An Algorithm better than AO*?

Blai Bonet
Departamento de Computación
Universidad Simón Bolívar
Caracas, Venezuela
bonet@ldc.usb.ve

Héctor Geffner
ICREA & Universitat Pompeu Fabra
Paseo de Circunvalación, 8
Barcelona, Spain
hector.geffner@upf.edu

Abstract

Recently there has been a renewed interest in AO* as planning problems involving uncertainty and feedback can be naturally formulated as AND/OR graphs. In this work, we carry out what is probably the first detailed empirical evaluation of AO* in relation to other AND/OR search algorithms. We compare AO* with two other methods: the well-known Value Iteration (VI) algorithm, and a new algorithm, Learning in Depth-First Search (LDFS). We consider instances from four domains, use three different heuristic functions, and focus on the optimization of cost in the worst case (Max AND/OR graphs). Roughly we find that while AO* does better than VI in the presence of informed heuristics, VI does better than recent extensions of AO* in the presence of cycles in the AND/OR graph. At the same time, LDFS and its variant Bounded LDFS, which can be regarded as extensions of IDA*, are almost never slower than either AO* or VI, and in many cases, are orders-of-magnitude faster.

Introduction

A* and AO* are the two classical heuristic best-first algorithms for searching OR and AND/OR graphs (Hart, Nilsson, & Raphael 1968; Martelli & Montanari 1973; Pearl 1983). The A* algorithm is taught in every AI class, and has been studied throughly both theoretically and empirically. The AO* algorithm, on the other hand, has found less uses in AI, and while prominent in early AI texts (Nilsson 1980) it has disappeared from current ones (Russell & Norvig 1994). In the last few years, however, there has been a renewed interest in the AO* algorithm in planning research where problems involving uncertainty and feedback can be formulated as search problems over AND/OR graphs (Bonet & Geffner 2000).

In this work, we carry out what is probably the first in-depth empirical evaluation of AO* in relation with other AND/OR graph search algorithms. We compare AO* with an old but general algorithm, *Value Iteration* (Bellman 1957; Bertsekas 1995), and a new algorithm, *Learning in Depth-First Search*, and its variant Bounded LDFS (Bonet & Geffner 2005). While VI performs a sequence of Bellman updates over all states in parallel until convergence, LDFS performs selective Bellman updates on top of successive depth-first searches, very much as Learning RTA* (Korf 1990) and RTDP (Barto, Bradtke, & Singh 1995) perform Bellman updates on top of successive greedy (real-time) searches.

Copyright © 2005, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

In the absence of accepted benchmarks for evaluating AND/OR graph search algorithms, we introduce four parametric domains, and consider a large number of instances, some involving millions of states. In all cases we focus on the computation of solutions with minimum cost in the worst case using three different and general admissible heuristic functions. We find roughly that while AO* does better than VI in the presence of informed heuristics, LDFS, with or without heuristics, tends to do better than both.

AO* is limited to handling AND/OR graphs without cycles. The difficulties arising from cycles can be illustrated by means of a simple graph with two states and two actions: an action a with cost 5 maps the initial state s_0 non-deterministically into either a goal state s_G or s_0 itself, and a second action b with cost 10 maps s_0 deterministically into s_G. Clearly, the problem has cost 10 and b is the only (optimal) solution, yet the simple cost revision step in AO* does not yield this result. Thus, for domains where such cycles appear, we evaluate a recent variant of AO*, CFC_{rev^*}, introduced in (Jimenez & Torras 2000) that is not affected by this problem. We could have used LAO* as well (Hansen & Zilberstein 2001), but this would be an overkill as LAO* is designed to minimize expected cost in probabilistic AND/OR graphs (MDPs) where *solutions themselves* can be cyclic, something that cannot occur in Additive or Max AND/OR graphs. Further algorithms for cyclic graphs are discussed in (Mahanti, Ghose, & Sadhukhan 2003). LDFS has no limitations of this type; unlike AO*, it is not affected by the presence of cycles in the graph, and unlike Value Iteration, it is not affected either by the presence of dead-ends in the state space if the problem is solvable.

