Part I: Heuristics

1

Heuristic Search for Planning under Uncertainty

Blai Bonet and Eric A. Hansen

1 Introduction

The artificial intelligence (AI) subfields of heuristic search and automated planning are closely related, with planning problems often providing a stimulus for developing and testing search algorithms. Classical approaches to heuristic search and planning assume a deterministic model of sequential decision making in which a solution takes the form of a sequence of actions that transforms a start state into a goal state. The effectiveness of heuristic search for classical planning is illustrated by the results of the planning competitions organized by the AI planning community, where optimal planners based on A*, and satisficing planners based on variations of best-first search and enforced hill climbing, have performed as well or better than many other planners in the deterministic track of the competition [Edelkamp, Hoffmann, and Littman 2004; Gerevini, Bonet, and Givan 2006].

Beginning in the 1990's, AI researchers became increasingly interested in the problem of planning under uncertainty and adopted Markov decision theory as a framework for formulating and solving such problems [Boutilier, Dean, and Hanks 1999]. The traditional dynamic programming approach to solving Markov decision problems (MDPs) [Bertsekas 1995; Puterman 1994] can be viewed as a form of "blind" or uninformed search. Accordingly, several AI researchers considered how to generalize well-known heuristic-search techniques in order to develop more efficient planning algorithms for MDPs. The advantage of heuristic search over traditional, blind dynamic programming is that it uses an admissible heuristic and intelligent search control to focus computation on solving the problem for relevant states, given a start state and goal states, without considering irrelevant or unreachable parts of the state space.

In this article, we present an overview of research on heuristic search for problems of sequential decision making where state transitions are stochastic instead of deterministic, an important class of planning problems that corresponds to the most basic kind of Markov decision process, called a fully-observable Markov decision process. For this special case of the problem of planning under uncertainty, a fairly mature theory of heuristic search has emerged over the past decade and a half. In reviewing this work, we focus on two key issues: how to generalize classic heuristic search algorithms in order to solve planning problems with stochastic state

transitions, and how to compute admissible heuristics for these search problems.

Judea Pearl's classic book, *Heuristics*, provides a comprehensive overview of heuristic search theory as of its publication date in 1984. One of our goals in this article is to show that the twin themes of that book, admissible heuristics and intelligent search control, have been central issues in the subsequent development of a class of algorithms for problems of planning under uncertainty. In this short survey, we rely on references to the literature for many of the details of the algorithms we review, including proofs of their properties and experimental results. Our objective is to provide a high-level overview that identifies the key ideas and contributions in the field and to show how the new search algorithms for MDPs relate to the classical search algorithms covered in Pearl's book.

2 Planning with uncertain state transitions

Many planning problems can be modeled by a set of states, S, that includes an initial state $s_{init} \in S$ and a set of goal states, $G \subseteq S$, and a finite set of applicable actions, $A(s) \subseteq A$, for each non-goal state $s \in S \backslash G$, where each action incurs a positive cost $c(s, a)$. In a classical, deterministic planning problem, an action $a \in A(s)$ causes a *deterministic* transition, where $f(s, a)$ is the next state after applying action a in state s. The objective of a planner is to find a sequence of actions, $\langle a_0, a_1, \ldots, a_n \rangle$, that when applied to the initial state results in a trajectory, $\langle s_0 = s_{init}, a_0, s_1, a_1, \ldots, a_n, s_{n+1} \rangle$, that ends in a goal state, $s_{n+1} \in G$, where $a_i \in A(s_i)$ and $s_{i+1} = f(s_i, a_i)$. Such a plan is optimal if its cost, $\sum_{i=0}^{n} c(s_i, a_i)$, is minimum among all possible plans that achieve a goal.

To model the uncertain effects of actions, we consider a generalization of this model in which the deterministic transition function is replaced by a *stochastic* transition function, $p(\cdot|s, a)$, where $p(s'|s, a)$ is the probability of making a transition to state s' after taking action a in state s. In general, the cost of an action depends on the successor state; but usually, it is sufficient to consider the expected cost of an action, denoted $c(s, a)$.

With this simple change of the transition function, the planning problem is changed from a deterministic shortest-path problem to a *stochastic shortest-path problem*. As defined by Bertsekas and Tsitsiklis [Bertsekas and Tsitsiklis 1991], a stochastic shortest-path problem can have actions that incur positive or negative costs. But several subsequent researchers, including Barto et al. [Barto, Bradtke, and Singh 1995], assume that a stochastic shortest-path problem only has actions that incur positive costs. The latter assumption is in keeping with the model of planning problems we sketched above, as well as classical models of heuristic search, and so we ordinarily assume that the actions of a stochastic shortest-path problem incur positive costs only. In case where we allow actions to have both positive and negative costs, we make this clear.

Defined in either way, a stochastic shortest-path problem is a special case of a fully-observable infinite-horizon Markov decision process (MDP). There are several

MDP models with different optimization criteria, and almost all of the algorithms and results we review in this article apply to other MDPs. The most widely-used model in the AI community is the discounted infinite-horizon MDP. In this model, there are rewards instead of costs, $r(s, a)$ denotes the expected reward for taking action a in state s, which can be positive or negative, $\gamma \in (0, 1)$ denotes a discount factor, and the objective is to maximize expected total discounted reward over an infinite horizon. Interestingly, any discounted infinite-horizon MDP can be reduced to an equivalent stochastic shortest-path problem [Bertsekas 1995; Bonet and Geffner 2009]. Thus, we do not sacrifice any generality by focusing our attention on stochastic shortest-path problems.

Adoption of a stochastic transition model has important consequences for the structure of a plan. A plan no longer takes the simple form of a sequence of actions. Instead, it is typically represented by a mapping from states to actions, $\pi : S \rightarrow A$, called a *policy* in the literature on MDPs. (For the class of problems we consider, where the horizon is infinite, a planner only needs to consider *stationary* policies, which are policies that are not indexed by time.) Note that this representation of a plan assumes closed-loop plan execution instead of open-loop plan execution. It also assumes that an agent always knows the current state of the system; this is what is meant by saying the MDP is *fully observable*.

A stochastic shortest-path problem is solved by finding a policy that reaches a goal state with probability one after a finite number of steps, beginning from any other state. Such a policy is called a *proper policy*. Given a stochastic transition model, it is not possible to bound the number of steps of plan execution it takes to achieve a goal, even for proper policies. Thus, a stochastic shortest-path problem is an *infinite-horizon* MDP. In the infinite-horizon framework, the termination of a plan upon reaching a goal state is modeled by specifying that goal states are zero-cost absorbing states, which means that for all $s \in G$ and $a \in A$, $c(s, a) = 0$ and $p(s|a, s) = 1$. Equivalently, we can assume that no actions are applicable in a goal state. To reflect the fact that plan execution terminates after a finite, but uncertain and unbounded, number of steps, this kind of infinite-horizon MDP is also called an *indefinite-horizon* MDP. Note that when the state set is finite and the number of steps of plan execution is unbounded, the same state can be visited more than once during execution of a policy. Thus, a policy specifies not only conditional behavior, but cyclic behavior too.

For a process that is controlled by a fixed policy π, stochastic trajectories beginning from state s_0, of the form $\langle s_0, \pi_0(s_0), s_1, \pi_1(s_1), \ldots\rangle$, are generated with probability $\prod_{i=0}^{\infty} p(s_{i+1}|s_i, \pi(s_i))$. These probabilities uniquely define a probability measure P_π on the set of trajectories from which the costs incurred by π can be calculated. Indeed, the cost (or value) of π for state s is the expected cost of these

trajectories when $s_0 = s$, defined as

$$V_\pi(s) = E_\pi\left[\sum_{k=0}^{\infty} c(X_k, \pi(X_k)) \,\middle|\, X_0 = s\right],$$

where the X_k's are *random variables* that denote states of the system at different time points, distributed according to P_π, and where E_π is the expectation with respect to P_π. The function V_π is called the state evaluation function, or simply the value function, for policy π. For a stochastic shortest-path problem, it is well-defined as long as π is a proper policy, and $V_\pi(s)$ equals the expected cost to reach a goal state from state s when using policy π.

A policy π for a stochastic shortest-path problem is optimal if its value function satisfies the Bellman optimality equation:

$$V^*(s) = \begin{cases} 0 & \text{if } s \in G, \\ \min_{a \in A(s)} \{c(s,a) + \sum_{s' \in S} p(s'|s,a) V^*(s')\} & \text{otherwise.} \end{cases} \quad (1)$$

The unique solution of this functional equation, denoted V^*, is the optimal value function; hence, all optimal policies have the same value function. Given the optimal value function, one can recover an optimal policy by acting greedily with respect to the value function. A *greedy policy* with respect to a value function V is defined as follows:

$$\pi_V(s) = \operatorname*{argmin}_{a \in A(s)} \left\{ c(s,a) + \sum_{s' \in S} p(s'|s,a) V(s') \right\}.$$

Thus, the problem of finding an optimal policy for an MDP is reduced to the problem of solving the optimality equation.

There are two basic dynamic programming approaches for solving Equation (1): value iteration and policy iteration. The value iteration approach is used by all of the heuristic search algorithms we consider, and so we review it here. Starting with an initial value function V_0, satisfying $V_0(s) = 0$ for $s \in G$, value iteration computes a sequence of updated value functions by performing, at each iteration, the following *backup* for all states $s \in S$:

$$V_{n+1}(s) := \min_{a \in A(s)} \left\{ c(s,a) + \sum_{s' \in S} p(s'|s,a) V_n(s') \right\}. \quad (2)$$

For a stochastic shortest-path problem, the sequence of value functions computed by value iteration is guaranteed to converge to an optimal value function if the following conditions are satisfied: (i) a proper policy exists, and (ii) any policy that is not proper has infinite cost for some state. (Note that if all action costs are positive, any policy that is not proper has infinite cost for some state.) The algorithm described by Equation (2) is called *synchronous value iteration* since all state values are updated in parallel. A variation of this algorithm, called *asynchronous value iteration*, updates only a subset of states at each iteration. As long as every state is guaranteed to be updated infinitely often over time, convergence is still guaranteed.

The convergence of value iteration is asymptotic. In practice, value iteration is stopped when the residuals, $|V_{n+1}(s) - V_n(s)|$, for all states are sufficiently small. The *Bellman residual*, $\max_{s \in S} |V_{n+1}(s) - V_n(s)|$, can be used to bound the suboptimality of a policy or value function for discounted MDPs. For stochastic shortest-path problems, however, suboptimality bounds are not generally possible, as shown by Bertsekas and Tsitsiklis [Bertsekas and Tsitsiklis 1991], yet there is always a sufficiently small (positive) Bellman residual that yields an optimal solution.

3 Heuristic search algorithms

Traditional dynamic programming algorithms for MDPs, such as value iteration and policy iteration, solve the optimization problem for the entire state space. By contrast, heuristic search algorithms focus on finding a solution for just the states that are reachable from the start state by following an optimal policy, and use an admissible heuristic to "prune" large parts of the remaining state space. For deterministic shortest-path problems, the effectiveness of heuristic search is well-understood, especially in the AI community. For example, dynamic programming algorithms such as Dijkstra's algorithm and the Bellman-Ford algorithm compute *all* single-source shortest paths, solving the problem for every possible starting state, whereas heuristic search algorithms such as A* and IDA* compute a shortest path from a particular start state to a goal state, usually considering just a fraction of the entire state space. This is the method used to optimally solve problems such as the Rubik's Cube from arbitrary initial configurations, when the enormous size of the state space, which is 4.3×10^{19} states for Rubik's Cube [Korf 1997], renders exhaustive methods inapplicable.

In the following, we show that the strategy of heuristic search can also be effective for stochastic shortest-path problems, and, in general, MDPs. The strategy is to solve the problem only for states that are reachable from the start state by following an optimal policy. This means that a policy found by heuristic search is a partial function from the state space to the action space, sometimes called a *partial policy*. A policy π is said to be *closed* with respect to state s if it is defined over all states that can be reached from s by following policy π, and it is said to be closed with respect to the initial state (or just closed) if it is closed with respect to s_{init}. Thus, the objective of a heuristic search algorithm for MDPs is to find a partial policy that is closed with respect to the initial state and optimal. The states that are reachable from the start state by following an optimal policy are sometimes called the *relevant states* of the problem. In solving a stochastic shortest-path problem for a given initial state, it is not necessarily the case that the set of relevant states is much smaller than the entire state space, nor is it always easy to estimate its size as a fraction of the state space. But when the set of relevant states *is* much smaller than the entire state set, the heuristic search approach can have a substantial advantage, similar to the advantage heuristic search has over traditional dynamic programming algorithms in solving deterministic shortest-path problems.

Algorithm 1 RTDP with admissible heuristic h.

Let V be the empty hash table whose entries $V(s)$ are initialized to $h(s)$ as needed.
repeat
  $s := s_{init}$.
  while s is not a goal state **do**
    For each action a, set $Q(s,a) := c(s,a) + \sum_{s' \in S} p(s'|s,a)V(s')$.
    Select a best action $\mathbf{a} := \text{argmin}_{a \in A}\, Q(s,a)$.
    Update value $V(s) := Q(s,\mathbf{a})$.
    Sample the next state s' with probability $p(s'|s,\mathbf{a})$ and set $s := s'$.
  end while
until some termination condition is met.

3.1 Real-Time Dynamic Programming

The first algorithm to apply a heuristic search approach to solving MDPs is called *Real-Time Dynamic Programming (RTDP)* [Barto, Bradtke, and Singh 1995]. RTDP generalizes a heuristic search algorithm developed by Korf [Korf 1990], called *Learning Real-Time A* (LRTA*)*, by allowing state transitions to be stochastic instead of deterministic.

Except for the fact that RTDP solves a more general class of problems, it is very similar to LRTA*. Both algorithms interleave planning with execution of actions in a real or simulated environment. They perform a series of *trials*, where each trial begins with an "agent" at the start state s_{init}. The agent takes a sequence of actions where each action is selected greedily based on the current state evaluation function. The trial ends when the agent reaches a goal state. The algorithms are called "real-time" because they perform a limited amount of search in the time interval between each action. At minimum, they perform a *backup* for the current state, as defined by Equation (2), which corresponds to a one-step lookahead search; but more extensive search and backups can be performed if there is enough time. They are called "learning" algorithms because they cache state values computed in the course of the search. In an efficient implementation, a hash table is used to store the updated state values and only values for states visited during a trial are stored in the hash table. For all other states, state values are given by an admissible heuristic function h. Algorithm 1 shows pseudocode for a trial of RTDP.

The properties of RTDP generalize the properties of Korf's LRTA* algorithm, and can be summarized as follows. First, if all state values are initialized with an admissible heuristic function h, then updated state values are always admissible. Second, if there is a proper policy, a trial of RTDP cannot get trapped in a loop and must terminate in a goal state after a finite number of steps. Finally, for the set of states that is reachable from the start state by following an optimal policy, which Barto et al. call the set of *relevant states*, RTDP converges asymptotically to optimal state values and an optimal policy. These results depend on the assumptions that

(i) all immediate costs incurred by transitions from non-goal states are positive, and (ii) the initial state evaluation function is admissible, with all goal states having an initial value of zero.[1]

Although we classify RTDP as a heuristic search algorithm, it is also a dynamic programming algorithm. We consider an algorithm to be a form of dynamic programming if it solves a dynamic programming recursion such as Equation (1) and caches results for subproblems in a table, so that they can be reused without needing to be recomputed. We consider it to be a form of heuristic search if it uses an admissible heuristic and reachability analysis, beginning from a start state, to prune parts of the state space. By these definitions, LRTA* and RTDP are both dynamic programming algorithms and heuristic search algorithms, and so is A*. We still contrast heuristic search to *simple* dynamic programming, which solves the problem for the entire state space. Value iteration and policy iteration are simple dynamic programming algorithms, as are Dijkstra's algorithm and Bellman-Ford. But heuristic search algorithms can often be viewed as a form of enhanced or focused dynamic programming, and that is how we view the algorithms we consider in the rest of this survey.[2] The relationship between heuristic search and dynamic programming comes into clearer focus when we consider LAO*, another heuristic search algorithm for solving MDPs.

3.2 LAO*

Whereas RTDP generalizes LRTA*, an online heuristic search algorithm, the next algorithm we consider, LAO* [Hansen and Zilberstein 2001], generalizes the classic AO* search algorithm, which is an offline heuristic search algorithm. The 'L' in LAO* indicates that it can find solutions with *loops*, unlike AO*. Table 1 shows how various dynamic programming and heuristic search algorithms are related, based on the structure of the solutions they find. As we will see, the branching *and cyclic* behavior specified by a policy for an indefinite-horizon MDP can be represented explicitly in the form of a cyclic graph.

Both AO* and LAO* represent the search space of a planning problem as an AND/OR graph. In an AND/OR graph, an OR node represents the choice of an action and an AND node represents a set of outcomes. AND/OR graph search was

[1] Although the convergence proof given by Barto et al. depends on the assumption that all action costs are positive, Bertsekas and Tsitsiklis [Bertsekas and Tsitsiklis 1996] prove that RTDP also converges for stochastic shortest-path problems with both positive and negative action costs, given the additional assumption that all improper policies have infinite cost. If action costs are positive and negative, however, the assumption that all improper policies have infinite cost is difficult to verify. In practice, it is often more convenient to assume that all action costs are positive.

[2] Not every heuristic search algorithm is a dynamic programming algorithm. Tree-search heuristic search algorithms, in particular, do not cache the results of subproblems and thus do not qualify as dynamic programming algorithms. For example, IDA*, which explores the tree expansion of a graph, does not cache the results of subproblems and thus does not qualify as a dynamic programming algorithm. On the other hand, IDA* extended with a transposition table caches the results of subproblems and thus is a dynamic programming algorithm.

