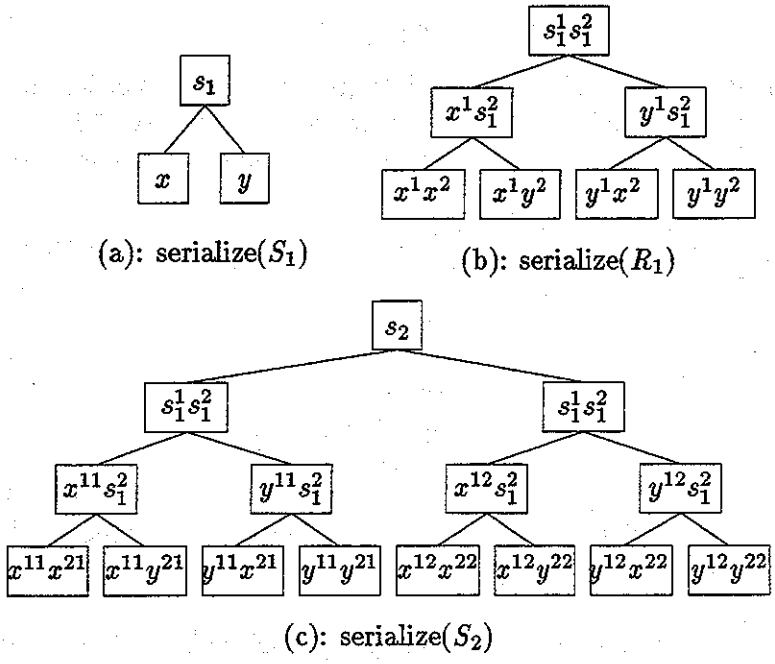
$$\text{gen}(\text{IDA*}, R_d) = O\left(\sum_{k=0}^{2^{d+1}-2} (k+2)\text{paths}(R_d)\right) = O(2^{2d}\text{paths}(R_d)) = O(2^{2^{d+2}+3d/2}). \tag{44}$$

Thus,

$$\text{gen}(\text{IDA*}, R) \approx \Theta(2^{2^{d+2}}) = \Theta(16^{2^d}). \tag{45}$$

Furthermore, since the length of every complete path of R is height$(R_d) = 2^{d+1} - 2$,

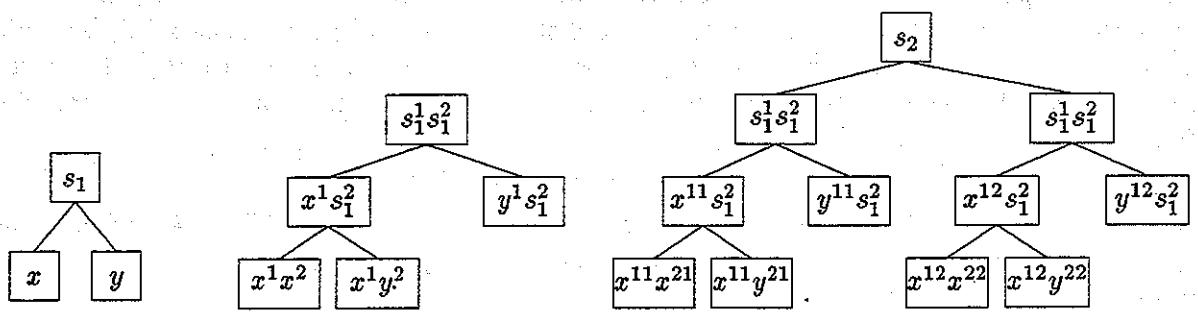
$$\text{storage}(\text{IDA*}, R_d) = \Theta(2^{d+1} - 2) = O(2^d). \tag{46}$$



(a): serialize(S_1) (b): serialize(R_1)

(c): serialize(S_2)

Figure 7: The state spaces serialize(S_1), serialize(R_1), and serialize(S_2), in the case where x and y are both goals. In this case, serialize(R_2) (not shown here) is a complete binary tree of height 6.



(a): serialize(S_1) (b): serialize(R_1) (c): serialize(S_2)

Figure 8: The state spaces serialize(S_1), serialize(R_1), and serialize(S_2), in the case where x is a goal but y is not. In this case, serialize(R_2) (not shown here) is a binary tree of height 6 whose structure is similar to what we would get by taking serialize(S_2) and attaching copies of itself to the four bottom-level leaves.