The paper is organized as follows: we consider first the models, then the algorithms, the experimental set up and the results, and close with a brief discussion.

Models

We consider AND/OR graphs that arise from non-deterministic state models as those used in planning with non-determinism and full observability, where there are

1. a discrete and finite state space S,

2. an initial state $s_0 \in S$,
3. a non-empty set of terminal states $S_T \subseteq S$,
4. actions $A(s) \subseteq A$ applicable in each non-terminal state,
5. a function mapping non-terminal states s and actions $a \in A(s)$ into *sets* of states $F(a,s) \subseteq S$,
6. action costs $c(a,s)$ for non-terminal states s, and
7. terminal costs $c_T(s)$ for terminal states.

Models where the states are only *partially observable,* can be described in similar terms, replacing states by *sets of states* or *belief states* (Bonet & Geffner 2000).

We assume that both $A(s)$ and $F(a,s)$ are non-empty, that action costs $c(a,s)$ are all positive, and terminal costs $c_T(s)$ are non-negative. When terminal costs are all zero, terminal states are called *goals*.

The mapping from non-deterministic state models to AND/OR graphs is immediate: non-terminal states s become OR nodes, connected to the AND nodes $< s, a >$ for each $a \in A(s)$, whose children are the states $s' \in F(a,s)$. The inverse mapping is also direct.

The solutions to this and various other state models can be expressed in terms of the so-called Bellman equation that characterizes the *optimal cost function* (Bellman 1957; Bertsekas 1995):

$$V(s) \stackrel{\text{def}}{=} \begin{cases} c_T(s) & \text{if } s \text{ terminal} \\ \min_{a \in A(s)} Q_V(a,s) & \text{otherwise} \end{cases} \quad (1)$$

where $Q_V(a,s)$ is an abbreviation of the cost-to-go, which for Max and Additive AND/OR graphs takes the form:

$$Q_V(a,s) \ : \ \begin{cases} c(a,s) + \max_{s' \in F(a,s)} V(s') & \text{(Max)} \\ c(a,s) + \sum_{s' \in F(a,s)} V(s') & \text{(Add)} \end{cases} \quad (2)$$

Other models can be handled in this way by choosing other forms for $Q_V(a,s)$. For example, for MDPs, it is the weighted sum $c(a,s) + \sum_{s' \in F(a,s)} V(s') P_a(s'|s)$ where $P_a(s'|s)$ is the probability of going from s to s' given a.

In the absence of dead-ends, there is a unique (optimal) value function $V^*(s)$ that solves the Bellman equation, and the optimal solutions can be expressed in terms of the policies π that are *greedy* with respect to $V^*(s)$. A policy π is a function mapping states $s \in S$ into actions $a \in A(s)$, and a policy π_V is greedy with respect to a value function $V(s)$, or simply greedy in V, iff π_V is the best policy assuming that the cost-to-go is given by $V(s)$; i.e.

$$\pi_V(s) \ = \ \operatorname*{argmin}_{a \in A(s)} Q_V(a,s)\,. \quad (3)$$

Since the initial state s_0 is known, it is actually sufficient to consider *closed (partial) policies* π that prescribe the actions to do in all (non-terminal) states reachable from s_0 and π. Any closed policy π relative to a state s has a cost $V^\pi(s)$ that expresses the cost of solving the problem starting from s. The costs $V^\pi(s)$ are given by the solution of (1) but with the operator $\min_{a \in A(s)}$ removed and the action a replaced by $\pi(s)$. These costs are well-defined when the resulting equations have a solution over the subset of states reachable from s_0 and π. For Max and Additive AND/OR graphs, this happens when π is *acyclic*; else $V^\pi(s_0) = \infty$. When π is acyclic, the costs $V^\pi(s_0)$ can be defined recursively starting with the terminal states s' for which $V^\pi(s') = c_T(s')$, and up to the non-terminal states s reachable from s_0 and π for which $V^\pi(s) = Q_{V^\pi}(\pi(s), s)$. In all cases, we are interested in computing a solution π that minimizes $V^\pi(s_0)$. The resulting value is the optimal cost of the problem $V^*(s_0)$.