	Solution form		
	simple path	acyclic graph	cyclic graph
Dynamic programming	Dijkstra's	backwards induction	value iteration
Offline heuristic search	A*	AO*	LAO*
Online heuristic search	LRTA*	RTDP	RTDP

Table 1. Classification of dynamic programming and heuristic search algorithms.

originally developed to model problem-reduction search problems, where a problem is solved by recursively dividing it into subproblems. But it can also be used to model conditional planning problems where the state transition caused by an action is stochastic, and each possible successor state must be considered by the planner.

In AND/OR graph search, a solution is a subgraph of an AND/OR graph that is defined as follows: (i) the root node (corresponding to the start state) belongs to the solution graph, (ii) for every OR node in the solution graph, exactly one of its branches (typically, the one with the lowest cost) belongs to the solution graph, and (iii) for every AND node in the solution graph, all of its branches belong to the solution graph. A solution graph is *complete* if every directed path that begins at the root node ends at a goal node. It is a *partial solution graph* if any directed path ends at an open (i.e., unexpanded) node.

The heuristic search algorithm AO* finds an acyclic solution graph by iteratively expanding nodes on the fringe of the best partial solution graph (beginning from a partial solution graph that consists only of the root node), until the best solution graph is complete. At each step, the best partial solution graph (corresponding to a partial policy) is determined by "greedily" choosing, for each OR node, the branch (or action) with the best expected value. For conditional planning problems with stochastic state transitions, AO* solves the dynamic programming recursion of Equation (1). It does so by repeatedly alternating two steps until convergence. In the forward or *expansion step*, it expands one or more nodes on the fringe of the current best partial solution graph. In the backward or *cost-revision step*, it propagates any change in the heuristic state estimates for the states in the fringe backwards through the graph. The first step is a form of forward reachability analysis, beginning from the start state. The second step is a form of dynamic programming, using backwards induction since the graph is assumed to be acyclic. Thus, AND/OR graph heuristic search is a form of dynamic programming that is enhanced by forward reachability analysis guided by an admissible heuristic.

The classic AO* algorithm only works for problems with acyclic spaces. But stochastic planning problems, such as MDPs, often contain cycles in space and their solutions may include cycles too. To generalize AO* on these models, the key idea is to use a more general dynamic programming algorithm in the cost-revision step, such as value iteration. This simple generalization is the key difference between AO*

Algorithm 2 Improved LAO* with admissible heuristic h.

The explicit graph initially consists of the start state s_{init}.
repeat
  Depth-first traversal of states in the current best (partial) solution graph.
  for each visited state s in postorder traversal **do**
    If state s is not expanded, expand it by generating each successor state s' and initializing its value $V(s')$ to $h(s')$.
    Set $V(s) := \min_{a \in A(s)} c(s,a) + \sum_{s'} p(s'|s,a)V(s')$ and mark the best action.
  end for
until the best solution graph has no unexpanded tip state and residual $< \epsilon$.
return An ϵ-optimal solution graph.

and LAO*. However, allowing a solution to contain loops substantially increases the complexity of the cost-revision step. For AO*, the cost-revision step requires at most one update per node. For LAO*, many updates per node may be required before convergence to exact values. As a result, a naive implementation of LAO* that expands a single fringe node at a time and performs value iteration in the cost-revision step until convergence to exact values can be extremely slow.

However, a couple of simple changes create a much more efficient version of LAO*. Although Hansen and Zilberstein did not give the modified algorithm a distinct name, it has been referred to in the literature as *Improved LAO**. Recall that in its expansion step, LAO* does a depth-first traversal of the current best partial solution graph in order to identify the open nodes on its fringe, and expands one or more of the open nodes. To improve efficiency, Improved LAO* expands all open nodes on the fringe of the best current partial solution graph (yet it is easily modified to expand less or more nodes), and then, during the cost-revision step, it performs only *one* backup for each node in the current solution graph. Conveniently, both the expansion and cost-revision steps can be performed in the same depth-first traversal of the best partial solution graph, since node expansions and backups can be performed when backtracking during a depth-first traversal. Thus, the complexity of a single iteration of the expansion and cost-revision steps is bounded by the number of nodes in the current best (partial) solution graph. Algorithm 2 shows the pseudocode.

RTDP and the more efficient version of LAO* have many similarities. Principal among them, both perform backups only for states that are reachable from the start state by choosing actions greedily based on the current value function. The key difference is how they choose the order in which to visit states and perform backups. RTDP relies on stochastic exploration based on real or simulated trials (an online strategy), whereas LAO* relies on systematic depth-first traversals (an offline strategy). In fact, all of the other heuristic search algorithms we review in the rest of this article rely on one of the other of these two general strategies for

traversing the reachable state space and updating the value function.

Experiments show that Improved LAO* finds a good solution as quickly as RTDP and converges to an optimal solution much faster; faster convergence is due to its use of systematic search instead of stochastic simulation to explore the state space. The test for convergence to an optimal solution generalizes the convergence test for AO*: the best solution graph is optimal if it is complete (i.e., it does not contain any unexpanded nodes), and if state values have converged to exact values for all nodes in the best solution graph. If the state values are not exact, it is possible to bound the suboptimality of the solution by adapting the error bounds developed for value iteration.

3.3 Bounds and faster convergence

In comparing the performance of Improved LAO* and RTDP, Hansen and Zilberstein made a couple of observations that inspired subsequent improvements of RTDP. One observation was that the convergence test used by LAO* could be adapted for use by RTDP. As formulated by Barto et al., RTDP is guaranteed to converge asymptotically but does not have an explicit convergence test or a way of bounding the suboptimality of a solution. A second observation was that RTDP's slow convergence relative to Improved LAO* is due to its reliance on stochastic exploration of the state space, instead of systematic search, and its rate of convergence could be improved by exploring the state space more systematically. We next consider several improved methods for testing for convergence and increasing the rate of convergence.

Labeling solved states. Bonet and Geffner [Bonet and Geffner 2003a; Bonet and Geffner 2003b] developed a pair of related algorithms, called Labeled RTDP (LRTDP) and Heuristic Dynamic Programming (HDP), that combine both of these ideas with a third idea adopted from the original AO* algorithm: the idea of labeling 'solved' states. In the classic AO* algorithm, a state s is labeled as 'solved' if it is a goal state or if every state that is reachable from s by taking the best action at each OR node is labeled 'solved'. Labeling speeds up the search because it is unnecessary to expend search effort in parts of the solution that have already converged; AO* terminates when the start node is labeled 'solved'.

When a solution graph contains loops, however, labeling states as 'solved' cannot be done in the traditional way. It is not even guaranteed to be useful; if the start state is reachable from every other state, for example, it is not possible to label any state as 'solved' before the start state itself is labeled as 'solved'. But in many cases, a solution graph with loops has a "partly acyclic" structure. Stated precisely, the solution graph can often be decomposed into strongly-connected components, using Tarjan's well-known algorithm. In this case, the states in one strongly-connected component can be labeled as 'solved' before the states in other, predecessor components are labeled.

Tarjan's algorithm decomposes a graph into strongly-connected components in

the course of a depth-first traversal of the graph. Since Improved LAO* expands and updates the states in the current best solution graph in the course of a depth-first traversal of the graph, the two algorithms are easily combined. In fact, Bonet and Geffner [Bonet and Geffner 2003a] present their HDP algorithm as a synthesis of Tarjan's algorithm and a depth-first search algorithm, similar to the one used in Improved LAO*.

The same idea of labeling states as 'solved' can also be combined with RTDP. In Labeled RTDP (LRTDP), trials are very much like RTDP trials except that they terminate when a solved stated is reached. (Initially only the goal states are solved.) At the end of a trial, a labeling procedure is invoked for each unsolved state visited in the trial, in reverse order from the last unsolved state to the start state. For each state s, the procedure performs a depth-first traversal of the states that are reachable from s by selecting actions greedily based on the current value function. If the residuals of these states are less than a threshold ϵ, then *all* of them are labeled as 'solved'. Like AO*, Labeled RTDP terminates when the initial state is labeled as 'solved'. The labeling procedure used by LRTDP is similar to the traversal procedures used in HDP and Improved LAO*. However, the innovation of LRTDP is that instead of always traversing the solution graph from the start state, it begins the traversal at each state visited in a trial, in backwards order from the last unsolved state, which allows the convergence of states near the goal to be recognized before states near the initial state have converged.

Experiments show that LRTDP converges much faster than RTDP, and somewhat faster than Improved LAO*, in solving benchmark "racetrack" problems. In general, the amount of improvement is problem-dependent since it depends on the extent to which the solution graph decomposes into strongly-connected components. In the racetrack domain, the improvement over Improved LAO* is due to labeling states as 'solved'; the more substantial improvement over RTDP is partly due to labeling, but also due to the more systematic traversal of the state space.

Lower and upper bounds. Both LRTDP and HDP gradually reduce the Bellman residual until it falls below a threshold ϵ. If the threshold is sufficiently small, the policy is optimal. But the residual, by itself, does not bound the suboptimality of the solution. To bound its suboptimality, we need an upper bound on the value of the starting state in addition to the lower-bound values computed by heuristic search. Once a closed policy is found, an obvious way to bound its suboptimality is to evaluate the policy; its value for the start state is an upper bound that can be compared to the admissible lower-bound value computed by heuristic search. But this approach does not allow the suboptimality of an incomplete solution (one for which the start state is not yet labeled 'solved') to be bounded.

McMahan et al. [McMahan, Likhachev, and Gordon 2005] and Smith and Simmons [Smith and Simmons 2006] describe two algorithms, called *Bounded RTDP (BRTDP)* and *Focused RTDP (FRTDP)* respectively, that compute upper bounds in order to bound the suboptimality of a solution, including incomplete solutions,

and use the difference between the upper and lower bounds on state values to focus search effort. The key assumption of both algorithms is that in addition to an admissible heuristic function that returns lower bounds for any state, there is a function that returns upper bounds for any state. Every time BRTDP or FRTDP visit a state, they perform two backups: a standard RTDP backup to compute a lower-bound value and another backup to compute an upper-bound value. In simulated trials, action outcomes are determined based on their probability *and* the largest difference between the upper and lower bound values of the possible successor states, which has the effect of biasing state exploration to where it is most likely to improve the value function.

This approach has a lot of attractive properties. In particular, being able to bound the suboptimality of an incomplete solution is useful when it is computationally prohibitive to compute a policy that is closed with respect to the start state. However, the approach is based on the assumption that an upper-bound value function is available and easily computed, and this assumption may not be realistic for many stochastic shortest-path problems. For discounted MDPs, on the other hand, such bounds are easily computed, as we show in Section 4.[3]

3.4 Learning Depth-First Search

AND/OR graphs can represent the search space of problem-reduction problems and MDPs, by appropriately defining the cost of complete solution graphs, and they can also be used to represent the search space of adversarial game-playing problems, non-deterministic planning problems, and even deterministic planning problems. Bonet and Geffner [Bonet and Geffner 2005a; Bonet and Geffner 2006] describe a *Learning Depth-First Search (LDFS)* algorithm that provides a unified framework for solving search problems in these different AI models. LDFS performs iterated depth-first searches over the current best partial solution graph, enhanced with backups and labeling of 'solved' states. Bonet and Geffner show that LDFS generalizes well-known algorithms in some cases and points to novel algorithms in other cases. For deterministic planning problems, for example, they show that LDFS instantiates to IDA* with transposition tables. For game-search problems, they show that LDFS corresponds to an Alpha-Beta search algorithm with null windows called MTD [Plaat, Schaeffer, Pijls, and de Bruin 1996], which is reminiscent of Pearl's SCOUT algorithm [Pearl 1983]. For MDPs, LDFS corresponds to a version of Improved LAO* enhanced with labeling of 'solved' states. For max AND/OR search problems, LDFS instantiates to a novel algorithm that experiments show is more efficient than existing algorithms [Bonet and Geffner 2005a].

[3]Before developing FRTDP, Smith and Simmons [Smith and Simmons 2005] developed a very similar heuristic search algorithm for partially observable Markov decision processes (POMDPs) that backs up both lower-bound and upper-bound state values in AND/OR graph search. A similar AND/OR graph-search algorithm for POMDPs was described earlier by Hansen [Hansen 1998]. Since both algorithms solve discounted POMDPs, both upper and lower bounds are easily available.

3.5 Symbolic heuristic search

The algorithms we have considered so far assume a "flat" state space and enumerate states, actions, and transitions individually. For very large state spaces, it is often more convenient to adopt a structured or symbolic representation that exploits regularities to represent the same information more compactly and manipulate it more efficiently, in terms of sets of states and sets of transitions. As an example, Hoey et al. [Hoey, St-Aubin, Hu, and Boutilier 1999] show how to perform symbolic value iteration for *factored MDPs*, which are represented in a propositional language, using algebraic decision diagrams as a compact data structure. Based on their approach, Feng and Hansen [Feng and Hansen 2002; Feng, Hansen, and Zilberstein 2003] describe a symbolic LAO* algorithm and a symbolic version of RTDP for factored MDPs. Boutilier et al. [Boutilier, Reiter, and Price 2001] show how to perform symbolic dynamic programming for MDPs represented in a first-order language, and Karabaev and Skvortsova [Karabaev and Skvortsova 2005] show that symbolic heuristic search can also be performed over such MDPs.

4 Admissible heuristics

Heuristic search algorithms require admissible heuristics to prune large state spaces effectively. As advocated by Pearl, an effective and domain-independent strategy for obtaining admissible heuristics consists in optimally solving a relaxation of the problem, an MDP in our case. In this section, we review some relaxation-based heuristics for MDPs. However, we first consider admissible heuristics that are not based on relaxations. Although such heuristics are not informative, they are useful when informative heuristics cannot be easily computed.

4.1 Non-informative heuristics

For stochastic shortest-path problems where all actions incur positive costs, a simple admissible heuristic assigns the value of zero to every state, $h(s) = 0, \forall s \in S$, since zero is a lower bound on the cost of an optimal solution. Note that this heuristic is equivalent to using a zero-constant admissible heuristic for A* when solving deterministic shortest-path problems. In problems with uniform costs this is equivalent to a breadth-first search.

For the more general model of stochastic shortest-path problems that allows both negative and positive action costs, it is not possible to bound the optimal value function in such a simple way, and simple, non-informative heuristics are not readily available. But for discounted infinite-horizon MDPs, the optimal value function is easily bounded both above and below. Note that for this class of MDPs, we adopt the reward-maximization framework. Let $R^U = \max_{s \in S, a \in A(s)} r(s,a)$ denote the maximum immediate reward for an MDP and let $R^L = \min_{s \in S, a \in A(s)} r(s,a)$ denote the minimum immediate reward. For an MDP with discount factor $\gamma \in (0,1)$, the function $h(s) = R^U/1-\gamma$ is an upper bound on the optimal value function and provides admissible heuristic estimates, and the function $l(s) = R^L/1-\gamma$ is a lower

bound on the optimal value function. The time required to compute these bounds is linear in the number of states and actions, but the bounds need to be computed just once as their value does not depend on the state s.

4.2 Relaxation-based heuristics

The relaxations that are used for obtaining admissible heuristics in deterministic planning can be used for MDPs as well, as we will see. But first, we consider a relaxation that applies only to search problems with uncertain transitions. It assumes the agent can control the transition by choosing the best outcome among the set of possible outcomes of an action.

Recall from Equation (1) that the equation that characterizes the optimal value function of a stochastic shortest-path problem has the form $V^*(s) = 0$ for goal states $s \in G$, and

$$V^*(s) = \min_{a \in A(s)} \left\{ c(s,a) + \sum_{s' \in S} p(s'|s,a) V^*(s') \right\},$$

for non-goal states $s \in S \backslash G$. A lower bound on V^* is immediately obtained if the expectation in the equation is replaced by a minimization over the values of the successor states, as follows,

$$V_{min}(s) = \min_{a \in A(s)} \left\{ c(s,a) + \min_{s' \in S(s,a)} V_{min}(s') \right\},$$

where $S(s,a) = \{s' : p(s'|s,a) > 0\}$ is the subset of successor states of s through the action a. Interestingly, this equation is the optimality equation for a deterministic shortest-path problem over the graph $G_{min} = (V, E)$ where $V = S$, and there is an edge (s, s') with cost $c(s, s') = \min\{c(s,a) : p(s'|s,a) > 0, a \in A(s)\}$ for $s' \in S(s,a)$. The graph G_{min} is a *relaxation of the MDP* on which the non-deterministic outcomes of an action are separated along different deterministic actions, in a way that the agent has the ability to choose the most convenient one. If this relaxation is solved optimally, the state values $V_{min}(s)$ provide an admissible heuristic for the MDP. This relaxation is called the *min-min relaxation* of the MDP [Bonet and Geffner 2005b]; its optimal value at state s is denoted by $V_{min}(s)$.

When the number of states is relatively small and can fit in memory, the state values $V_{min}(s)$ can be obtained using Dijkstra's algorithm in time polynomial in the number of states and actions. Otherwise, the values can be obtained, as needed, using a search algorithm such as A* or IDA* on the graph G_{min}. Indeed, the state value $V_{min}(s)$ is the cost of a minimum-cost path from s to any goal state. A* and IDA* require an admissible heuristic function $h(s)$ for searching G_{min}; if nothing better is available, the non-informative heuristic $h(s) = 0$ can be used.

Given a deterministic relaxation of an MDP, such as this, another approach to computing admissible heuristics for the original MDP is based on the recognition that any admissible heuristic for the deterministic relaxation is also admissible

for the original MDP. That is, if an estimate $h(s)$ is a lower bound on the value $V_{min}(s)$, it is also a lower bound on the value $V^*(s)$ for the MDP. Therefore, we can use any method for computing admissible heuristics for deterministic shortest-path problems in order to compute admissible heuristics for the corresponding stochastic shortest-path problems. Since such methods often rely on state abstraction, the heuristics can be stored in memory even when the state space of the original problem is much too large to fit in memory.