Case 3. Suppose we do a problem-reduction search, using a procedure like AO* that keeps track of every node that it generates. AO* will generate each node of $\text{ao}(R_d)$ once, so

$$\text{gen}(\text{AO}^*, R_d) = \text{nodes}(\text{ao}(R_d)) = \Theta(2^{2d}). \tag{47}$$

Since AO* stores every node that it generates,

$$\text{storage}(\text{AO}^*, R_d) = \text{nodes}(\text{ao}(R_d)) = \Theta(2^{2d}). \tag{48}$$

Case 4. Suppose we do a problem-reduction search, using a limited-memory iterative-deepening procedure a procedure like IDAO* that only keeps track of the nodes on the AND-tree that it is currently exploring. As discussed in Section 2.3, IDAO* will generate nodes in $\text{ao}(R_d)$ in the same order that IDA* would do on the state space $\text{serialize}(R_d)$. The worst case occurs when every leaf node of S_1 is a goal node. In this case, $\text{serialize}(R_d)$ is a complete binary tree of height $2^{d+1} - 2$, as illustrated in Fig. 7. Thus from Eqs. 3–4 and 7–8,

$$\begin{aligned} \text{gen}(\text{IDAO}^*, R_d) &= O(\text{gen}(\text{IDA}^*, \text{serialize}(R_d))) \\ &= O(2^{2^{d+1}-2}) = O(4^{2^d}); \end{aligned} \tag{49}$$

$$\begin{aligned} \text{storage}(\text{IDAO}^*, R) &= O(\text{storage}(\text{IDA}^*, \text{serialize}(R_d))) \\ &= O(2^{d+1} - 2) = O(2^d). \end{aligned} \tag{50}$$

The best case occurs when only one of S_1's leaf nodes is a goal node. In this case, it is straightforward to prove by induction that $\text{serialize}(R_d)$ is a tree of depth $2^{d+1} - 2$ with at least $\sqrt{2}^k$ nodes at depth k for each k (for example, see Fig. 8), for an effective branching factor of $\geq \sqrt{2}$. Thus from Eqs. 3–8,

$$\text{gen}(\text{IDAO}^*, R_d) = \Omega(\sqrt{2}^{2^{d+1}-2}) = \Omega(2^{2^d}); \tag{51}$$

$$\text{storage}(\text{IDAO}^*, R) = \Omega(2^{d+1} - 2) = \Omega(2^d). \tag{52}$$

5 Comparisons

Table 1 summarizes the results of Sections 3 and 4. The results in Section 3 assume that the branching factor is b, and the results in Section 4 assume that the branching factor is 2; so to make these results comparable, Table 1 uses a branching factor $b = 2$ throughout.

First, consider the case where only one problem-reduction step is possible. Here, the number of node generations is exponential for all four procedures (with AO* better than A* and IDAO*, and A* and IDAO* better than IDA*). But for IDAO* and IDA*, the number of nodes stored is only linear, whereas for A* and AO*, it is exponential (with AO* better than A*). Thus, AO* has the best time complexity, and IDAO* and IDA* have the best space complexity.

Second, consider the case where multiple problem-reduction steps are possible. For AO*, the number of node generations is singly exponential, but for the other three procedures it is doubly exponential (with IDAO* better than A* better than IDA*). For A*, the number of nodes stored is doubly exponential, and for the other three procedures it is only singly exponential (with IDAO* and IDA* better than AO*). Thus again, AO* has the best time complexity, and IDAO* and IDA* have the best space complexity.

Table 1: Time and space complexity of problem solving, depending on how many problem-reduction steps are available, whether the search procedure does problem reduction, and whether it remembers which nodes it has visited.