Algorithms

We consider three algorithms for computing such optimal solutions for AND/OR graphs: Value Iteration, AO*, and Learning in Depth First Search.

Value Iteration

Value iteration is a simple and quite effective algorithm that computes the fixed point $V^*(s)$ of Bellman equation by plugging an estimate value function $V_i(s)$ in the right-hand side and obtaining a new estimate $V_{i+1}(s)$ on the left-hand side, iterating until $V_i(s) = V_{i+1}(s)$ for all $s \in S$ (Bellman 1957). In our setting, this convergence is guaranteed provided that there are no dead-end states, i.e., states s for which $V^*(s) = \infty$. Often convergence is accelerated if the same value function vector $V(s)$ is used on both left and right. In such a case, in each iteration, the states values are *updated* sequentially from first to last as:

$$V(s) \ := \ \min_{a \in A(s)} Q_V(a,s)\,. \quad (4)$$

The iterations continue until V satisfies the Bellman equation, and hence $V = V^*$. Any policy π greedy in V^* provides then an optimal solution to the problem. VI can deal with a variety of models and is very easy to implement.

AO*

AO* is a best-first algorithm for solving acyclic AND/OR graphs (Martelli & Montanari 1973; Nilsson 1980; Pearl 1983). Starting with a partial graph G containing only the initial state s_0, two operations are performed iteratively: first, a best partial policy over G is constructed and a non-terminal tip state s reachable with this policy is expanded; second, the value function and best policy over the updated graph are incrementally recomputed. This process continues until the best partial policy is complete. The second step, called the *cost revision step*, exploits the acyclicity of the AND/OR graph for recomputing the optimal costs and policy over the partial graph G *in a single pass,* unlike Value Iteration (yet see (Hansen & Zilberstein 2001)). In this computation, the states outside G are regarded as terminal states with costs given by their heuristic values. When the AND/OR graph contains cycles, however, this basic cost-revision operation is not adequate. In this paper, we use the AO* variant developed in (Jimenez & Torras 2000), called CFC_{rev^*}, which is based in the cost revision operation from (Chakrabarti 1994) and is able to handle cycles.

Unlike VI, AO* can solve AND/OR graphs without having to consider the entire state space, and exploits lower bounds for focusing the search. Still, expanding the partial graph one state at a time, and recomputing the best policy over the graph after each step, imposes an overhead that, as we will see, does not always appear to pay off.

Learning DFS

LDFS is an algorithm akin to IDA* with transposition tables which applies to a variety of models (Bonet & Geffner 2005). While IDA* consists of a sequence of DFS iterations that backtrack upon encountering states with costs exceeding a given bound, LDFS consists of a sequence of DFS iterations that backtrack upon encountering states that are *inconsistent*: namely states s whose values are not consistent with the values of its children; i.e. $V(s) \neq \min_{a \in A(s)} Q_V(a, s)$. The expression $Q_V(a, s)$ encodes the type of model: OR graphs, Additive or Max AND/OR graphs, MDPs, etc. Upon encountering such inconsistent states, LDFS updates their values (making them consistent) and backtracks, updating along the way ancestor states as well. In addition, when the DFS beneath a state s does not find an inconsistent state (a condition kept by $flag$ in Fig. 1), s is labeled as *solved* and is not expanded again. The DFS iterations terminate when the initial state s_0 is solved. Provided the initial value function is admissible and monotonic (i.e., $V(s) \leq \min_{a \in A(s)} Q_V(a, s)$ for all s), LDFS returns an optimal policy if one exists. The code for LDFS is quite simple and similar to IDA* (Reinefeld & Marsland 1994); see Fig. 1.