Instead of applying relaxation methods for deterministic shortest-path problems to a deterministic relaxation of an MDP, another approach is to apply similar relaxation methods directly to the MDP. This strategy was explored by Dearden and Boutilier [Dearden and Boutilier 1997], who describe an approach to state abstraction for factored MDPs that can be used to compute admissible heuristics. Their approach ignores certain state variables of the original MDP in order to create an exponentially smaller abstract MDP that can be solved more easily. Such a relaxation can be useful when it is not desirable to abstract away all stochastic aspects of a problem.

4.3 Planning languages and heuristics

Most approaches to state abstraction for MDPs, including that of Dearden and Boutilier, assume the MDP has a factored or otherwise structured representation, instead of a "flat" representation that explicitly enumerates individual states, actions, and transitions. To allow scalability, the representation languages used by most planners are high-level languages based on propositional logic or a fragment of first-order logic, that permits the description of large problems in a succinct way; often, a problem with n states and m actions can be described with $O(\log nm)$ bits.

As an example, the PPDDL language [Younes and Littman 2004] has been used in the International Planning Competition to describe MDPs [Bryce and Buffet 2008; Gerevini, Bonet, and Givan 2006]. PPDDL is an extension of PDDL [McDermott, Ghallab, Howe, Knoblock, Ram, Veloso, Weld, and Wilkins 1998] that handles actions with non-deterministic effects and multiple initial situations. Like PDDL, it is a STRIPS language extended with types, conditional effects, and disjunctive goals and conditions.

The fifth International Planning Competition used a fragment of PPDDL, consisting of STRIPS extended with negative conditions, conditional effects and simple probabilistic effects. The fragment disallows the use of existential quantification, disjunction of conditions, nested conditional effects, and probabilistic effects inside conditional effects. What remains, nevertheless, is a simple representation language for probabilistic planning in which a large collection of challenging problems can be modeled. For our purposes, it is a particularly interesting fragment because it allows standard admissible heuristics for classical STRIPS planning to be easily adapted and thus "lifted" for probabilistic planning. In the rest of this section, we briefly present a STRIPS language extended with conditional effects, some of its

variants for probabilistic planning, and how to compute admissible heuristics for it.

A STRIPS planning problem with conditional effects (simply STRIPS) is a tuple $\langle F, I, G, O \rangle$ where F is a set of fluent symbols, $I \subseteq F$ is the initial state, $G \subseteq F$ denotes the set of goal states, and O is a set of operators. A state is a valuation of fluent symbols that is denoted by the subset of fluents are true in the state. An operator $a \in O$ consists of a precondition $Pre \subseteq F$, and a collection CE of conditional effects of the form $C \rightarrow L$, where C and L are sets of literals that denote the condition and effect of the conditional effect.

A simple probabilistic STRIPS problem (simply sp-STRIPS) is a STRIPS problem in which each operator a has a precondition Pre and a list of probabilistic outcomes of the form $\langle (p_1, CE_1), \ldots, (p_n, CE_n) \rangle$ where $p_i > 0$, $\sum_i p_i \leq 1$, and each CE_i is a set of conditional effects. In sp-STRIPS, the state that results after applying action a on state s is equal to the state that result after applying the conditional effects in CE_i on s with probability p_i, or the same state s with probability $1 - \sum_{i=1}^{n} p_i$.

In PPDDL, probabilistic effects are expressed using statements of the form

`(probabilistic` $\{$`<rational> <det-effect>`$\}^+$`)`

where `<rational>` is a rational number and `<det-effect>` is a deterministic, possibly compound, effect. The intuition is that the deterministic effect occurs with the given probability and that no effect occurs with the remaining probability. A specification without probabilistic effects can be converted in polynomial time to STRIPS. However, when there are probabilistic effects involved, it is necessary to consider all possible simultaneous executions. For example, an action that simultaneously tosses three coins can be specified as follows:

```
(:action toss-three-coins
  :parameters (c1 c2 c3 - coin)
  :precondition (and (not (tossed c1)) (not (tossed c2)) (not (tossed c3)))
  :effect (and (tossed c1)
               (tossed c2)
               (tossed c3)
               (probabilistic 1/2 (heads c1) 1/2 (tails c1))
               (probabilistic 1/2 (heads c2) 1/2 (tails c2))
               (probabilistic 1/2 (heads c3) 1/2 (tails c3))))
```

This action is not an sp-STRIPS action since its outcomes are factored along multiple probabilistic effects. An equivalent sp-STRIPS action has as precondition the same precondition but effects of the form $\langle (1/8, CE_1), (1/8, CE_2), \ldots, (1/8, CE_8) \rangle$ where each CE_i stands for a deterministic outcome of the action; e.g., $CE_1 =$ `(and (heads c1) (heads c2) (heads c3))`.

Under the assumptions that there are no probabilistic effects inside conditional effects and that there are no nested conditional effects, a probabilistic planning problem described with PPDDL can be transformed into an equivalent sp-STRIPS

problem by taking the cross products of the probabilistic effects within each action; a translation that takes exponential time in the maximum number of probabilistic effects per action. However, once in sp-STRIPS, the problem can be further relaxed into (deterministic) STRIPS by converting each action of form $\langle Pre, \langle (p_1, CE_1), \ldots, (p_n, CE_n) \rangle \rangle$ into n deterministic actions of the form $\langle Pre, CE_i \rangle$. This relaxation is the min-min relaxation now implemented at the level of the representation language, without the need to explicitly generate the state and action spaces of the MDP.

The min-min relaxation of a PPDDL problem is a deterministic planning problem whose optimal solution provides an admissible heuristic for the probabilistic planning problem. Thus, any admissible heuristic for the deterministic problem provides an admissible heuristic for the probabilistic problem. (This is the approach used in the mGPT planner for probabilistic planning [Bonet and Geffner 2005b].)

Above relaxation gives an interesting and fruitful connection with the field of (deterministic) automated planning in which the computation of domain-independent and admissible heuristics is an important area of research. Over the last decade, the field has witnessed important progresses in the development of novel and powerful heuristics that can be used for probabilistic planning.

5 Conclusions

We have shown that increased interest in the problem of planning under uncertainty has led to the development of a new class of heuristic search algorithms for these planning problems. The effectiveness of these algorithms illustrates the wide applicability of the heuristic search approach. This approach is influenced by ideas that can be traced back to some of the fundamental contributions in the field of heuristic search laid down by Pearl.

In this brief survey, we only reviewed search algorithms for the special case of the problem of planning under uncertainty in which state transitions are uncertain. Many other forms of uncertainty may need to be considered by a planner. For example, planning problems with imperfect state information are often modeled as partially observable Markov decision processes for which there are also algorithms based on heuristic search [Bonet and Geffner 2000; Bonet and Geffner 2009; Hansen 1998; Smith and Simmons 2005]. For some planning problems, there is uncertainty about the parameters of the model. For other planning problems, there is uncertainty due to the presence of multiple agents. The development of effective heuristic search algorithms for these more complex planning problems remains an important and active area of research.

References

Barto, A., S. Bradtke, and S. Singh (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence 72*(1), 81–138.

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

Bertsekas, D. and J. Tsitsiklis (1991). Analysis of stochastic shortest path problems. *Mathematics of Operations Research 16*(3), 580–595.

Bertsekas, D. and J. Tsitsiklis (1996). *Neuro-Dynamic Programming*. Belmont, Massachusetts: Athena Scientific.

Bonet, B. and H. Geffner (2000). Planning with incomplete information as heuristic search in belief space. In S. Chien, S. Kambhampati, and C. Knoblock (Eds.), *Proc. 6th Int. Conf. on Artificial Intelligence Planning and Scheduling (AIPS-00)*, Breckenridge, CO, pp. 52–61. AAAI Press.

Bonet, B. and H. Geffner (2003a). Faster heuristic search algorithms for planning with uncertainty and full feedback. In G. Gottlob and T. Walsh (Eds.), *Proc. 18th Int. Joint Conf. on Artificial Intelligence (IJCAI-03)*, Acapulco, Mexico, pp. 1233–1238. Morgan Kaufmann.

Bonet, B. and H. Geffner (2003b). Labeled RTDP: Improving the convergence of real-time dynamic programming. In E. Giunchiglia, N. Muscettola, and D. S. Nau (Eds.), *Proc. 13th Int. Conf. on Automated Planning and Scheduling (ICAPS-03)*, Trento, Italy, pp. 12–21. AAAI Press.

Bonet, B. and H. Geffner (2005a). An algorithm better than AO*? In M. M. Veloso and S. Kambhampati (Eds.), *Proc. 20th National Conf. on Artificial Intelligence (AAAI-05)*, Pittsburgh, USA, pp. 1343–1348. AAAI Press.

Bonet, B. and H. Geffner (2005b). mGPT: A probabilistic planner based on heuristic search. *Journal of Artificial Intelligence Research 24*, 933–944.

Bonet, B. and H. Geffner (2006). Learning depth-first search: A unified approach to heuristic search in deterministic and non-deterministic settings, and its application to MDPs. In D. Long, S. F. Smith, D. Borrajo, and L. McCluskey (Eds.), *Proc. 16th Int. Conf. on Automated Planning and Scheduling (ICAPS-06)*, Cumbria, UK, pp. 142–151. AAAI Press.

Bonet, B. and H. Geffner (2009). Solving POMDPs: RTDP-Bel vs. point-based algorithms. In C. Boutilier (Ed.), *Proc. 21st Int. Joint Conf. on Artificial Intelligence (IJCAI-09)*, Pasadena, California, pp. 1641–1646. AAAI Press.

Boutilier, C., T. Dean, and S. Hanks (1999). Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research 11*, 1–94.

Boutilier, C., R. Reiter, and B. Price (2001). Symbolic dynamic programming for first-order MDPs. In B. Nebel (Ed.), *Proc. 17th Int. Joint Conf. on Artificial Intelligence (IJCAI-01)*, Seattle, WA, pp. 690–697. Morgan Kaufmann.

Bryce, D. and O. Buffet (Eds.) (2008). *6th International Planning Competition: Uncertainty Part*, Sydney, Australia.

Dearden, R. and C. Boutilier (1997). Abstraction and approximate decision-theoretic planning. *Artificial Intelligence 89*, 219–283.

Edelkamp, S., J. Hoffmann, and M. Littman (Eds.) (2004). *4th International Planning Competition*, Whistler, Canada.

Feng, Z. and E. Hansen (2002). Symbolic heuristic search for factored Markov decision processes. In R. Dechter, M. Kearns, and R. S. Sutton (Eds.), *Proc. 18th National Conf. on Artificial Intelligence (AAAI-02)*, Edmonton, Canada, pp. 455–460. AAAI Press.

Feng, Z., E. Hansen, and S. Zilberstein (2003). Symbolic generalization for on-line planning. In C. Meek and U. Kjaerulff (Eds.), *Proc. 19th Conf. on Uncertainty in Artificial Intelligence (UAI-03)*, Acapulco, Mexico, pp. 209–216. Morgan Kaufmann.

Gerevini, A., B. Bonet, and R. Givan (Eds.) (2006). *5th International Planning Competition*, Cumbria, UK.

Hansen, E. (1998). Solving POMDPs by searching in policy space. In *Proc14th Conf. on Uncertainty in Artificial Intelligence (UAI-98)*, Madison, WI, pp. 211–219.

Hansen, E. and S. Zilberstein (2001). LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence 129*(1–2), 139–157.

Hoey, J., R. St-Aubin, A. Hu, and C. Boutilier (1999). SPUDD: Stochastic planning using decision diagrams. In *Proc. 15th Conf. on Uncertainty in Artificial Intelligence (UAI-99)*, Stockholm, Sweden, pp. 279–288. Morgan Kaufmann.

Karabaev, E. and O. Skvortsova (2005). A heuristic search algorithm for solving first-order MDPs. In F. Bacchus and T. Jaakkola (Eds.), *Proc21st Conf. on Uncertainty in Artificial Intelligence (UAI-05)*, Edinburgh, Scotland, pp. 292–299. AUAI Press.

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

Korf, R. (1997). Finding optimal solutions to rubik's cube using pattern databases. In B. Kuipers and B. Webber (Eds.), *Proc. 14th National Conf. on Artificial Intelligence (AAAI-97)*, Providence, RI, pp. 700–705. AAAI Press / MIT Press.

McDermott, D., M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. M. Veloso, D. Weld, and D. Wilkins (1998). PDDL – The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, New Haven, USA.

McMahan, H. B., M. Likhachev, and G. Gordon (2005). Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In L. D. Raedt and S. Wrobel (Eds.), *Proc. 22nd Int. Conf. on Machine Learning (ICML-05)*, Bonn, Germany, pp. 569–576. ACM.

Pearl, J. (1983). *Heuristics*. Morgan Kaufmann.

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

Puterman, M. (1994). *Markov Decision Processes – Discrete Stochastic Dynamic Programming*. John Wiley and Sons, Inc.

Smith, T. and R. Simmons (2005). Point-based POMDP algorithms: Improved analysis and implementation. In F. Bacchus and T. Jaakkola (Eds.), *Proc. 21st Conf. on Uncertainty in Artificial Intelligence (UAI-05)*, Edinburgh, Scotland, pp. 542–547. AUAI Press.

Smith, T. and R. G. Simmons (2006). Focused real-time dynamic programming for MDPs: Squeezing more out of a heuristic. In Y. Gil and R. J. Mooney (Eds.), *Proc. 21st National Conf. on Artificial Intelligence (AAAI-06)*, Boston, USA, pp. 1227–1232. AAAI Press.

Younes, H. and M. Littman (2004). PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects. `http://www.cs.cmu.edu/~lorens/papers/ppddl.pdf`.

2

Heuristics, Planning and Cognition

Hector Geffner

1 Introduction

In the book *Heuristics,* Pearl studies the strategies for the control of problem solving processes in human beings and machines, pondering how people manage to solve an extremely broad range of problems with so little effort, and how machines could do the same [Pearl 1983, pp. vii]. The central concept in the book, as captured in the title, are the *heuristics*: the "criteria, methods, or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal" [Pearl 1983, pp. 3]. Pearl places special emphasis on heuristics that take the form of *evaluation functions* and which provide quick but approximate estimates of the distance or cost-to-go from a given state to the goal. These heuristic evaluation functions provide the search with a sense of direction with actions resulting in states that are closer to the goal being preferred. An informative heuristic $h(s)$ in the 15-puzzle, for example, is the well known 'sum of Manhattan distances', that adds up the Manhattan distance of each tile, from its location in the state s to its goal location.

The book *Heuristics* laid the foundations for the work in automated problem solving in Artificial Intelligence (AI) and is still a basic reference in the field. On the other hand, as an account of human problem solving, the book has not been as influential. A reason for this is that while the book devotes one chapter to discuss the derivation of *heuristics,* most of the book is devoted to the formulation and analysis of heuristic search *algorithms.* Most of these algorithms, such as A* and AO*, are complete and optimal, meaning that they will find a solution if there is one, and that the solution found will have minimal cost (provided that the heuristic does not overestimate the true costs). Yet, while people excel at solving a wide variety of problems almost effortlessly, it's only in puzzle-like problems where they need to restore to search, and then, they are not particularly good at it and are even worse when solutions must be optimal.

Thus, the account of problem solving in the book exhibits a gap that has been characteristic of AI systems, that result in programs that rival the best human experts in specialized domains but are no match to children in their general problem solving abilities.

In this article, I aim to present recent work in AI Planning, a form of *domain-independent problem solving,* that builds on Pearl's work and bears on this gap.

Planners are general problem solvers aimed at solving an infinite collection of problems automatically. The problems are instances of various classes of models all of which are intractable in the worst case. In order to solve these problems effectively thus, a planner must automatically recognize and exploit their structure. This is the key challenge in planning and, more generally, in domain-independent problem solving. In planning, this challenge has been addressed *by deriving the heuristic evaluations functions automatically* from the problems, an idea explored by Pearl and developed more fully in recent planning research. The resulting domain-independent planners are not as efficient as specialized solvers but are more general, and thus, behave in a way that is closer to people. Moreover, the resulting evaluation functions often enable the solution of problems with almost no search, and appear to play the role of the 'intuitions' and 'feelings' that guide human problem solving and have been difficult to capture explicitly by means of rules. We will see indeed how such heuristic evaluation functions are defined and computed in a domain-independent fashion, and why they can be regarded as relevant from a cognitive point of view.

The organization of the article is the following. We consider in order, planning models, languages, and algorithms (Section 2), the automatic extraction of heuristic evaluation functions and other developments in planning (Sections 3 and 4), the cognitive interpretation of these heuristics (Section 5), and then, more generally, the relation between AI and Cognitive Science (Section 6).

2 Planning

Planning is an area of AI concerned with the selection of actions for achieving goals. The first AI planner and one of the first AI programs was the General Problem Solver (GPS) developed by Newell, Shaw, and Simon in the late 50's [Newell, Shaw, and Simon 1958; Newell and Simon 1963]. Since then, planning has remained a central topic in AI while changing in significant ways: on the one hand, it has become *more mathematical,* with a variety of planning problems defined and studied; on the other, it has become *more empirical,* with planning algorithms evaluated experimentally and planning competitions held periodically.

Planning can be understood as representing one of the three main approaches for *selecting the action to do next*; a problem that is central in the design of autonomous systems, called often the *control problem* in AI.

In the *programming-based approach*, the programmer solves the control problem in its head and makes the solution explicit in the program. For example, for a robot moving in an office environment, the program may say to back up when too close to a wall, to search for a door if the robot has to move to another room, etc. [Brooks 1987; Mataric 2007].

In the *learning-based approach*, the control knowledge is not provided explicitly by a programmer but is learned by trial and error, as in reinforcement learning [Sutton and Barto 1998], or by generalization from examples, as in supervised learning [Mitchell 1997].

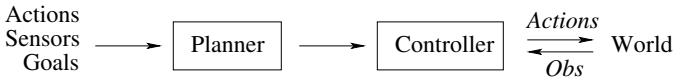


Figure 1. Planning is the model-based approach to autonomous behavior: a planner is a solver that accepts a compact model of the actions, sensors, and goals, and outputs a plan or controller that determines the action to do next given the observations.

Finally, in the *model-based approach*, the control knowledge is derived automatically from a model of the actions, sensors, and goals.