	One problem-reduction step is available		d problem-reduction steps are available	
Type of search procedure	node generations	nodes stored	node generations	nodes stored
limited-memory state-space (IDA*)	$\approx \Theta(16^d)$	$\Theta(d)$	$\approx \Theta(16^{2^d})$	$\Theta(2^d)$
full-memory state-space (A*)	$\Theta(4^d)$	$\Theta(4^d)$	$\approx \Theta(6.17^{2^d})$	$\approx \Theta(6.17^{2^d})$
limited-memory prob. reduction (IDAO*)	$\Omega(d2^d), O(4^d)$	$\Theta(d)$	$\Omega(2^{2^d}), O(4^{2^d})$	$\Theta(2^d)$
full-memory prob. reduction (AO*)	$\Theta(2^d)$	$\Theta(2^d)$	$\Theta(4^d)$	$\Theta(4^d)$

6 Conclusions

In this paper I have compared three approaches for trial-and-error problem solving: state-space search, problem reduction, and limited-memory iterative deepening. I have tightened Korf's results [3] (and thus also the results of Yang, Nau, and Hendler [12]) about what happens if one level of problem reduction is possible. However, the more interesting results are the ones telling what happens if multiple levels of problem reduction are possible, as occurs in an ordinary AND/OR tree—so these are the results I discuss below.

In the AND/OR-tree model, problem reduction reduces both the number of node generations and the number of stored nodes from approximately $\Theta(6.17^{2^d})$ down to $\Theta(4^d)$. This means that in problems for which problem reduction is possible, it can produce truly dramatic savings.

In contrast, the results for limited-memory iterative deepening are mixed. Without problem reduction, the number of stored nodes decreases from approximately $\Theta(6.17^{2^d})$ down to $\Theta(2^d)$; and with problem reduction, it decreases from $\Theta(4^d)$ to $\Theta(2^d)$. But in both cases, this reduction in memory comes at the expense of an increase in time: without problem reduction, the number of node generations increases from approximately $\Theta(6.17^{2^d})$ to approximately $\Theta(16^{2^d})$; and with problem reduction, it increases from $\Theta(4^d)$ to $\Theta(4^{2^d})$.[2] Thus, whether we should use a procedure that remembers every node it has visited, or one that just remembers the nodes on the current path or solution tree, depends on which is more important: saving memory or saving time.

Note, however, that these results say nothing about the use of iterative deepening in game-tree searching. On game trees, the nature and purpose of iterative deepening is very different: the idea is not to try to save memory, but instead to find the largest value of k such that we can search to depth k in the amount of time available (for a more detailed description, see for example [11]). On a binary game tree, in the worst case an alpha-beta search to depth d takes $\Theta(d)$ space and $\Theta(2^d)$ time; and so does an iterative-deepening alpha-beta search.

[2] Furthermore, if we had used an AND/OR-graph model rather than an AND/OR-tree model, there would be even more paths to each node, so limited-memory iterative deepening might do even more node generations.

Acknowledgement

I would like to acknowledge the students in my graduate-level Artificial Intelligence course (particularly Tamara Gibson), whose questions about one of my homework assignments prompted me to do the work reported in this paper.

References

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw Hill, 1990.

[2] Richard Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[3] Richard Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1):65–88, September 1987.

[4] V. Kumar, D. S. Nau, and L. Kanal. A general branch-and-bound formulation for and/or graph and game tree search. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 91–130. Springer-Verlag, New York, 1988.

[5] A. Mahanti, D. S. Nau, S. Ghosh, A. K. Pal, and L. N. Kanal. Performance of limited-memory heuristic search algorithms for networks: A theoretical and empirical analysis. 1991. Submitted for publication.

[6] A. Mahanti, D. S. Nau, S. Ghosh, A. K. Pal, and L. N. Kanal. Performance of IDA* on trees and graphs. In *Proc. AAAI-92*, pages 539–544, July 1992.

[7] A. Martelli and U. Montanari. Additive and/or graphs. In *Proc. Third Internat. Joint Conf. on Artif. Intell.*, pages 1–11, 1973.

[8] D. S. Nau, V. Kumar, and L. N. Kanal. General branch and bound, and its relation to A* and AO*. *Artificial Intelligence*, 23:29–58, 1984.

[9] N. Nilsson. Searching problem solving and game playing trees for minimum cost solutions. In A. J. H. Morrel, editor, *Information Processing-68*, volume 2, pages 1556–1562. 1968.

[10] Nils Nilsson. *Principles of Artificial Intelligence.* Morgan Kaufmann, 1980.

[11] T. R. Truscott. Techniques used in minimax game-playing programs. Master's thesis, Duke University, Durham, NC, 1981.

[12] Q. Yang, D. S. Nau, and J. Hendler. Merging separately generated plans with restricted interactions. *Computational Intelligence*, 8(2):648–676, February 1992.