Bounded LDFS, shown in Fig. 2, is a slight variation of LDFS that accommodates an explicit $bound$ parameter for focusing the search further on paths that are 'critical' in the presence of Max rather than Additive models. For Game Trees, Bounded LDFS reduces to the state-of-the-art MTD$(-\infty)$ algorithm: an iterative alpha-beta search procedure with null windows and memory (Plaat *et al.* 1996). The code in Fig. 2, unlike the code in (Bonet & Geffner 2005) is for general Max AND/OR graphs and not only trees, and replaces the boolean SOLVED(s) tag in LDFS by a numerical tag $U(s)$ that stands for an *upper bound*; i.e., $U(s) \geq V^*(s) \geq V(s)$. This change is needed because Bounded LDFS, unlike LDFS, minimizes $V^\pi(s_0)$ but not necessarily $V^\pi(s)$ for all states s reachable from s_0 and π (in Additive models, the first condition implies the second). Thus, while the SOLVED(s) tag in LDFS means that an optimal policy for s has been found, the $U(s)$ tag in Bounded LDFS means only that a policy π with cost $V^\pi(s) = U(s)$ has been found. Bounded LDFS ends however when the lower and upper bounds for s_0 coincide. The upper bounds $U(s)$ are initialized to ∞. The code in Fig. 2 is for Max AND/OR graphs; for Additive graphs, the term $\sum_{s''} V(s'')$ needs to be substracted from the right-hand side of line $nb := bound - c(a, s)$ for s'' in $F(a, s)$ and $s'' \neq s'$. The resulting procedure however is equivalent to LDFS.

```
LDFS-DRIVER(s0)
begin
    repeat solved := LDFS(s0) until solved
    return (V, π)
end

LDFS(s)
begin
    if s is SOLVED or terminal then
        if s is terminal then V(s) := c_T(s)
        Mark s as SOLVED
        return true

    flag := false
    foreach a ∈ A(s) do
        if Q_V(a, s) > V(s) then continue
        flag := true
        foreach s' ∈ F(a, s) do
            flag := LDFS(s') & [Q_V(a, s) ≤ V(s)]
            if ¬flag then break
        if flag then break

    if flag then
        π(s) := a
        Mark s as SOLVED
    else
        V(s) := min_{a∈A(s)} Q_V(a, s)
    return flag
end
```

Algorithm 1: Learning DFS

Experiments

We implemented all algorithms in C++. Our AO* code is a careful implementation of the algorithm in (Nilsson 1980), while our CFC$_{rev^*}$ code is a modification of the code obtained from the authors (Jimenez & Torras 2000) that makes it roughly an order-of-magnitude faster.

For all algorithms we initialize the values of the terminal states to their true values $V(s) = c_T(s)$ and non-terminals to some *heuristic values* $h(s)$ where h is an admissible and monotone heuristic function. We consider three such heuristics: the first, the non-informative $h = 0$, and then two functions h_1 and h_2 that stand for the value functions that result from performing n iterations of value iteration, and an equivalent number of 'random' state updates respectively,[1] starting with $V(s) = 0$ at non-terminals. In all the experiments, we set n to $N_{vi}/2$ where N_{vi} is the number of iterations that value iteration takes to converge. These heuristics are informative but expensive to compute, yet we use them for assessing how well the various algorithms are able to exploit heuristic information. The times for computing the heuristics are common to all algorithms and are not included in the runtimes.

We are interested in *minimizing cost in the worst case* (Max AND/OR graphs). Some relevant features of the instances considered are summarized in Table 1. A brief description of the domains follows.