Planning is the model-based approach to autonomous behavior. A planner is a solver that accepts a model of the actions, sensors, and goals, and outputs a plan or controller that determines the action to do next given the observations gathered (Fig. 1). Planners come in a wide variety, depending on the type of model that they target [Ghallab, Nau, and Traverso 2004]. *Classical planners* address deterministic state models with full information about the initial situation, while *conformant planners* address state models with non-deterministic actions and incomplete information about the initial state. In both cases, the resulting plans are open-loop controllers that do not take observations into account. On the other hand, contingent and POMDP planners address scenarios with both uncertainty and feedback, and output genuine closed-loop controllers where the selection of actions depends on the observations gathered.

In all cases, the models are intractable in the worst case, meaning that brute force methods do not scale up to problems involving many actions and variables. Domain-independent approaches aimed at solving these models effectively must thus automatically *recognize* and *exploit* the structure of the individual problems that are given. Like in other AI models such as Constraint Satisfaction Problems and Bayesian Networks [Dechter 2003; Pearl 1988], the key to exploiting the structure of problems in planning models, is *inference.* The most common form of inference in planning is the automatic derivation of heuristic evaluation functions to guide the search. Before considering such domain-independent heuristics, however, we will make precise some of the models used in planning and the languages used for representing them.

2.1 Planning Models

Classical planning is concerned with the selection of actions in environments that are *deterministic* and whose initial state is *fully known.* The model underlying classical planning can thus be described as a state space featuring:

- a finite and discrete set of states S,
- a *known initial state* $s_0 \in S$,
- a set $S_G \subseteq S$ of goal states,

- actions $A(s) \subseteq A$ applicable in each state $s \in S$,
- a *deterministic state transition function* $f(a, s)$ for $a \in A(s)$ and $s \in S$, and
- positive *action costs* $c(a, s)$ that may depend on the action and the state.

A solution or *plan* is a sequence of actions $a_0, \ldots, a_n$ that generates a state sequence $s_0, s_1, \ldots, s_{n+1}$ such that a_i is applicable in the state s_i and results in the state $s_{i+1} = f(a_i, s_i)$, the last of which is a goal state.

The cost of a plan is the sum of the action costs, and a plan is optimal if it has minimum cost. The cost of a problem is the cost of its optimal solutions. When action costs are all 1, a situation that is common in classical planning, plan cost reduces to plan length, and the optimal plans are simply the shortest ones.

The computation of a classical plan can be cast as a *path-finding* problem in a directed graph whose nodes are the states, and whose source and target nodes are the initial state s_0 and the goal states S_G. Algorithms for solving such problems are polynomial in the number of nodes (states), which is exponential in the number of problem variables (see below). The use of heuristics for guiding the search for plans in large graphs is aimed at improving such worst case behavior.

The model underlying classical planning does not account for either uncertainty or sensing and thus gives rise to plans that represent open-loop controllers where observations play no role. Other planning models in AI take these aspects into account and give rise to different types of controllers.

Conformant planning is planning in the presence of uncertainty in the initial situation and action effects. In the resulting model, the initial state s_0 is replaced by a *set* S_0 of possible initial states, and the deterministic transition function $f(a, s)$ that maps the state s into the unique successor state $s' = f(a, s)$, is replaced by a non-deterministic transition function $F(a, s)$ that maps s into a *set* of possible successor states $s' \in F(a, s)$. A solution to such model, called a conformant plan, is an action sequence that achieves the goal with certainty for *any* possible initial state and *any* possible state transition [Goldman and Boddy 1996]. The search for conformant plans can also be cast as a path-finding problem but over a different, exponentially larger graph whose nodes represent *belief states*. In this formulation, a belief state b stands for the set of states deemed possible, the initial belief state is $b_0 = S_0$, and actions a, whether deterministic or not, map a belief state b into a unique successor belief state b_a, where $s' \in b_a$ if there is a state s in b such that $s' \in F(a, s)$ [Bonet and Geffner 2000].

Planning with sensing, often called contingent planning in AI, refers to planning in the face of both uncertainty and feedback. The model extends the one for conformant planning with a characterization of sensing. A *sensor model* expresses the relation between the observations and the true but possibly hidden states, and can be codified through a set $o \in O$ of observation tokens and a function $o(s)$ that maps states s into observation tokens. An environment is fully observable if different states give rise to different observations, i.e., $o(s) \neq o(s')$ if $s \neq s'$, and partially

observable otherwise. While the model for planning with sensing is a slight variation of the model for conformant planning, the resulting solution or plan forms are quite different as observations can and must be taken into account in the selection of actions. Indeed, solution to planning with sensing problems can be expressed equivalently as either *trees* [Weld, Anderson, and Smith 1998], *policies* mapping beliefs into actions [Bonet and Geffner 2000], or *finite-state controllers* [Bonet, Palacios, and Geffner 2009]. A finite-state controller is an automata defined by a collection of tuples of the form $\langle q, o, a, q' \rangle$ that prescribe to do action a and move to the controller state q' after getting the observation o in the controller state q.

The probabilistic versions of these models are also used in planning. The models that result when the actions have stochastic effects and the states are fully observable are the familiar Markov Decision Processes (MDPs) used in Operations Research and Control Theory [Bertsekas 1995], while the models that result when action and sensors are stochastic, are the Partial Observable MDPs (POMDPs) [Kaelbling, Littman, and Cassandra 1998].

2.2 Planning Languages

A domain-independent planner is a general solver over a class of models: classical planners are solvers over the class of basic state models where actions are deterministic and the initial state is fully known, conformant planners are solvers over the class of models where actions are non-deterministic and the initial state is partially known, and so on. In all cases, the corresponding state model that characterizes a given planning problem is not given explicitly but in a compact form, with the states associated with the values of a given set of variables.

One of the most common languages for representing classical problems is Strips, a planning language that can be traced back to the early 70's [Fikes and Nilsson 1971]. A planning problem in Strips is a tuple $P = \langle F, O, I, G \rangle$ where

- F stands for the set of relevant *variables* or *fluents*,
- O stands for the set of relevant *operators* or *actions*,
- $I \subseteq F$ stands for the *initial situation*, and
- $G \subseteq F$ stands for the *goal situation*.

In Strips, the actions $o \in O$ are represented by three sets of atoms from F called the Add, Delete, and Precondition lists, denoted as $Add(o)$, $Del(o)$, $Pre(o)$. The first, describes the atoms that the action o makes true, the second, the atoms that o makes false, and the third, the atoms that must be true in order for the action to be applicable. A Strips problem $P = \langle F, O, I, G \rangle$ encodes in compact form the state model $\mathcal{S}(P)$ where

- the states $s \in S$ are the possible *collections of atoms* from F,
- the initial state s_0 is I,

- the goal states s are those for which $G \subseteq s$,
- the actions a in $A(s)$ are the ones in O with $Prec(a) \subseteq s$,
- the state transition function is $f(a, s) = (s \setminus Del(a)) \cup Add(a)$, and
- the action costs $c(a)$ are equal to 1 by default.

The states in $S(P)$ represent the possible valuations over the boolean variables in F. Thus, if the set of variables F has cardinality $|F| = n$, the number of states in $S(P)$ is 2^n. A state s represents the valuation where the variables appearing in s are taken to be true, while the variables not appearing in s are false.

As an example, a planning domain that involves three locations l_1, l_2, and l_3, and three tasks t_1, t_2, and t_3, where t_i can be performed only at l_i, can be modeled with a set F of fluents $at(l_i)$ and $done(t_i)$, and a set O of actions $go(l_i, l_j)$ and $do(t_i)$, $i, j = 1, \ldots, 3$, with precondition, add, and delete lists

$$Pre(a) = \{at(l_i)\} \ , \ Add(a) = \{at(l_j)\} \ , \ Del(a) = \{at(l_i)\}$$

for $a = go(l_i, l_j)$, and

$$Pre(a) = \{at(l_i)\} \ , \ Add(a) = \{done(t_i)\} \ , \ Del(a) = \{\}$$

for $a = do(t_i)$. The problem of doing tasks t_1 and t_2 starting at location l_3 can then be modeled by the tuple $P = \langle F, I, O, G \rangle$ where

$$I = \{at(l_3)\} \ \text{ and } \ G = \{done(t_1), done(t_2)\} \ .$$

A solution to P is an applicable action sequence that maps the state $s_0 = I$ into a state where the goals in G are all true. In this case one such plan is the sequence

$$\pi = \{go(l_3, l_1), do(t_1), go(l_1, l_2), do(t_2)\} \ .$$

The number of states in the problem is 2^6 as there are 6 boolean variables. Still, it can be shown that many of these states are not reachable from the initial state. Indeed, the atoms $at(l_i)$ for $i = 1, 2, 3$ are mutually exclusive and exhaustive, meaning that every state reachable from s_0 makes one and only one of these atoms true. These boolean variables encode indeed the possible values of the multi-valued variable that represents the agent location.

Strips is a planning language based on variables that are boolean, yet planning languages featuring primitive multi-valued variables and richer syntactic constructs are commonly used for describing both classical and non-classical planning models [McDermott 1998; Younes, Littman, Weissman, and Asmuth 2005].

2.3 Planning Algorithms

We have presented some of the models used in domain-independent planning, and one of the languages used for describing them in compact form. We focus now on the algorithms developed for solving them.

GPS, the first AI planner introduced by Newell, Shaw, and Simon, used a technique called means-ends analysis where differences between the current state and the goal situation were identified and mapped into operators that could decrease those differences [Newell and Simon 1963]. Since then, the idea of means-ends analysis has been refined and extended in many ways, seeking planning algorithms that are *sound* (only produce plans), *complete* (produce a plan if one exists), and *effective* (scale up to large problems). By the early 90's, the state-of-the-art method was UCPOP [Penberthy and Weld 1992], an elegant algorithm based on partial-order causal link planning [Sacerdoti 1975; Tate 1977; McAllester and Rosenblitt 1991], a planning method that is sound and complete, but which doesn't scale up too well.

The situation in planning changed drastically in the middle 90's with the introduction of Graphplan [Blum and Furst 1995], a planning algorithm based on the Strips representation but which otherwise had little in common with previous approaches, and scaled up better. Graphplan works iteratively in two phases. In the first phase, Graphplan builds a *plan graph* in polynomial time, made up of a sequence of layers $F_0, A_0, \ldots, F_{n-1}, A_{n-1}, F_n$ where F_i and A_i denote sets of fluents and actions respectively. F_0 is the set of fluents true in the initial situation and n is a planning horizon, initially the index of the first layer F_i where all the goals appear. In this construction, certain pairs of actions and certain pairs of fluents are marked as mutually exclusive or mutex. The meaning of these layers and mutexes is roughly the following: if a fluent p is not in layer F_i, then no plan can achieve p in i steps or less, while if the pair p and q is in F_i but marked as mutex, then no plan can achieve p and q *jointly* in i steps or less. Graphplan makes then an attempt to extract a plan from the graph, a computation that is exponential in the worst case. If the plan extraction fails, the planning horizon n is increased by 1, the plan graph is extended one level, and the plan extraction procedure is tried again. Blum and Furst showed that the planning algorithm is sound, complete, and *optimal*, meaning that the plan obtained minimizes the number of time steps provided that certain sets of actions can be done in parallel. More importantly, they showed *experimentally* that this planning approach scaled up much better than previous approaches.

Due to the new ideas and the emphasis on the empirical evaluation of planning algorithms, Graphplan had a great influence in planning research that has seen two new approaches in recent years that scale up better than Graphplan using methods that are not specific to planning.

In the SAT approach to planning [Kautz and Selman 1996], Strips problems are converted into *satisfiability* problems expressed as a set of clauses (a formula in Conjunctive Normal Form) that are fed into state-of-the-art SAT solvers. If for some horizon n, the clauses are satisfiable, a parallel plan that solves the problem can be read from the model returned by the solver. If not, like in Graphplan, the plan horizon is increased by 1 and the process is repeated until a plan is found. The approach works well when the required horizon is not large and optimal parallel

plans are sought.

In the *heuristic search* approach [McDermott 1996; Bonet, Loerincs, and Geffner 1997], the planning problem is solved by heuristic search algorithms with heuristic evaluation functions extracted automatically from the problem encoding. In forward or progression-based planning, the state space $S(P)$ for a problem P is searched for a path connecting the initial state with a goal state. In backward or regression-based planning, plans are searched backwards from the goal. Heuristic search planners have been shown to scale up to very large problems when solutions are not required to be optimal.

The heuristic search approach has actually not only delivered performance but also an explanation for why Graphplan scaled up better than its predecessors. While not described in this form, Graphplan is a heuristic search planner using a heuristic evaluation function encoded implicitly in the planning graph, and a well known admissible search algorithm [Bonet and Geffner 2001]. The difference in performance between recent and older planning algorithms is thus the result of *inference:* while planners searched for plans blindly until Graphplan, they all search with automatically derived heuristics now, or with unit resolution and clause learning when based on the SAT formulation. Domain-independent solvers whose search is not informed by inference of some sort, do not scale up, as there are too many alternatives to choose from, with a few of them leading to the goal.

3 Domain-Independent Planning Heuristics

The main novelty in state-of-the-art planners is the use of automatically derived heuristics to guide the search for plans. In *Heuristics*, Pearl showed how heuristics such as the sum of Manhattan distances for the 15-puzzle, the Euclidian distance for Road Map finding, and the Minimum Spanning Tree for the Travelling Saleman Problem, can all be understood as optimal cost functions of suitable problem *relaxations.* Moreover, for the 15-puzzle, Pearl explicitly considered relaxations obtained mechanically from a Strips representation, showing that both the number of misplaced tiles and the sum of Manhattan distances heuristics are optimal cost functions of relaxations where some *preconditions* of the actions for moving tiles are dropped.

Pearl focused then on the conditions under which a problem relaxation is 'simple enough' so that its optimal cost can be computed in polynomial time. This research problem attracted his attention at the time, and explains his interest on the *graphical structures* underlying various types of problems, including problems of combinatorial optimization, constraint satisfaction, and probabilistic inference. One kind of structure that appeared to result in 'easy' problems in all these contexts was *trees.* Pearl and his students showed indeed that inference on probabilistic Bayesian Trees and Constraint Satisfaction Trees was *polynomial* [Pearl 1982; Dechter and Pearl 1985], even if the general problems are NP-hard (see also [Mackworth and Freuder 1985]). The notion of graphical structures underlying inference problems

and the conditions under which they render inference polynomial have been generalized since then in the notion of *treewidth,* a parameter that measures how tree-like is a graph structure [Pearl 1988; Dechter 2003].

Research on the automatic derivation of heuristics in planning builds on Pearl's intuition but takes a different path. The relaxation P^+ that underlies most current heuristics in domain-independent planning is obtained from a Strips problem P by dropping, not the preconditions, but the *delete lists.* This relaxation is quite informative but is not 'easy'; indeed finding an optimal solution to a delete-free problem P^+ is not easier from a complexity point of view than finding an optimal solution to the original problem P. On the other hand, finding *one solution* to P^+, not necessarily optimal, can be done easily, in low polynomial time. The result is that heuristics obtained from P^+ are informative but not *admissible* (they may overestimate the true cost), and hence, they can be used effectively for finding plans but not for finding *optimal* plans.

If $P(s)$ refers to a planning problem that is like $P = \langle F, I, O, G\rangle$ but with $I = s$, and $\pi(s)$ is the solution found for the delete-relaxation $P^+(s)$, the heuristic $h(s)$ that estimates the cost of the problem $P(s)$ is defined as

$$h(s) = Cost(\pi(s)) = \sum_{a \in \pi(s)} cost(a) \ .$$

The plans $\pi(s)$ for the relaxation $P^+(s)$ are called *relaxed plans*, and there have been many proposals for defining and computing them. We explain below one such method that corresponds to running Graphplan on the delete-relaxation $P^+(s)$ [Hoffmann and Nebel 2001]. In delete-free problems, Graphplan runs in polynomial time and its plan graph construction is simplified as there are no mutex relations to keep track of.

The layers F_0, A_0, F_1, ..., F_{n-1}, A_{n-1}, F_n in the plan graph for $P^+(s)$ are computed starting with $F_0 = s$, by placing in A_i, $i = 1, \ldots, n-1$, all the actions a in P whose preconditions $Pre(a)$ are in F_i, and placing in F_{i+1}, the add effects of those actions along with the fluents in F_i. This construction is terminated when the goals G are all in F_n, or when $F_n = F_{n+1}$. Then if $G \not\subseteq F_n$, $h(s) = \infty$, as it can be shown then that the relaxed problem $P^+(s)$ and the original problem $P(s)$ have no solution. Otherwise, a (relaxed) parallel plan $\pi(s)$ for $P^+(s)$ can be obtained backwards from the layer F_n by collecting the actions that add the goals, and recursively, the actions that add the preconditions of those actions that are not true in the state s.

More precisely, for $G_n = G$ and i from $n-1$ to 0, B_i is set to a minimal collection of actions in A_i that add all the atoms in $G_{i+1} \setminus F_i$, and G_i is set to $Pre(B_i) \cup (G_{i+1} \cap F_i)$ where $Pre(B_i)$ is the collection of fluents that are preconditions of actions in B_i. It can be shown then that $\pi(s) = B_0, \ldots, B_{n-1}$ is a parallel plan for the relaxation $P^+(s)$; the plan being parallel because the actions in each set B_i are assumed to be done in parallel. The heuristic $h(s)$ is then just $Cost(\pi(s))$. This

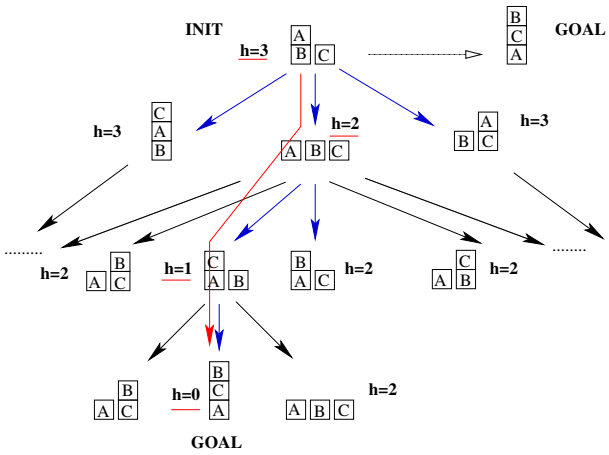


Figure 2. A simple planning problem involving three blocks with initial and goal situations I and G as shown. The actions allow to move a clear block on top of another clear block or to the table. A plan for the problem is a path that connects I with G in the directed graph partially shown. In this example, the plan can be found greedily by taking in each state s, starting with $s = I$, the action that results in a state s' that is closer to the goal according to the heuristic. The heuristic values (shown) are derived automatically from the problem as described in the text.

is indeed the heuristic introduced in the FF planner [Hoffmann and Nebel 2001], which is suitable when action costs are uniform. For non-uniform action costs, other heuristics are more convenient [Keyder and Geffner 2008].

4 Meaning of Domain-Independent Heuristics

In order to illustrate the meaning and derivation of domain-independent heuristics, let us consider the example shown in Fig. 2, where blocks a, b, and c initially arranged so that a is on b, and b and c are on the table, must be rearranged so that b is on c, and c is on a. The actions allow to move a clear block (a block with no block on top) on top of another clear block or to the table. The problem can be expressed as a Strips problem $P = \langle F, I, O, G \rangle$ with a set of atoms F given by $on(x, y)$, $ontable(x)$, and $clear(x)$, where x and y range over the block labels a, b, and c. In the heuristic search approach to planning, the solution to P becomes a path-finding problem in the directed graph associated with the state model $S(P)$, where the nodes stand for the states in $S(P)$, and the actions $a \in O$ are mapped into edges connecting a state s with a state s' when a is applicable in s and maps s into s'.

The Blocks World is simple for people, but until recently, not so simple for *domain-independent* planners. Indeed, the size of the graph to search is exponential in the number of blocks n, with $n!$ possible towers of n blocks, and additional

combinations of shorter towers.

Figure 2 shows the search that results from a planner using the heuristic described above, whose value $h(s)$ for each of the states in the graph is shown. All action costs are assumed to be 1. With the heuristic shown, the solution to the problem can be found with no search at all by just selecting in each state s the action that leads to the state s' that is closest to the goal (lowest heuristic value). In the initial state, this action is the one that places block a on the table, in the following state, the action that places c on a, and so on.

In order to understand the numbers shown in the figure, let us see how the value $h(s) = 3$ for the initial state s is derived. The heuristic $h(s)$ is $|\pi(s)|$ where $\pi(s)$ is the plan found for the relaxation $P^+(s)$. The relaxed plan $\pi(s)$ is obtained by constructing first the layered graph F_0, A_0, ..., F_{n-1}, A_{n-1}, F_n, where $n > 0$ as none of the goals $on(b, c)$ and $on(c, a)$ are in $F_0 = s$. The actions in A_0 are the actions applicable given the atoms in F_0, i.e., the actions a with $Pre(a) \subseteq F_0$. This set includes the actions of moving c to a, a to c, and a to the table, but does not include actions that move b as the precondition $clear(b)$ is not part of F_0. The set F_1 extends F_0 with all the atoms added by the actions in A_0, and includes $on(c, a)$, $on(a, c)$, $ontable(a)$, and $clear(b)$, but not the goal $on(b, c)$. Yet with $clear(b)$ and $clear(c)$ in F_1, the action for moving b to c appears in layer A_1, and therefore, the other goal atom $on(b, c)$ appears in F_2. By collecting the actions that first add the goal atoms $on(c, a)$ and $on(b, c)$, and recursively, the preconditions of those actions that are not in s, a relaxed plan $\pi(s)$ with 3 actions is obtained so that $h(s) = 3$. There are several choices for the actions in $\pi(s)$ that result from the way ties in the plan extraction procedure are broken. One possible relaxed plan involves moving a to the table and c to a in the first step, and b to c in the second step. Another involves moving a to c and c to a first, and then b to c.

It is important to notice that fluent layers such as F_1 in the plan graph do not represent any 'real' states in the original problem P as they include atoms pairs like $on(a, c)$ and $on(c, a)$ that cannot be achieved jointly in any state s' reachable from the initial state. The layer F_1 is instead an *abstraction* that *approximates* the set of all states reachable in one step from the initial state by taking their union. This approximation implies that finding an atom p in a layer F_n with $n > 1$ is no guarantee that there is a real plan for p in $P(s)$ that achieves p in n time steps, rather than one such parallel plan exists in the relaxation. Similarly, the relaxed plans $\pi(s)$ obtained above are quite 'meaningless'; they move a to the table or to c at the same time that they move c to a. Yet, these 'meaningless' relaxed plans $\pi(s)$ yield the heuristic values $h(s)$ that provide the search with a very meaningful and effective sense of direction.

Let us finally point out that the computation of the domain-independent heuristic $h(s)$ results in valuable information that goes beyond the *numbers* $h(s)$. Indeed, from the computation of the heuristic value $h(s)$, it is possible to determine the actions applicable in the state s that are most relevant to the goal, and then focus

on the evaluation of the states that result from those actions only. This type of *action pruning* has been shown to be quite effective [Hoffmann and Nebel 2001], and in slightly different form is part of state-of-the-art planners [Richter, Helmert, and Westphal 2008].

5 Other Developments in Planning

Domain-independent planning is concerned with non-classical models also where information about the initial situation is incomplete, actions may have non-deterministic effects, and states may be fully or partially observable. A number of *native solvers* for such models, that include Markov Decision Processes (MDPs) and Partially Observable MDPs have been developed, and progress in the area has been considerable too. Moreover, many of these solvers are also based on *heuristic search* methods (see the article by Bonet and Hansen in this volume). I will not review this literature here but focus instead on two ways in which the results obtained for *classical planning* are relevant to such richer settings too.

First, it's often possible to plan under *uncertainty* without having to model the uncertainty explicitly. This is well known by control engineers that normally design closed-loop controllers for stochastic systems ignoring the 'noise'. Indeed, the error in the model is compensated by the feedback loop. In planning, where non-linear models are considered, the same simplification works too. For instance, in a Blocks World where the action of moving a block may fail, an effective closed-loop policy can be obtained by replanning from the current state when things didn't progress as predicted by the simplified model. Indeed, the planner that did best in the first probabilistic planning competition [Younes, Littman, Weissman, and Asmuth 2005], was not an MDP planner, but a classical replanner of this type. Of course, this approach is not suitable when it may be hard or impossible to recover from failures, or when the system state is not fully observable. In everyday planning, however, such cases may be the exception.

Second, it has been recently shown that it's often possible to efficiently transform problems featuring uncertainty and sensing into classical planning problems that do not. For example, problems P involving uncertainty in the initial situation and no sensing, namely conformant planning problems, can be compiled into classical problems $K(P)$ by adding new actions and fluents that express conditionals [Palacios and Geffner 2007]. The translation from the conformant problem P into the classical problem $K(P)$ is sound and complete, and provided that a *width* parameter defined over P is bounded, it is polynomial too. The result is that the conformant plans for P can be read from the plans for $K(P)$ that can be computed using a classical planner. Moreover, this technique has been recently used for deriving *finite-state controllers* that solve problems featuring both incomplete information and sensing [Bonet, Palacios, and Geffner 2009]. A finite-state controller is an automata that given the current (controller) state and the current observation selects an action and updates the controller state, and so on, until reaching the

goal. Figure 3 shows one such problem (left) and the resulting controller (right). The problem, motivated by the work on deictic representations in the selection of actions [Chapman 1989; Ballard, Hayhoe, Pook, and Rao 1997], is about placing a visual marker on top of a green block in a blocks-world scene where the location of the green blocks is not known. The visual marker, initially at the lower left corner of the scene (shown as an eye), can be moved in the four directions, one cell at a time. The observations are whether the cell beneath the marker is empty ('C'), a non-green block ('B'), or a green block ('G'), and whether it is on the table ('T') or not ('-'). The controller shown on the right has been derived by running a classical planner over a classical problem obtained by an automatic translation from the original problem that involves both uncertainty and sensing. In the figure, the controller states q_i are shown in circles while the label o/a on an edge connecting two states q to q' means to do action a when observing o in q and then switching to q'. In the classical planning problem obtained from the translation, the actions are tuples $(f_q, f_o, a, f_{q'})$ whose effects are those of the action a but conditional on the fluents f_q and f_o representing the controller state q and observation o being true. In such a case, the fluent $f_{q'}$ representing the controller state q' is made true and f_q is made false. The two appealing features of this formulation is that the resulting classical plans encode very succint closed-loop controllers, and that these controllers are quite general. Indeed, the controller shown in the figure not only solves the problem for the configuration of blocks shown, but for *any configuration* involving *any number of blocks.* The controller prescribes to move the 'eye' up until there are no blocks, then to move it down until reaching the table and right, and to repeat this process until a green block is found ('G'). Likewise, the 'eye' must move right when there are no blocks in a given spot (both 'T' and 'C' observed). See [Bonet, Palacios, and Geffner 2009] for details.

6 Heuristics and Cognition

Heuristic evaluation functions are used also in other settings such as Chess playing programs [Pearl 1983] and reinforcement learning [Sutton and Barto 1998]. The difference between evaluations functions in Chess, reinforcement learning, and domain-independent planning mimic actually quite closely the relation among the three approaches to action selection mentioned in the introduction: programming-based, learning-based, and model-based. Indeed, the evaluation functions are programmed by hand in Chess, are learned by trial-and-error in reinforcement learning, and are derived from a (relaxed) model in domain-independent planning.

Heuristic evaluation functions in reinforcement learning, called simply valuation functions, are computed by stochastic sampling and dynamic programming updates. This is a *model-free method* that has been shown to be effective in low-level tasks that do not involve large state spaces, and which provides an accurate account of learning in the brain [Schultz, Dayan, and Montague 1997].

Heuristic evaluation functions as used in domain-independent planning are com-

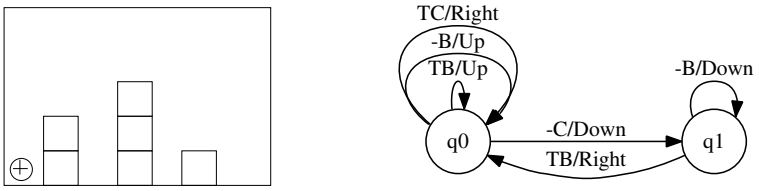


Figure 3. *Left:* The visual marker shown as an 'eye' must be placed on a green block in the blocks-world scene shown, where the locations of the green blocks are not known. The visual marker can be moved in the four directions, one cell at a time. The observations are whether the cell beneath the marker is empty ('C'), a non-green block ('B'), or a green block ('G'), and whether the marker is on the table ('T') or not ('-'). *Right:* The controller derived for this problem using a classical planner over a suitable automatic transformation. The controller states q_i are shown in circles while the label o/a on an edge connecting q to q' means to do a when observing o in q switching then to q'. The controller works not only for the problem instance shown on the left, but for *any* instance resulting from changes in the configuration or in the number of blocks.

puted by *model-based methods* where suitable relaxations are solved from scratch. The technique has been shown to work over large problems involving hundred of actions and fluents. Here I want to argue these methods also have features that make them interesting from a cognitive point of view as a plausible basis for an account of 'feelings', 'emotions', or 'appraisals' in high-level human problem solving. I focus on three of these features.

First, domain-independent heuristics are fast (low-polynomial time) and effective, as the 'fast and frugal' heuristics advocated by Gigerenzer and others [Gigerenzer and Todd 1999; Gigerenzer 2007], and yet, they are general too: they apply indeed to all the problems that fit the (classical planning) model and to problems that can be cast in that form (like the visual-marker problem above).

Second, the derivation of these heuristics sheds light on why appraisals may be opaque from a cognitive point of view, and thus not conscious. This is because the heuristic values are obtained from a relaxed model where the meaning of the symbols is different than the meaning of the symbols in the 'true' model. For example, the action of moving an object from one place to another, deletes the old place in the true model but not in the delete-relaxation where an object can thus appear in multiple places at the same time. Thus, if the agent selecting the actions with the resulting heuristic does not have access to the relaxation, it won't be able to explain how the heuristic evaluations are produced nor what they stand for. The importance of the unconscious in everyday cognition is a topic that has been receiving increased attention in recent years, with conscious, deliberate reasoning, appearing to rely heavily on unconscious processing and representing just the tip of the 'cognitive iceberg' [Wilson 2002; Hassin, Uleman, and Bargh 2005; Evans

2008]. While this is evident in vision and natural language processing, where it is clear that one does not have access to how one 'sees' or 'understands', this is likely to be true in most cognitive tasks, including apparently simple problems such as the Blocks World where our ability to find reasons for the actions selected, does not explain how such actions are selected in the first place. In this sense, the focus of cognitive psychology on puzzles such as the Tower of Hanoi may be misplaced: simple problems, such as the Blocks World, are not simple for *domain-independent solvers,* and there is no question that people are capable of solving domains that they have never seen where the combinatorics would defy a naive, blind solver.

Third, the heuristics provide the agent with a sense of direction or 'gut feelings' that guide the action selection in the presence of many alternatives, while avoiding an infinite regress in the decision process. Indeed, emotions long held to interfere with the decision process and rationality, are now widely perceived as a requisite in contexts where it is not possible to consider all alternatives. Emotions and gut feelings are thus perceived as the 'invisible hand' that successfully guides us out of these mental labyrinths [Ketelaar and Todd 2001; Evans 2002].[1] The 'rationality of the emotions' have been defended on theoretical grounds by philosophers [De Sousa 1990; Elster 1999], and on empirical grounds by neuroscientists that have studied the impairments in the decision process that result from lesions in the frontal lobes [Damasio 1995]. The link between emotions and evaluation functions, point to their computational role as well.

While emotions are currently thought as providing the appraisals that are necessary for navigating in a complex world, there are actually very few accounts of *how such appraisals may be computed.* Reinforcement learning methods provide one such account that works well in low level tasks without requiring a model. Heuristic planning methods provide another account that works well in high-level tasks where the model is known. Moreover, as discussed above, heuristic planning methods do not only provide an account of the appraisals, but also of the actions that are worth evaluating. These are the actions a in the state s that are deemed relevant to the goal in the computation of the heuristic $h(s)$; the so-called helpful actions [Hoffmann and Nebel 2001]. This form of *action pruning* may account for a key difference between programs and humans in games such as Chess: while the former consider all possible moves and responses (up to a certain depth), the latter focus on the analysis and evaluation of a few moves and countermoves. Domain-independent heuristics can account in principle for both the focus and the evaluation, the latter in the *value* of the heuristic function $h(s)$, the former in its *structure.*

[1] Some philosophers and cognitive scientists refer to this combinatorial problem as the 'frame problem' in AI. This terminology, however, is not accurate. The frame problem in AI [McCarthy and Hayes 1969] refers to the problem that arises in logical accounts of actions and change where the description of the action effects does not suffice to capture what does not change. E.g., the number of chairs in the room does not change if the bell rings. The frame problem is the problem of capturing what does not change from a concise logical description of what changes [Ford and Pylyshyn 1996].

7 AI and Cognitive Science: Past and Future

Pearl's ideas on the mechanical discovery of heuristics has received renewed attention in the area of domain-independent planning where heuristic evaluation functions, derived automatically from the problem encoding, are used to guide the search for plans in large spaces. Heuristic search planners are powerful domain-independent solvers that have been *empirically tested* over many large domains involving hundred of actions and variables.

The developments in planning parallel those in other areas of AI and bear on the relevance of Artificial Intelligence to the understanding of the human mind. AI and Cognitive Science were twin disciplines until the 80's, with AI looking to the human mind for inspiration, and Cognitive Science looking to computation as a language for modeling. The relationship between AI and Cognitive Science has changed, however, and the two disciplines do not appear to be that close now. Below, I go over some of the relevant changes that explain this divorce, and explain why, in spite to them, AI remains and will likely remain critically relevant for understanding the human mind, a premise that underlies and motivates the work of Judea Pearl and others AI scientists.

A lot of work in AI until the 80's was about writing programs capable of displaying intelligence over ill-defined problems, either by appealing to introspection or by interviewing an expert. Many good ideas came out from this work, yet few have had a lasting scientific value. The methodological problem with the 'knowledge-based' approach in AI was that the resulting programs were not robust and they always appeared to be missing critical knowledge; either declarative (e.g., that men don't get pregnant), procedural (e.g., which rule or action to apply next), or both. This situation led to an impasse in the 80's, and to many debates and criticisms, like that 'good old fashioned AI' is 'rule application' but human intelligence is not [Haugeland 1993], that representation is not needed for intelligent behavior and gets in the way [Brooks 1991], that subsymbolic neural networks and genetic algorithms are the way to go [Rumelhart and McClelland 1986; Holland 1992], etc.

In part due to the perceived limitations of the knowledge-based approach and the criticisms, and in part due to its own evolution, mainstream AI has changed substantially since the 80's. One of the key methodological changes is that many researchers have moved from the early paradigm of *writing programs for ill-defined problems* to *writing solvers for well-defined mathematical models*. These models include Constraint Satisfaction Problems, Strips Planning, Bayesian Networks and Partially Observable Markov Decision Processes, among others. Solvers are programs that take a compact description of a particular model instance (a planning problem, a CSP instance, and so on) and automatically compute its solution. Unlike the early AI programs, solvers are *general* as they must deal with any problem that fits the model (any instance). Moreover, some of these models, like POMDPs, are extremely expressive. The challenge in this research agenda is mainly *com-*

putational: how to make these domain-independent solvers scale up to large and interesting problems given that all these models are intractable in the worst case. Work in these areas has uncovered techniques that accomplish this by automatically recognizing and exploiting the structure of the problems at hand. In planning, these techniques have to do with the automatic derivation and use of heuristic evaluation functions; in SAT and CSPs, with constraint propagation and learning, while in CSPs and Bayesian Networks, with the use of the underlying graphical structure.