Coins: There are N coins including a counterfeit coin that is either lighter or heavier than the others, and a 2-pan balance. A strategy is needed for identifying the counterfeit coin, and whether it is heavier or lighter than the others (Pearl 1983). We experiment with $N = 10, 20, \ldots, 60$. In order to reduce symmetries we use the representation from (Fuxi, Ming, & Yanxiang 2003) where a (belief) state is a tuple of non-negative integers (s, ls, hs, u) that add up to N

[1] More precisely, the random updates are done by looping over the states $s \in S$, selecting and updating states s with probability $1/2$ til $n \times |S|$ updates are made.

```
B-LDFS-DRIVER(s_0)
begin
    repeat B-LDFS(s_0, V(s_0)) until V(s_0) ≥ U(s_0)
    return (V, π)
end

B-LDFS(s, bound)
begin
    if s is terminal or V(s) ≥ bound then
        if s is terminal then V(s) := U(s) := c_T(s)
        return

    flag := false
    foreach a ∈ A(s) do
        if Q_V(a, s) > bound then continue
        flag := true
        foreach s' ∈ F(a, s) do
            nb := bound − c(a, s)
            flag := B-LDFS(s', nb) & [Q_V(a, s) ≤ bound]
            if ¬flag then break
        if flag then break

    if flag then
        π(s) := a
        U(s) := bound
    else
        V(s) := min_{a∈A(s)} Q_V(a, s)
end
```

Algorithm 2: Bounded LDFS for Max AND/OR Graphs

and stand for the number of coins that are known to be of standard weight (s), standard or lighter weight (ls), standard or heavier weight (hs), and completely unknown weight (u). See (Fuxi, Ming, & Yanxiang 2003) for details.

Diagnosis: There are N binary tests for finding out the true state of a system among M different states (Pattipati & Alexandridis 1990). An instance is described by a binary matrix T of size $M \times N$ such that $T_{ij} = 1$ iff test j is positive when the state is i. The goal is to obtain a strategy for identifying the true state. The search space consists of all non-empty subsets of states or 'belief states', and the actions are the tests. Solvable instances can be generated by requiring that no two rows in T are equal, and $N > \log_2(M)$ (Garey 1972). We performed two classes of experiments: a first class with N fixed to 10 and M varying in $\{10, 20, \ldots, 60\}$, and a second class with M fixed to 60 and N varying in $\{10, 12, \ldots, 28\}$. In each case, we report average runtimes and standard deviations over 5 random instances.

Rules: We consider the derivation of atoms in acyclic rule systems with N atoms, and at most R rules per atom, and M atoms per rule body. In the experiments $R = M = 50$ and N is in $\{5000, 10000, \ldots, 20000\}$. For each value of N, we report average times and standard deviations over 5 random solvable instances.

Moving Target Search: A predator must catch a prey that moves non-deterministically to a non-blocked adjacent cell in a given random maze of size $N \times N$. At each time, the predator and prey move one position. Initially, the predator is in the upper left position and the prey in the bottom right position. The task is to obtain an optimal strategy for catching the prey. In (Ishida & Korf 1991), a similar problem is considered in a real-time setting where the predator moves 'faster' than the prey, and no optimality requirements are made. Solvable instances are generated by ensuring that the undirected graph underlying the maze is connected and loop free. Such loop-free mazes can be generated by performing random Depth-First traversals of the $N \times N$ empty grid, inserting 'walls' when loops are encountered. We consider $N = 15, 20, \ldots, 40$, and in each case report average times and standard deviations over 5 random instances. Since the resulting AND/OR graphs involve cycles, the algorithm CFC_{rev^*} is used instead of AO*.

problem	$\|S\|$	V^*	N_{VI}	$\|A\|$	$\|F\|$	$\|\pi^*\|$
coins-10	43	3	2	172	3	9
coins-60	1,018	5	2	315K	3	12
mts-5	625	17	14	4	4	156
mts-35	$1,5M$	573	322	4	4	220K
mts-40	$2,5M$	684	–	4	4	304K
diag-60-10	29,738	6	8	10	2	119
diag-60-28	$> 15M$	6	–	28	2	119
rules-5000	5,000	156	158	50	50	4,917
rules-20000	20,000	592	594	50	50	19,889