The relevance of the early work in AI to Cognitive Science was based on *intuition:* programs provided a way for specifying intuitions precisely and for trying them out. The more recent work on *domain-independent solvers* is more technical and experimental, and is focused not on reproducing intuitions but on *scalability.* This may give the impression, confirmed by the current literature, that recent work in AI is less relevant to Cognitive Science than work in the past. This impression, however, may prove wrong on at least two grounds. First, intuition is not what it used to be, and it is now regarded as the tip of an iceberg whose bulk is made of massive amounts of shallow, fast, but unconscious inference mechanisms that cannot be rendered explicit in the form of rules [Wilson 2002; Hassin, Uleman, and Bargh 2005; Gigerenzer 2007]. Second, whatever these mechanisms are, they appear to work pretty well and to scale up. This is no small feat, given that most methods, whether intuitive or not, do not. Indeed, if the techniques that really scale up are not that many, a plausible conjecture at this point, it may well be the case that *the twin goals of accounting reliably for the intuitions and of scaling up* have a large overlap. By focusing then on the study of meaningful models and the computational methods for dealing with them *effectively,* AI may prove its relevance to human cognition in ways that may go well beyond the rules, cognitive architectures, and knowledge structures of the 80's. Human Cognition, indeed, still provides the inspiration and motivation for a lot of research in AI. The use of Bayesian Networks in Development Psychology for understanding how children acquire and use causal relations [Gopnik, Glymour, Sobel, Schulz, , Kushnir, and Danks 2004], and the use of Reinforcement Learning algorithms in Neuroscience for interpreting the activity of dopamine cells in the brain [Schultz, Dayan, and Montague 1997], are two examples of general AI techniques that have made it recently into Cognitive Science. As AI focuses on models and solvers able to scale up, more techniques are likely to follow. One such candidate is the automatic derivation of heuristic functions as used in planning, which like the research on Bayesian Networks, owes a lot to the work of Judea Pearl.

References

Ballard, D., M. Hayhoe, P. Pook, and R. Rao (1997). Deictic codes for the embodiment of cognition. *Behavioral and Brain Sciences 20*(4), 723–742.

Bertsekas, D. (1995). *Dynamic Programming and Optimal Control, Vols 1 and 2.*

Athena Scientific.

Blum, A. and M. Furst (1995). Fast planning through planning graph analysis. In *Proceedings of IJCAI-95*, pp. 1636–1642. Morgan Kaufmann.

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

Bonet, B. and H. Geffner (2001). Planning as heuristic search. *Artificial Intelligence 129*(1–2), 5–33.

Bonet, B., G. Loerincs, and H. Geffner (1997). A robust and fast action selection mechanism for planning. In *Proceedings of AAAI-97*, pp. 714–719. MIT Press.

Bonet, B., H. Palacios, and H. Geffner (2009). Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *Proc. Int. Conf. on Automated Planning and Scheduling (ICAPS-09)*.

Brooks, R. (1987). A robust layered control system for a mobile robot. *IEEE J. of Robotics and Automation 2*, 14–27.

Brooks, R. (1991). Intelligence without representation. *Artificial Intelligence 47*(1–2), 139–159.

Chapman, D. (1989). Penguins can make cake. *AI magazine 10*(4), 45–50.

Damasio, A. (1995). *Descartes' Error: Emotion, Reason, and the Human Brain*. Quill.

De Sousa, R. (1990). *The rationality of emotion*. MIT Press.

Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann.

Dechter, R. and J. Pearl (1985). The anatomy of easy problems: a constraint-satisfaction formulation. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI-85)*, pp. 1066–1072.

Elster, J. (1999). *Alchemies of the Mind: Rationality and the Emotions*. Cambridge University Press.

Evans, D. (2002). The search hypothesis of emotion. *British J. Phil. Science 53*, 497–509.

Evans, J. (2008). Dual-processing accounts of reasoning, judgment, and social cognition. *Annual Review of Pschycology 59*, 255–258.

Fikes, R. and N. Nilsson (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence 1*, 27–120.

Ford, K. and Z. Pylyshyn (1996). *The robot's dilemma revisited: the frame problem in artificial intelligence*. Ablex Publishing.

Ghallab, M., D. Nau, and P. Traverso (2004). *Automated Planning: theory and practice*. Morgan Kaufmann.

Gigerenzer, G. (2007). *Gut feelings: The intelligence of the unconscious*. Viking Books.

Gigerenzer, G. and P. Todd (1999). *Simple Heuristics that Make Us Smart*. Oxford University Press.

Goldman, R. P. and M. S. Boddy (1996). Expressive planning and explicit knowledge. In *Proc. AIPS-1996*.

Gopnik, A., C. Glymour, D. Sobel, L. Schulz, , T. Kushnir, and D. Danks (2004). A theory of causal learning in children: Causal maps and Bayes nets. *Psychological Review 111*(1), 3–31.

Hassin, R., J. Uleman, and J. Bargh (2005). *The new unconscious*. Oxford University Press, USA.

Haugeland, J. (1993). *Artificial intelligence: The very idea*. MIT press.

Hoffmann, J. and B. Nebel (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research 14*, 253–302.

Holland, J. (1992). *Adaptation in natural and artificial systems*. MIT Press.

Kaelbling, L., M. Littman, and T. Cassandra (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence 101*(1–2), 99–134.

Kautz, H. and B. Selman (1996). Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. AAAI*, pp. 1194–1201.

Ketelaar, T. and P. M. Todd (2001). Framing our thoughts: Evolutionary psychology's answer to the computational mind's dilemma. In H. Holcomb (Ed.), *Conceptual Challenges in Evolutionary Psychology*. Kluwer.

Keyder, E. and H. Geffner (2008). Heuristics for planning with action costs revisited. In *Proc. ECAI-08*, pp. 588–592.

Mackworth, A. and E. C. Freuder (1985). The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence 25*(1), 65–74.

Mataric, M. J. (2007). *The Robotics Primer*. MIT Press.

McAllester, D. and D. Rosenblitt (1991). Systematic nonlinear planning. In *Proceedings of AAAI-91*, Anaheim, CA, pp. 634–639. AAAI Press.

McCarthy, J. and P. Hayes (1969). Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*, pp. 463–502. Edinburgh University Press.

McDermott, D. (1996). A heuristic estimator for means-ends analysis in planning. In *Proc. Third Int. Conf. on AI Planning Systems (AIPS-96)*.

McDermott, D. (1998). PDDL – the planning domain definition language. At `http://ftp.cs.yale.edu/pub/mcdermott`.

Mitchell, T. (1997). *Machine Learning*. McGraw-Hill.

Newell, A., J. Shaw, and H. Simon (1958). Elements of a theory of human problem solving. *Psychology Review 23*, 342–343.

Newell, A. and H. Simon (1963). GPS: a program that simulates human thought. In E. Feigenbaum and J. Feldman (Eds.), *Computers and Thought*, pp. 279–293. McGraw Hill.

Palacios, H. and H. Geffner (2007). From conformant into classical planning: Efficient translations that may be complete too. In *Proc. 17th Int. Conf. on Planning and Scheduling (ICAPS-07)*.

Pearl, J. (1982). Reverend Bayes on inference engines: A distributed hierarchical approach. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-82)*, pp. 133–136.

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

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann.

Penberthy, J. and D. Weld (1992). UCPOP: A sound, complete, partially order planner for ADL. In *Proceedings KR'92*.

Richter, S., M. Helmert, and M. Westphal (2008). Landmarks revisited. In *Proc. AAAI*, pp. 975–982.

Rumelhart, D. and J. McClelland (Eds.) (1986). *Parallel distributed processing: explorations in the microstructure of cognition. Vol. 1.* MIT Press.

Sacerdoti, E. (1975). The nonlinear nature of plans. In *Proceedings of IJCAI-75*, Tbilisi, Georgia, pp. 206–214.

Schultz, W., P. Dayan, and P. Montague (1997). A neural substrate of prediction and reward. *Science 275*(5306), 1593–1599.

Sutton, R. and A. Barto (1998). *Introduction to Reinforcement Learning*. MIT Press.

Tate, A. (1977). Generating project networks. In *Proc. IJCAI*, pp. 888–893.

Weld, D., C. Anderson, and D. Smith (1998). Extending Graphplan to handle uncertainty and sensing actions. In *Proc. AAAI-98*, pp. 897–904. AAAI Press.

Wilson, T. (2002). *Strangers to ourselves*. Belknap Press.

Younes, H., M. Littman, D. Weissman, and J. Asmuth (2005). The first probabilistic track of the international planning competition. *Journal of Artificial Intelligence Research 24*, 851–887.

3

Mechanical Generation of Admissible Heuristics

Robert Holte, Jonathan Schaeffer, and Ariel Felner

1 Introduction

This chapter takes its title from Section 4.2 of Judea Pearl's landmark book *Heuristics* [Pearl 1984], and explores how the vision outlined there has unfolded in the quarter-century since its appearance. As the book's title suggests, it is an in-depth summary of classical artificial intelligence (AI) heuristic search, a subject to which Pearl and his colleagues contributed substantially in the early 1980s.

The purpose of heuristic search is to find a least-cost path in a state space from a given start state to a goal state. In principle, such problems can be solved by classical shortest path algorithms, such as Dijkstra's algorithm [Dijkstra 1959], but in practice the state spaces of interest in AI are far too large to be solved in this way. One of the seminal insights in AI was recognizing that even extremely large search problems can be solved quickly if the search algorithm is provided with additional information in the form of a heuristic function $h(s)$ that estimates the distance from any given state s to the nearest goal state [Doran and Michie 1966; Hart, Nilsson, and Raphael 1968]. A heuristic function $h(s)$ is said to be *admissible* if, for every state s, $h(s)$ is a lower bound on the true cost of reaching the nearest goal from state s. Admissibility is desirable because it guarantees the optimality of the solution found by the most widely-used heuristic search algorithms.

Most of the chapters in *Heuristics* contain mathematically rigorous definitions and analysis. In contrast, Chapter 4 offers a conceptual account of where heuristic functions come from, and a vision of how one might create algorithms for automatically generating effective heuristics from a problem description. An early version of the chapter had been published previously in the widely circulated *AI Magazine* [Pearl 1983].

Chapter 4's key idea is that distances in the given state space can be estimated by computing exact distances in a "simplified" version of the state space. There are many different ways a state space can be simplified. Pearl focused almost exclusively on *relaxation*, which is done by weakening or eliminating one or more of the conditions that restrict how one is allowed to move from one state to another. For example, to estimate the driving distance between two cities, one can ignore the constraint that driving must be done on roads. In this relaxed version of the problem, the distance between two cities is simply the straight-line distance. It is

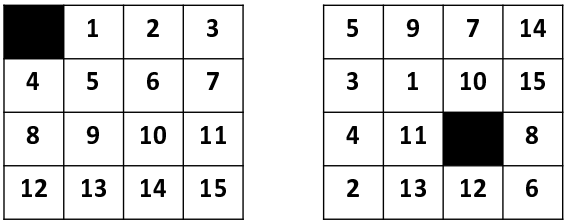


Figure 1. 15-puzzle

easy to see, in general, that distances in a relaxed space cannot exceed distances in the given state space, and therefore the heuristic functions defined in this way are guaranteed to be admissible. An alternate way of looking at this is to view the elimination of conditions as equivalent to adding new edges to the search graph. Therefore, optimal solutions to the relaxed graph (with the additional edges) must be a lower bound on the solution to the original problem.

As a second example of relaxation, consider the 15-puzzle shown in Figure 1, which consists of a set of tiles numbered 1-15 placed in a 4×4 grid, leaving one square in the grid unoccupied (called the "blank" and shown as a black square). The only moves that are permitted are to slide a tile that is adjacent to the blank into the blank position, effectively exchanging the tile with the blank. For example, four moves are possible in the right-hand side of Figure 1: tile 10 can be moved down, tile 11 can be moved right, tile 8 can be moved left, and tile 12 can be moved up. To solve the puzzle is to find a sequence of moves that transforms a given scrambled state (right side of Figure 1) into a goal state (such as the one on the left). One possible relaxation of the 15-puzzle state space can be defined by removing the restriction that a tile must be adjacent to the blank to be moveable. In this relaxation any tile can move from its current position to any adjacent position at any time, regardless of whether the adjacent position is occupied or not. The number of moves required to solve this relaxed version (called the Manhattan Distance) is clearly less than or equal to the number of moves required to solve the 15-puzzle itself. Note that in this case the relaxed state space has many more states than the original 15-puzzle (many tiles can now occupy a single location) but it is easier to solve, at least for humans (tiles move entirely independently of one another).

Pearl observes that in AI a state space is almost always defined implicitly by a set of operators that describe a successor relation between states. Each operator has a precondition defining the states to which it can be applied and a postcondition describing how the operator changes the values of the variables used to describe a state. This implies that relaxing a state space description by eliminating one or more preconditions is a simple syntactic operation, and the set of all possible relaxations of a state space description (by eliminating combinations of preconditions) is well-defined and, in fact, easy to enumerate. Hence it is entirely feasible for a mechanical

system to generate heuristic functions and, indeed, to search through the space of heuristic functions defined by eliminating preconditions in all possible ways.

The mechanical search through a space of heuristic functions has as its goal, in Pearl's view, a heuristic function with two properties. First, the heuristic function should return values that are as close to the true distances as possible (Chapter 6 in *Heuristics* justifies this). Second, the heuristic function must be efficiently computable, otherwise the reduction in search effort that the heuristic function produces might be outweighed by the increase in computation time caused by the calculation of the heuristic function. Pearl saw the second requirement as the more difficult to detect automatically and proposed that mechanically-recognizable forms of decomposability of the relaxed state space would be the key to mechanically generating efficiently-computable heuristic functions. Pearl recognized that the search for a good heuristic function might itself be quite time-consuming, but argued that this cost was justified because it could be amortized over an arbitrarily large number of problem instances that could all be solved much more efficiently using the same heuristic function.

The preceding paragraphs summarize Pearl's vision for how effective heuristics might be generated automatically from a state space description. The remainder of our chapter contains a brief look at the research efforts directed towards realizing Pearl's vision. We conclude that Pearl correctly anticipated a fundamental breakthrough in heuristic search in the general terms he set out in Chapter 4 of *Heuristics* although not in all of its specifics. Our discussion is informal and the ideas presented and their references are illustrative, not exhaustive.

2 The Vision Emerges

The idea of using a solution in a simplified state space to guide the search for a solution in the given state space dates to the early days of AI [Minsky 1963] and was first implemented and shown to be effective in the ABSTRIPS system [Sacerdoti 1974]. However, these early methods did not use the cost of the solution in the simplified space as a heuristic function; they used the solution itself as a skeleton which was to be *refined* into a solution in the given state space by inserting additional operators.

The idea of using distances in a simplified space as heuristic estimates of distances in the given state space came later. It did not originate with Judea Pearl (in fact, he credits Stan Rosenschein for drawing the idea to his attention). However, by devoting a chapter of his otherwise technical book to the speculative idea that admissible heuristic functions could be created automatically, he became an important early promoter of it.

The idea was first developed in the Milan Polytechnic Artificial Intelligence Project in the period 1973-1979. In a series of papers (*e.g.* [Sangiovanni-Vincentelli and Somalvico 1973; Guida and Somalvico 1979]) the Milan group developed the core elements of Pearl's vision. They proposed defining a heuristic function as the exact distance in a relaxed state space and proved that such heuristic func-

tions would be both admissible and *consistent*.[1] To make the computation of such heuristic functions efficient the Milan group envisaged a hierarchy of relaxed spaces, with search at one level being guided by a heuristic function defined by distances in the level above. The Milan group also foresaw the possibility of algorithms for searching through the space of possible simplified state spaces, although the first detailed articulation of this idea, albeit in a somewhat different context, was by Richard Korf [1980].

John Gaschnig [1979] picked up on the Milan work. He made the key observation that if a heuristic function is calculated by searching in a relaxed space, the total time required to solve the problem using the heuristic function could exceed the time required to solve the problem directly with breadth-first search (*i.e.* without using the heuristic function). This was formally proven shortly afterwards by Marco Valtorta [1981, 1984]. This observation led to a focus on the efficiency with which distances in the simplified space could be computed. The favorite approach to doing this (as exemplified in *Heuristics*) was to search for simplified spaces that could be decomposed.

3 The Vision Becomes a Reality

Directly inspired by Pearl's vision, Jack Mostow and Armand Prieditis set themselves the task of automating what had hitherto been paper-and-pencil speculation. The result was their ABSOLVER system [Mostow and Prieditis 1989; Prieditis 1993], which fully vindicated Pearl's enthusiasm for the idea of mechanically generating effective, admissible heuristics.

The input to ABSOLVER was a state space description in the standard STRIPS notation [Fikes and Nilsson 1971]. ABSOLVER had a library containing two types of transformations, each of which would take as input a STRIPS representation of a state space and produce as output one or more other STRIPS representations. The first type of transformation were *abstracting* transformations. Their purpose was to create a simplification (or "abstraction") of the given state space. One of these was *drop precondition*, exactly as Pearl had proposed. Their other abstracting transformations were a type of simplification that Pearl had not anticipated—they were *homomorphisms*, which are many-to-one mappings of states in the given space to states in the abstract space. Homomorphic state space abstractions for the purpose of defining heuristic functions were first described by Dennis Kibler in an unpublished report [1982], but their importance was not appreciated until ABSOLVER and the parallel work done by Keki Irani and Suk Yoo [1988].

An example of a homomorphic abstraction of the 15-puzzle is shown in Figure 2. Here tiles 9-15 and the blank are just as in the original puzzle (Figure 1) but tiles 1-8 have had their numbers erased so that they are not distinguishable from each other. Hence for any particular placement of tiles 9-15 and the blank, all the different ways

[1] Heuristic function $h(s)$ is consistent if, for any two states s_1 and s_2, $h(s_1) \leq dist(s_1, s_2) + h(s_2)$, where $dist(s_1, s_2)$ is the distance from s_1 to s_2.

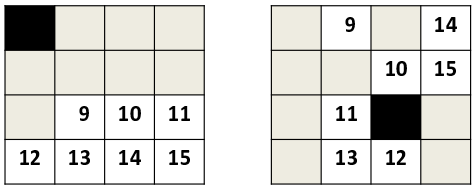


Figure 2. Homomorphic abstraction of the 15-puzzle

of permuting tiles 1-8 among the remaining positions produce 15-puzzle states that map to the same abstract state, even though they would all be distinct states in the original state space. For example, the abstract state in the left part of Figure 2 is the abstraction of the goal state in the original 15-puzzle (left part of Figure 1), but it is also the abstraction of all the non-goal states in the original puzzle in which tiles 9-15 and the blank are in their goal positions but some or all of tiles 1-8 are not. Using this abstraction, the distance from the 15-puzzle state in the right part of Figure 1 to the 15-puzzle goal state would be estimated by calculating the true distance, in the abstract space, from the abstract state in the right part of Figure 2 to the state in the left part of Figure 2.

In addition to abstracting transformations, ABSOLVER's library contained "optimizing" transformations, which would create an equivalent description of a given STRIPS representation in which search could be completed more quickly. This included the "factor" transformation that would, if possible, decompose the state space into independent subproblems, one of the methods Pearl had suggested.

ABSOLVER was applied to thirteen state spaces and found effective heuristic functions in six of them. Five of the functions it discovered were novel, including a simple, effective heuristic for Rubik's Cube that had been overlooked by experts:

> after extensive study, Korf was unable to find a single good heuristic evaluation function for Rubik's Cube [Korf 1985]. He concluded that "if there does exist a heuristic, its form is probably quite complex."
>
> ([Mostow and Prieditis 1989], page 701)

4 Dawn of the Modern Era

Despite ABSOLVER's success, it did not launch the modern era of abstraction-based heuristic functions. That would not happen until 1994, when Joe Culberson and Jonathan Schaeffer's work on *pattern databases* (PDBs) first appeared [Culberson and Schaeffer 1994]. They used homomorphic abstractions of the kind illustrated in Figure 2 and, as explained above, defined the heuristic function, $h(s)$, of state s to be the actual distance in the abstract space between the abstract state corresponding to s and the abstract goal. The key idea behind PDBs is to store the heuristic function as a lookup table so that its calculation during a search is extremely fast.

To do this, it is necessary to precompute all the distances to the goal state in the abstract space. This is typically done by a backwards breadth-first search starting at the abstract goal state. Each abstract state reached in this way is associated with a specific storage location in the PDB, and the state's distance from the abstract goal is stored in this location as the value in the PDB.

Precomputing abstract distances to create a lookup-table heuristic function was actually one of the optimizing transformations in ABSOLVER, but Culberson and Schaeffer had independently come up with the idea. Unlike the ABSOLVER work, they validated it by producing a two orders of magnitude reduction in the search effort (measured in nodes expanded) needed to solve instances of the 15-puzzle, as compared to the then state-of-the-art search algorithms using an enhanced Manhattan Distance heuristic. To achieve this they used two PDBs totaling almost one gigabyte of memory, a very large amount in 1994 when the experiments were performed [Culberson and Schaeffer 1994]. The paper's referees were sharply critical of the exorbitant memory usage, rejecting the paper three times before it finally was accepted [Culberson and Schaeffer 1996].

Such impressive results on the 15-puzzle could not go unnoticed. The fundamental importance of PDBs was established beyond doubt in 1997 when Richard Korf used PDBs to enable standard heuristic search techniques to find optimal solutions to instances of Rubik's Cube for the first time [Korf 1997].

Since then, PDBs have been used to build effective heuristic functions in numerous applications, including various combinatorial puzzles [Felner, Korf, and Hanan 2004; Felner, Korf, Meshulam, and Holte 2007; Korf and Felner 2002], multiple sequence alignment [McNaughton, Lu, Schaeffer, and Szafron 2002; Zhou and Hansen 2004], pathfinding [Anderson, Holte, and Schaeffer 2007], model checking [Edelkamp 2007], planning [Edelkamp 2001; Edelkamp 2002; Haslum, Botea, Helmert, Bonet, and Koenig 2007], and vertex cover [Felner, Korf, and Hanan 2004].

5 Current Status

The use of abstraction to create heuristic functions has profoundly advanced the fields of planning and heuristic search. But the current state of the art is not entirely as Pearl envisaged. Although he recognized that there were other types of state space abstraction, Pearl emphasized relaxation. In this detail, he was too narrowly focused. Researchers have largely abandoned relaxation in favor of homomorphic abstractions, of which many types have been developed and shown useful for defining heuristic functions, such as domain abstraction [Hernádvölgyi and Holte 2000], h-abstraction [Haslum and Geffner 2000], projection [Edelkamp 2001], constrained abstraction [Haslum, Bonet, and Geffner 2005], and synchronized products [Helmert, Haslum, and Hoffmann 2007].

Pearl argued for the automatic creation of effective heuristic functions by searching through a space of abstractions. There has been some research in this direction [Prieditis 1993; Hernádvölgyi 2003; Edelkamp 2007; Haslum, Botea, Helmert,

Bonet, and Koenig 2007; Helmert, Haslum, and Hoffmann 2007], but more is needed. However, important progress has been made on the subproblem of evaluating the effectiveness of a heuristic function, with the development of a generic, practical method for accurately predicting how many nodes IDA* (a standard heuristic search algorithm) will expand for any given heuristic function [Korf and Reid 1998; Korf, Reid, and Edelkamp 2001; Zahavi, Felner, Burch, and Holte 2008].

Finally, Pearl anticipated that efficiency in calculating the heuristic function would be achieved by finding abstract state spaces that were decomposable in some way. This has not come to pass, although there is now a general theory of when it is admissible to add the values returned by two or more different abstractions [Yang, Culberson, Holte, Zahavi, and Felner 2008]. Instead, the efficiency of the heuristic calculation has been achieved either by precomputing the heuristic function's values and storing them in a lookup table, as PDBs do, or by creating a hierarchy of abstractions and using distances at one level as a heuristic function to guide the calculation of distances at the level below [Holte, Perez, Zimmer, and MacDonald 1996; Holte, Grajkowski, and Tanner 2005], as anticipated by the Milan group.

6 Conclusion

Judea Pearl has received numerous accolades for his prodigious research and its impact. Amidst this impressive body of work are his often-overlooked contributions to the idea of the automatic discovery of heuristic functions. Even though *Heuristics* is over 25 years old (ancient by Computing Science standards), Pearl's ideas still resonate today.

Acknowledgments: The authors gratefully acknowledge the support they have received over the years for research in this area from Canada's Natural Sciences and Engineering Research Council (NSERC), Alberta's Informatics Circle of Research Excellence (iCORE), and the Israeli Science Foundation (ISF).

References

Anderson, K., R. Holte, and J. Schaeffer (2007). Partial pattern databases. In *Symposium on Abstraction, Reformulation and Approximation*, pp. 20–34. Springer-Verlag LNAI #4612.

Culberson, J. and J. Schaeffer (1994). Efficiently searching the 15-puzzle. Technical Report 94-08, Department of Computing Science, University of Alberta.

Culberson, J. and J. Schaeffer (1996). Searching with pattern databases. In G. McCalla (Ed.), *AI'96: Advances in Artificial Intelligence*, pp. 402–416. Springer-Verlag LNAI #1081.

Dijkstra, E. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik 1*, 269–271.

Doran, J. and D. Michie (1966). Experiments with the graph traverser program. In *Proceedings of the Royal Society A*, Volume 294, pp. 235–259.

Edelkamp, S. (2001). Planning with pattern databases. In *European Conference on Planning*, pp. 13–24.

Edelkamp, S. (2002). Symbolic pattern databases in heuristic search planning. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pp. 274–283.

Edelkamp, S. (2007). Automated creation of pattern database search heuristics. In *Model Checking and Artificial Intelligence*, pp. 35–50. Springer-Verlag LNAI #4428.

Felner, A., R. Korf, and S. Hanan (2004). Additive pattern database heuristics. *Journal of Artificial Intelligence Research (JAIR) 22*, 279–318.

Felner, A., R. Korf, R. Meshulam, and R. Holte (2007). Compressed pattern databases. *Journal of Artificial Intelligence Research (JAIR) 30*, 213–247.

Fikes, R. and N. Nilsson (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2*(3/4), 189–208.

Gaschnig, J. (1979). A problem similarity approach to devising heuristics: First results. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 301–307.

Guida, G. and M. Somalvico (1979). A method for computing heuristics in problem solving. *Information Sciences 19*, 251–259.

Hart, P., N. Nilsson, and B. Raphael (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SCC-4(2)*, 100–107.

Haslum, P., B. Bonet, and H. Geffner (2005). New admissible heuristics for domain-independent planning. In *National Conference on Artificial Intelligence (AAAI)*, pp. 1163–1168.

Haslum, P., A. Botea, M. Helmert, B. Bonet, and S. Koenig (2007). Domain-independent construction of pattern database heuristics for cost-optimal planning. In *National Conference on Artificial Intelligence (AAAI)*, pp. 1007–1012.

Haslum, P. and H. Geffner (2000). Admissible heuristics for optimal planning. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pp. 140–149.

Helmert, M., P. Haslum, and J. Hoffmann (2007). Flexible abstraction heuristics for optimal sequential planning. In *Automated Planning and Scheduling*, pp. 176–183.

Hernádvölgyi, I. (2003). Solving the sequential ordering problem with automatically generated lower bounds. In *Operations Research 2003 (Heidelberg, Germany)*, pp. 355–362.

Hernádvölgyi, I. and R. Holte (2000). Experiments with automatically created memory-based heuristics. In *Symposium on Abstraction, Reformulation and Approximation*, pp. 281–290. Springer-Verlag LNAI #1864.

Holte, R., J. Grajkowski, and B. Tanner (2005). Hierarchical heuristic search revisited. In *Symposium on Abstraction, Reformulation and Approximation*, pp. 121–133. Springer-Verlag LNAI #3607.

Holte, R., M. Perez, R. Zimmer, and A. MacDonald (1996). Hierarchical A*: Searching abstraction hierarchies efficiently. In *National Conference on Artificial Intelligence (AAAI)*, pp. 530–535.

Irani, K. and S. Yoo (1988). A methodology for solving problems: Problem modeling and heuristic generation. *IEEE Transactions on Pattern Analysis and Machine Intelligence 10*(5), 676–686.

Kibler, D. (1982). Natural generation of admissible heuristics. Technical Report TR-188, University of California at Irvine.

Korf, R. (1980). Towards a model of representation changes. *Artificial Intelligence 14*(1), 41–78.

Korf, R. (1985). *Learning to solve problems by searching for macro-operators*. Marshfield, MA, USA: Pitman Publishing, Inc.

Korf, R. (1997). Finding optimal solutions to Rubik's Cube using pattern databases. In *National Conference on Artificial Intelligence (AAAI)*, pp. 700–705.

Korf, R. and A. Felner (2002). Disjoint pattern database heuristics. *Artificial Intelligence 134*(1-2), 9–22.

Korf, R. and M. Reid (1998). Complexity analysis of admissible heuristic search. In *National Conference on Artificial Intelligence (AAAI)*, pp. 305–310.

Korf, R., M. Reid, and S. Edelkamp (2001). Time complexity of iterative-deepening-A*. *Artificial Intelligence 129*(1-2), 199–218.

McNaughton, M., P. Lu, J. Schaeffer, and D. Szafron (2002). Memory efficient A* heuristics for multiple sequence alignment. In *National Conference on Artificial Intelligence (AAAI)*, pp. 737–743.

Minsky, M. (1963). Steps toward artificial intelligence. In E. Feigenbaum and J. Feldman (Eds.), *Computers and Thought*, pp. 406–452. McGraw-Hill.

Mostow, J. and A. Prieditis (1989). Discovering admissible heuristics by abstracting and optimizing: A transformational approach. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 701–707.

Pearl, J. (1983). On the discovery and generation of certain heuristics. *AI Magazine 4*(1), 23–33.

Pearl, J. (1984). *Heuristics – Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Prieditis, A. (1993). Machine discovery of effective admissible heuristics. *Machine Learning 12*, 117–141.

Sacerdoti, E. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence 5*(2), 115–135.

Sangiovanni-Vincentelli, A. and M. Somalvico (1973). Theoretical aspects of state space approach to problem solving. In *International Congress on Cybernetics*, Namur, Belgium.

Valtorta, M. (1981). *A Result on the Computational Complexity of Heuristic Estimates for the A* Algorithm*. Ph.D. thesis, Department of Computer Science, Duke University.

Valtorta, M. (1984). A result on the computational complexity of heuristic estimates for the A* algorithm. *Information Sciences 55*, 47–59.

Yang, F., J. Culberson, R. Holte, U. Zahavi, and A. Felner (2008). A general theory of additive state space abstractions. *Journal of Artificial Intelligence Research (JAIR) 32*, 631–662.

Zahavi, U., A. Felner, N. Burch, and R. Holte (2008). Predicting the performance of IDA* with conditional probabilities. In *National Conference on Artificial Intelligence (AAAI)*, pp. 381–386.

Zhou, R. and E. Hansen (2004). Space-efficient memory-based heuristics. In *National Conference on Artificial Intelligence (AAAI)*, pp. 677–682.

4

Space Complexity of Combinatorial Search

Richard E. Korf

1 Introduction: The Problem

It is well-known that most complete search algorithms take exponential time to run on most combinatorial problems. The reason for this is that many combinatorial problems are NP-hard, and most complete search algorithms guarantee an optimal solution, so unless P=NP, the time complexity of these algorithms must be exponential in the problem size.

What is not so often appreciated is that the limiting resource of many search algorithms is not time, but the amount of memory they require. For example, a simple brute-force breadth-first search (BFS) of an implicit problem space stores every node it generates in memory. If we assume we can generate ten million nodes per second, can store a node in four bytes of memory, and have four gigabytes of memory, we will exhaust our memory in a hundred seconds, or less than two minutes. If our problem is too large to be solved in this amount of time, the memory limitation becomes the bottleneck.

This problem has existed since the first computers were built. While memory capacities have increased by many orders of magnitude over that time, processors have gotten faster at roughly the same pace, and the problem persists. We describe here the major approaches to this problem over the past 25 years. While most of the algorithms discussed have both brute-force and heuristic versions, we focus primarily on the brute-force algorithms, since they are simpler, and the issues are largely the same in both cases.

2 Depth-First Search

One solution to this problem in some settings is depth-first search (DFS), which requires memory that is only linear in the maximum search depth. The reason is that at any point in time, it saves only the path from the start node to the current node being expanded, either on an explicit node stack, or on the call stack of a recursive implementation. As a result, the memory requirement of DFS is almost never a limitation. For finite tree-structured problem space graphs, where all solutions are equally desirable, DFS solves the memory problem. For example, chronological backtracking is a DFS for constraint satisfaction problems, and does not suffer any memory limitations in practice.

With an infinite search tree, or when we want a shortest solution path, however, DFS has significant drawbacks. In an infinite search tree, which can result from a

depth-first search of a finite graph with multiple paths to the same state, DFS is not complete, but can traverse a single path until it exhausts memory. For example, the problem space graphs of the well-known sliding-tile puzzles are finite, but a depth-first search of these spaces explores a tree-expansion of the graph, which is infinite. Even with a finite search tree, the first solution found by DFS will not be a shortest solution in general.

3 Iterative Deepening

3.1 Depth-First Iterative Deepening

One solution to these limitations of DFS is depth-first iterative-deepening (DFID) [Korf 1985a]. DFID performs a series of depth-first searches, each to a successively greater depth. DFID simulates BFS, but using memory that is only linear in the maximum search depth. It is guaranteed to find a solution if one exists, even on an infinite tree, and the first solution it finds will be a shortest one.

DFID is essentially the same as iterative-deepening searches used in two-player game programs [Slate and Atkin 1977], but is used to solve a completely different problem. In a two-player game, iterative deepening is used to determine the search depth, because moves must be made within a given time period, and it is difficult to predict how long it will take to search to a given depth. In contrast, DFID is applied to single-agent problems where a shortest solution is required, in order to avoid the memory limitation of BFS.

DFID first appeared in a Columbia University technical report [Korf 1984]. It was independently published in two different papers in IJCAI-85 [Korf 1985b; Stickel and Tyson 1985], and called "consecutively bounded depth-first search" in the latter.

3.2 Iterative-Deepening-A*

While discussing DFID with Judea Pearl on a trip to UCLA in 1984, he suggested its extension to heuristic search that became Iterative-Deepening-A* (IDA*) [Korf 1985a]. IDA* overcomes the memory limitation of the A* algorithm [Hart, Nilsson, and Raphael 1968] for heuristic searches the same way that DFID overcomes the memory limitation of BFS for brute-force searches. In particular, it uses the A* cost function of $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the current path from the root to node n, and $h(n)$ is a heuristic estimate of the lowest cost of any path from node n to a goal node. IDA* performs a series of depth-first search iterations, where each branch of the search is terminated when the cost of the last node on that branch exceeds a cost threshold for that iteration. The cost threshold of the first iteration is set to the heuristic estimate of the start state, and the cost threshold of each successive iteration is set to the minimum cost of all nodes generated but not expanded on the previous iteration. Like A*, IDA* guarantees an optimal solution if the heuristic function is *admissible*, or never overestimates actual cost. IDA* was the first algorithm to find optimal solutions to the Fifteen Puzzle, the famous four-by-four sliding-tile puzzle [Korf 1985a]. It was also the first algorithm to find

optimal solutions to Rubik's Cube [Korf 1997].