Table 1: Data for smallest and largest instances: number of (reachable) states or belief states, optimal cost, number of iterations taken by VI, max branching in OR and AND nodes, and size of optimal solution ($M = 10^6$; $K = 10^3$)

Results

The results of the experiments are shown in Fig. 2, along with a detailed explanation of the data. Each square depicts the runtimes in seconds for a given domain and heuristic in a logarithmic scale. The domains from top to bottom are COINS, DIAGNOSIS 1 and 2, RULES, and MTS, while the heuristics from left to right are $h = 0$, h_1, and h_2. As mentioned above, MTS involves cycles, and thus, CFC_{rev^*} is used instead of AO*. Thus leaving this domain aside for a moment, we can see that with the two (informed) heuristics, AO* does better than VI in almost all cases, with the exception of COINS where VI beats both AO* and LDFS by a small margin. Indeed, as it can be seen in Table 1, VI happens to solve COINS in very few iterations (this actually has to do with a topological sort done in our implementation of VI for finding first the states that are reachable). In DIAGNOSIS, and indeed in COINS with h_1, AO* runs one or more orders of magnitude faster than VI. With $h = 0$, the results are mixed, with VI doing better, and in certain cases (DIAGNOSIS) much better. Adding now LDFS to the picture, we see that it is never worse than either AO* or VI, except in COINS with $h = 0$ and h_2, and RULES with $h = 0$ where it is slower than VI and AO* respectively by a small factor (in the latter case 2). In most cases, however, LDFS runs faster than both AO* and VI for the different heuristics, in several of them by one or more orders of magnitude. Bounded LDFS

in turn does never worse than LDFS, and in a few cases, including DIAGNOSIS with $h = 0$, runs an order of magnitude faster. Finally, in MTS, a problem which involves cycles in the AND/OR graph, AO* cannot be used, CFC_{rev^*} solves only the smallest problem, and VI solves all but the largest problem, an order of magnitude slower than LDFS, which in turn is slower than Bounded LDFS.

The difference in performance between VI and the other algorithms for $h \neq 0$ suggests that the latter make better use of the initial heuristic values. At the same time, the difference between LDFS and AO* suggests that often the overhead involved in expanding the partial graph one state at a time, and recomputing the best policy over the graph after each step, does not always pay off. LDFS makes use of the heuristic information but makes no such (best-first) commitments. Last, the difference in performance between LDFS and Bounded LDFS can be traced to a theoretical property mentioned above and discussed in further detail in (Bonet & Geffner 2005): while LDFS (and AO* and VI) compute policies π that are optimal over all the states reachable from s_0 and π (globally optimal); Bounded LDFS computes policies π that are optimal only where needed; i.e. in s_0. For OR and Additive AND/OR graphs, the latter notion implies the former, but for Max models does not. Bounded LDFS (and Game Tree algorithms) exploits this distinction, while LDFS, AO*, and Value Iteration do not.

Discussion

We have carried an empirical evaluation of AND/OR search algorithms over a wide variety of instances, using three heuristics, and focusing in the optimization of cost in the worst case (Max AND/OR graphs). Over these examples and with these heuristics, the studied algorithms rank from fastest to slowest as Bounded LDFS, LDFS, AO*, and VI, with some small variations. We have considered the solution of Max AND/OR graphs as it relates well to problems in planning where one aims to minimize cost in the worst case. Additive AND/OR graphs, on the other hand, do not provide a meaningful cost criteria for the problems considered, as in the presence of common subproblems they count repeated solution subgraphs multiple times. The semantics of Max AND/OR graphs does not have this problem. Still we have done preliminary tests under the Additive semantics to find out whether the results change substantially or not. Interestingly, in some domains like diagnosis, the results do not change much, but in others, like RULES they do,[2] making indeed AO* way better than LDFS and VI, and suggesting, perhaps not surprisingly, that the effective solution of Additive and Max AND/OR graphs may require different ideas in each case. In any case, by making the various problems and source codes available, we hope to encourage the necessary experimentation that has been lacking so far in the area.

References

Barto, A.; Bradtke, S.; and Singh, S. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence* 72:81–138.

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press.

Bertsekas, D. 1995. *Dynamic Programming and Optimal Control, Vols 1 and 2.* Athena Scientific.

Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In *Proc. of AIPS-2000*, 52–61. AAAI Press.

Bonet, B., and Geffner, H. 2005. Learning in DFS: A unified approach to heuristic search in deterministic, non-deterministic, probabilistic, and game tree settings.

Chakrabarti, P. P. 1994. Algorithms for searching explicit AND/OR graphs and their applications to problem reduction search. *Artificial Intelligence* 65(2):329–345.

Fuxi, Z.; Ming, T.; and Yanxiang, H. 2003. A solution to billiard balls puzzle using AO* algorithm. In Palade, V.; Howlett, R.; and Jain, L., eds., *Proc. 7th Int. Conf. on Knowledge-Based Int. Information & Engineering Systems*, 1015–1022. Springer.

Garey, M. 1972. Optimal binary identification procedures. *SIAM Journal on Applied Mathematics* 23(2):173–186.

Hansen, E., and Zilberstein, S. 2001. Lao*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence* 129:35–62.

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.* 4:100–107.

Ishida, T., and Korf, R. 1991. Moving target search. In *Proc. IJCAI-91*, 204–211.

Jimenez, P., and Torras, C. 2000. An efficient algorithm for searching implicit AND/OR graphs with cycles. *Artificial Intelligence* 124(1):1–30.

Korf, R. 1990. Real-time heuristic search. *Artificial Intelligence* 42:189–211.

Mahanti, A.; Ghose, S.; and Sadhukhan, S. K. 2003. A framework for searching and/or graphs with cycles. *CoRR cs.AI/0305001*. At http://arxiv.org/abs/cs.AI/0305001.

Martelli, A., and Montanari, U. 1973. Additive AND/OR graphs. In *Proc. IJCAI-73*, 1–11.

Nilsson, N. 1980. *Principles of Artificial Intelligence*. Tioga.

Pattipati, K., and Alexandridis, M. 1990. Applications of heuristic search and information theory to sequential fault diagnosis. *IEEE Trans. System, Man and Cybernetics* 20:872–887.

Pearl, J. 1983. *Heuristics*. Addison Wesley.

Plaat, A.; Schaeffer, J.; Pijls, W.; and Bruin, A. 1996. Best-first fixed-depth minimax algorithms. *Artificial Intelligence* 87(1-2):255–293.

Reinefeld, A., and Marsland, T. 1994. Enhanced iterative-deepening search. *IEEE Trans. on Pattern Analysis and Machine Intelligence* 16(7):701–710.

Russell, S., and Norvig, P. 1994. *Artificial Intelligence: A Modern Approach*. Prentice Hall.

[2] Note that due to the common subproblems, the algorithms would *not* minimize the number of rules in the derivations.

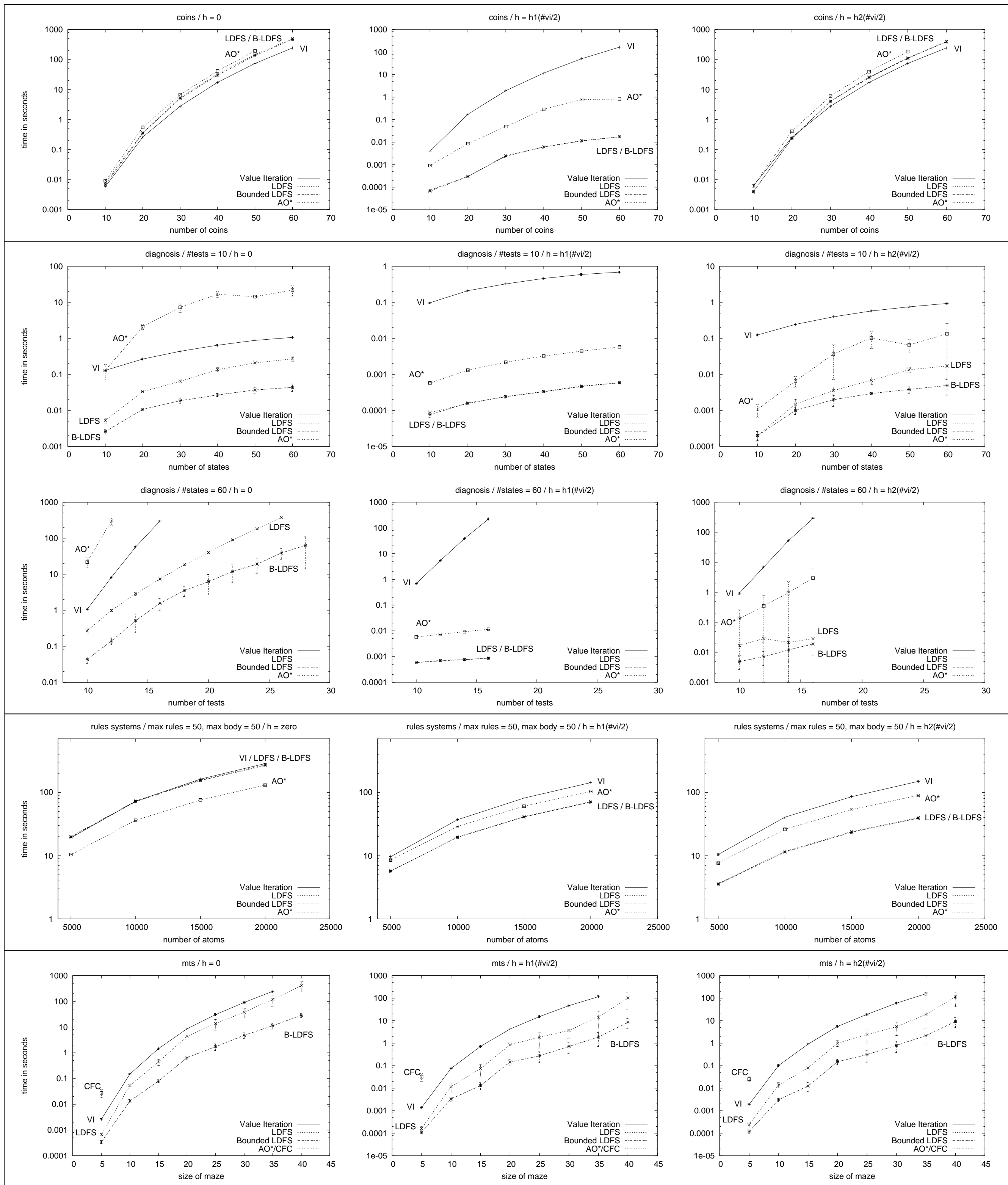

Table 2: Experiments: each square depicts runtimes in seconds for problems with a given domain and heuristic. The domains are from top to bottom: COINS, DIAGNOSIS 1 and 2, RULES, and MTS, and the heuristics from left to right: $h = 0$, h_1, and h_2. In the first diagnosis domain, the number of states is increased, while in the second, the number of tests. Problems with more than 16 tests are not solved for h_1 and h_2 as these heuristics could not be computed beyond that point. Such problems are solved by LDFS and Bounded LDFS with $h = 0$. All runtimes are shown in logarithmic scales, yet the range of scales vary.