I invited Judea to be a co-author on both the IJCAI paper [Korf 1985b], and a subsequent AI journal paper [Korf 1985a] that described both DFID and IDA*. At the time, I was a young assistant professor, and he declined the co-authorship with typical generosity, saying that he didn't need another paper, and that the paper would be much more important to me at that stage of my career. In retrospect, I regret that I didn't insist on joint authorship.

4 Other Limited-Memory Search Algorithms

A number of researchers noted that linear-space search algorithms such as DFID and IDA* use very little space in practice, and explored whether better performance could be achieved by using all the memory available on a machine.

Perhaps the simplest of these algorithms is MREC [Sen and Bagchi 1989]. MREC is a hybrid of A* and IDA*. It runs A* until memory is almost full, and then runs successive iterations of IDA* from each node generated but not expanded by A*.

Perhaps the most elegant of these algorithms is MA* [Chakrabarti, Ghose, Acharya, and de Sarkar 1989]. MA* also runs A* until memory is almost full. Then, in order to get enough memory to expand the best node, it finds a group of sibling leaf nodes with the worst cost values, and deletes them, leaving behind only their parent node, with a stored cost equal to the minimum of its children's values. The algorithm alternates between expanding the best nodes, and contracting the worst nodes, until a solution is chosen for expansion.

Unfortunately, the overhead of this algorithm makes it impractical compared to IDA*. There have been at least two attempts to make this basic algorithm more efficient, namely SMA* for simplified MA* [Russell 1992] and ITS for Iterative Threshold Search [Ghosh, Mahanti, and Nau 1994], but none of these algorithms significantly outperform IDA*.

5 Recursive Best-First Search

Best-first search is a general class of search algorithms that maintains both a Closed list of expanded nodes, and an Open list of nodes that have been generated but not yet expanded. Initially, the Open list contains just the start node, and the Closed list is empty. Each node n on Open has an associated cost $f(n)$. At each step of the algorithm, an Open node of lowest cost is chosen for expansion, moved to the Closed list, and its children are placed on Open along with their associated costs. The algorithm continues until a goal node is chosen for expansion. Different best-first search algorithms differ only in their cost functions. For example, if the cost of a node is simply its depth, then best-first search becomes breadth-first search. Alternatively, if the cost of a node n is $f(n) = g(n) + h(n)$, then best-first search becomes the A* algorithm.

A cost function is *monotonic* if the cost of a child node is always greater than or equal to the cost of its parent. The cost function $g(n)$ is monotonic if all edges

have non-negative cost. The A* cost function $f(n) = g(n) + h(n)$ is monotonic if the heuristic function $h(n)$ is *consistent*, meaning that $h(n) \leq k(n, n') + h(n')$, where n' is a child of node n, and $k(n, n')$ is the cost of the edge from n to n'. Many heuristic functions are both admissible and consistent. If the cost function is monotonic, then the order in which nodes are first expanded by IDA* is the same as for a best-first search with the same cost function.

Not all useful cost functions are monotonic, however. For example, Weighted A* (WA*) is a best-first search with the cost function $f(n) = g(n) + w * h(n)$. If w is greater than one, then WA* usually finds solutions much faster than A*, but at a small cost in solution quality. With w greater than one, however, $f(n) = g(n)+w*h(n)$ is not monotonic, even with a consistent $h(n)$. With a non-monotonic cost-function, IDA* does not expand nodes in best-first order. In particular, in parts of the search tree where the cost of nodes is lower than the cost threshold for the current iteration, IDA* behaves as a brute-force search, expanding nodes in the order in which they are generated.

Can any linear-space search algorithm simulate a best-first search with a non-monotonic cost function? Surprisingly, the answer is yes. *Recursive best-first search* [Korf 1993] (RBFS) maintains a path from the start node to the last node generated, along with the siblings of nodes on that path. Stored with each node is a cost value. If the node has never been expanded before, its stored cost is its original cost. If it has been expanded before, its stored cost is the minimum cost of all its descendents that have been generated but not yet expanded, which are not stored in memory. The sibling node off the current path of lowest cost is the ancestor of the next leaf node that would be expanded by a best-first search. By propagating these values up the tree, and inheriting these values down the tree as a previously explored path is regenerated, RBFS always finds the next leaf node expanded by a best-first search. Thus, it simulates a best-first search even with a non-monotonic cost function. Furthermore, with a monotonic cost function it can outperform IDA* if there are many different unique cost values in the search tree. For details on RBFS, see [Korf 1993].

6 Drawback of Linear-Space Search Algorithms

The advantage of linear-space searches, such as DFS, DFID, IDA* and RBFS, is that they use very little memory, and hence can run for weeks or months on large problems. They all share a significant liability relative to best-first searches such as BFS, A* or WA*, however. In particular, on search graphs with multiple paths to the same node, or cycles in the graph, linear-space algorithms can generate exponentially more nodes than best-first search.

For example, consider a rectangular grid problem-space graph. From any node in such a graph, moving North and then East generates the same state as moving East and then North. These are referred to as duplicate nodes, since they represent the same state arrived at via two different paths. When a best-first search is run on

such a graph, as each node is generated it is checked to see if the same state already appears on the Open or Closed lists. If so, only the node reached by a shortest path is stored, and the duplicate node is eliminated. Thus, by detecting and rejecting such duplicate nodes, a breadth-first search to a radius of r on a grid graph would expand $O(r^2)$ nodes.

A linear-space algorithm doesn't store most of the nodes it generates however, and hence cannot detect most duplicate nodes. In a grid graph, each node has four neighbors. A linear-space search will not normally regenerate its immediate parent as one of its children, reducing the number of children to three for all but the start node. Thus, a depth-first search of a grid graph to a radius r will generate $O(3^r)$ nodes, compared to $O(r^2)$ nodes for a best-first search. This is an enormous overhead on graphs with many paths to the same state, rendering linear-space algorithms completely impractical in such problem spaces.

7 Frontier Search

Fortunately, there is another technique that can significantly reduce the memory required by a search algorithm on problem spaces with many duplicate nodes. The basic idea is to save only the Open list and not the Closed list. This algorithm schema is called *frontier search*, since the Open list represents the frontier of nodes that have been generated but not yet expanded [Korf, Zhang, Thayer, and Hohwald 2005]. When a node is expanded in frontier search, it is simply deleted rather than being moved to a Closed list.

The advantage of this technique is that the Open list can be much smaller than the Closed list. In the grid graph, for example, the Closed list grows as $O(r^2)$, whereas the Open list grows only as $O(r)$, where r is the radius of the search.

For ease of explanation, we'll assume a problem space with reversible operators, but the method also applies to some directed problem-space graphs as well. There are two reasons to save the Closed list. One is to detect duplicate nodes, and the other is to return the solution path. We first consider duplicate detection.

7.1 Detecting Duplicate Nodes

Imagine the search frontier as a continuous boundary of Open nodes containing a region of Closed nodes. To minimize the memory needed, we need to prevent Closed nodes from being regenerated. There are two ways that this might happen. One is by a child node regenerating its parent node. This is prevented in frontier search by storing with each Open node a *used-operator bit* for each operator that could generate that node. This bit is set to one whenever the corresponding operator is used to generate the Open node. When an Open node is expanded, the inverses of those operators whose used bits are set to one are not applied, thus preventing a node from regenerating its parent.

The other way a Closed node could be regenerated is by the frontier looping back on itself, like a wave breaking through the surface of the water below. If the frontier

is unbroken, when this happens the Open node being expanded would first have to generate another Open node on the frontier of the search before generating a Closed node on the interior. When this happens, the duplicate Open node is detected, and the the union of the used operator bits set in each of the two copies is stored with the single copy retained. In other words, one part of the frontier cannot invade another part of the interior without passing through another part of the frontier first, where the intrusion is detected. By storing and managing such used-operator bits, frontier search detects all duplicate node generations and prevents a node from being expanded more than once.

An alternative to used-operator bits is to save several levels of the search at a time [Zhou and Hansen 2003]. In particular, Closed nodes are stored until all their children are expanded, and then deleted.

7.2 Reconstructing the Solution Path

The other reason to store the Closed list is to reconstruct the solution path at the end of a search. In a best-first search, this is done by storing with each node a pointer to its parent node. Once a goal state is reached, these pointers are followed back from the goal to the initial state, generating the solution path in reverse order. This can't be done with frontier search directly, since the Closed list is not saved.

Frontier search can reconstruct the solution path using *divide-and-conquer bidirectional frontier search*. We search simultaneously both forward from the initial state and backward from the goal state. When the two search frontiers meet and a "middle" state on a optimal path has been found, we then use the same algorithm recursively to search from the initial state to the middle state, and from the middle node to the goal node.

The solution path can also be constructed using unidirectional search, as long as one can identify nodes that are approximately half way along an optimal path. For example, the problem of two-way sequence alignment in computational biology can be mapped to finding a shortest path in a two-dimensional grid [Needleman and Wunsch 1970]. In such a path, a state on the midline of the grid will be about half way along an optimal solution. In such a problem, we search forward from the initial state to the goal state. For every node on the Open list past the midpoint of the grid, we store a pointer to its ancestor on the midpoint. Once we reach the goal state, its ancestor on the midline is approximately in the middle of the optimal solution path. We then recursively apply the same algorithm to find a path from the initial state to the middle state, and from the middle state to a goal state.

Tree-structured search spaces and densely connected graphs such as grids represent two ends of the connectivity spectrum. On a tree, linear-space search algorithms perform very well, since they generate no duplicate nodes. Frontier search doesn't save much memory on a tree, however, since the number of leaf nodes dominates the number of interior nodes. Conversely, on a grid graph, linear-space search performs very poorly because undetected duplicate nodes dwarf the number

of unique nodes, while frontier search performs very well, reducing the memory required from quadratic to linear space.

8 Disk-Based Search

Even on problems where frontier search is effective, memory is still the resource that limits its applicability. An additional approach to this memory limitation is to use magnetic disk to store nodes rather than semiconductor memory. While semiconductor memory has gotten much larger and cheaper over time, it still costs about $30 per gigabyte. In contrast, magnetic disk storage costs about $100 per terabyte, which is 300 times cheaper. The problem with simply replacing semiconductor memory with magnetic disks, however, is that random access of a byte on disk can take up to ten milliseconds, which is five orders of magnitude slower than for memory. Thus, disk access must be sequential for efficiency.

Consider a simple breadth-first search (BFS), which is usually implemented with a first-in first-out queue. Nodes are read from the head of the queue, expanded, and their children are written to the tail of the queue. Such a queue can efficiently be stored on disk, since all accesses are sequential.

In order to detect duplicate nodes efficiently, however, the nodes are also stored in a hash table. Nodes are looked up in the hash table as they are generated, and duplicate nodes are discarded. Such a hash table cannot be directly implemented on magnetic disk, however, due to the long latency of random access.

A solution to this problem is called *delayed duplicate detection* [Korf 2008] or DDD for short. The BFS queue is stored on disk, but nodes are not checked for duplicates as they are generated. Rather, duplicate nodes are appended to the queue, and are only eliminated periodically, such as at the end of each depth iteration. There are several ways to eliminate duplicate nodes from a large file stored on disk.

The simplest way is to sort the nodes based on their state representation. This will bring duplicate nodes to adjacent positions. Then, a simple linear scan of the file can be used to detect and merge duplicate nodes. The drawback of this approach is that the sorting takes $O(n \log n)$ time, where n is the number of nodes. With a terabyte of storage, and four bytes per state, n can be as large as 250 billion, and hence $\log n$ as large as 38.

An alternative is to use hash-based DDD. This scheme relies on two orthogonal hash functions defined on the state representation. In the first phase, the input file is read, and nodes are output to separate files based on the value of the first hash function. Thus, any sets of duplicate node will be confined to the same file. In the second phase, the nodes in each individual file are hashed into memory using the second hash function, and duplicates are detected and merged in memory. The advantage of this approach is that the time complexity is only linear in the number of nodes, rather than $O(n \log n)$ time for sorting-based DDD.

The overall DDD algorithm proceeds in alternating phases of node expansion followed by merging duplicate nodes. Combined with frontier search, DDD has

been used to perform complete breadth-first searches of sliding-tile puzzles as large as the Fifteen Puzzle, with over 10^{13} nodes [Korf and Schultze 2005]. It has also been used for large heuristic searches of the four-peg Towers of Hanoi problem with up to 31 disks, generating over 2.5×10^{13} nodes [Korf and Felner 2007]. These searches take weeks to run, and time is the limiting resource, not storage capacity.

An alternative to DDD for disk-based search is called *structured duplicate detection* (SDD) [Zhou and Hansen 2004]. In this approach, the problem space is partitioned into subsets, so that when expanding nodes in one subset, the children only belong to a small number of other subsets, referred to as its *duplicate detection scope*. The subset of nodes currently being expanded is kept in memory, along with the subsets in its duplicate detection scope, detecting any duplicates generated immediately, while storing other subsets on disk. As different subsets of nodes are expanded, currently resident duplicate detection scopes are swapped out to disk to make room for the duplicate detection scopes of the new nodes being expanded.

9 Summary and Conclusions

We have presented a number of different algorithms, designed over the past 25 years, to deal with the space complexity of brute-force and heuristic search. They fall into two general categories. The linear-space search algorithms, including depth-first search, depth-first iterative-deepening, iterative-deepening-A*, and recursive best-first search, use very little memory but cannot detect most duplicate nodes. They perform very well on trees, but poorly on highly-connected graphs, such as a grid. The best-first search algorithms, including breadth-first search, A*, weighted A*, frontier search, and disk-based algorithms, detect all duplicate nodes, and hence perform well on highly-connected graphs. The best-first algorithms are limited by the amount of memory available, except for the disk-based techniques, which are limited by time in practice.

Acknowledgments: This research was supported continuously by the National Science Foundation, most recently under grant No. IIS-0713178.

References

Chakrabarti, P., S. Ghose, A. Acharya, and S. de Sarkar (1989, December). Heuristic search in restricted memory. *Artificial Intelligence 41*(2), 197–221.

Ghosh, R., A. Mahanti, and D. Nau (1994, August). An efficient limited-memory heuristic tree search algorithm. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, Seattle, WA, pp. 1353–1358.

Hart, P., N. Nilsson, and B. Raphael (1968, July). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC-4*(2), 100–107.

Korf, R. (1984). The complexity of brute-force search. technical report, Computer Science Department, Columbia University, New York, NY.

Korf, R. (1985a). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence 27*(1), 97–109.

Korf, R. (1985b, August). Iterative-deepening-a*: An optimal admissible tree search. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, Los Angeles, CA, pp. 1034–1036.

Korf, R. (1993, July). Linear-space best-first search. *Artificial Intelligence 62*(1), 41–78.

Korf, R. (1997, July). Finding optimal solutions to rubik's cube using pattern databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, Providence, RI, pp. 700–705.

Korf, R. (2008, December). Linear-time disk-based implicit graph search. *Journal of the Association for Computing Machinery 55*(6), 26:1 to 26:40.

Korf, R. and A. Felner (2007, January). Recent progress in heuristic search: A case study of the four-peg towers of hanoi problem. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, Hyderabad, India, pp. 2334–2329.

Korf, R. and P. Schultze (2005, July). Large-scale, parallel breadth-first search. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, Pittsburgh, PA, pp. 1380–1385.

Korf, R., W. Zhang, I. Thayer, and H. Hohwald (2005, September). Frontier search. *Journal of the Association for Computing Machinery 52*(5), 715–748.

Needleman, S. and C. Wunsch (1970). A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology 48*, 443–453.

Russell, S. (1992, August). Efficient memory-bounded search methods. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI-92)*, Vienna, Austria.

Sen, A. and A. Bagchi (1989, August). Fast recursive formulations for best-first search that allow controlled use of memory. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, Detroit, MI, pp. 297–302.

Slate, D. and L. Atkin (1977). Chess 4.5 - the northwestern university chess program. In P. Frey (Ed.), *Chess Skill in Man and Machine*, pp. 82–118. New York, NY: Springer-Verlag.

Stickel, M. and W. Tyson (1985, August). An analysis of consecutively bounded depth-first search with applications in automated deduction. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, Los Angeles, CA, pp. 1073–1075.

Zhou, R. and E. Hansen (2003, August). Sparse-memory graph search. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, Acapulco, Mexico, pp. 1259–1266.

Zhou, R. and E. Hansen (2004, July). Structured duplicate detection in external-memory graph search. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, San Jose, CA, pp. 683–688.

5

Paranoia versus Overconfidence in Imperfect-Information Games

Austin Parker, Dana Nau, and V.S. Subrahmanian

Only the paranoid survive.
–Andrew Grove, Intel CEO

Play with supreme confidence, or else you'll lose.
–Joe Paterno, college football coach

1 Introduction

In minimax game-tree search, the *min* part of the minimax backup rule derives from what we will call the *paranoid assumption*: the assumption that the opponent will always choose a move that minimizes our payoff and maximizes his/her payoff (or our estimate of the payoff, if we cut off the search before reaching the end of the game). A potential criticism of this assumption is that the opponent may not have the ability to decide accurately what move this is. But in several decades of experience with game-tree search in chess, checkers, and other zero-sum perfect-information games, the paranoid assumption has worked so well that such criticisms are generally ignored.

In game-tree search algorithms for imperfect-information games, the backup rules are more complicated. Many of them (see Section 6) involve computing a weighted average over the opponent's possible moves (or a Monte Carlo sample of them), where each move's weight is an estimate of the probability that this is the opponent's best possible move. Although such backup rules do not take a *min* at the opponent's move, they still tacitly encode the paranoid assumption, by assuming that the opponent will choose optimally from the set of moves he/she is actually capable of making.

Intuitively, one might expect the paranoid assumption to be less reliable in imperfect-information games than in perfect-information games; for without perfect information, it may be more difficult for the opponent to judge which move is best. The purpose of this paper is to examine whether it is better to err on the side of paranoia or on the side of overconfidence. Our contributions are as follows:

1. **Expected utility.** We provide a recursive formula for the expected utility of a move in an imperfect-information game, that explicitly includes the opponent's strategy σ. We prove the formula's correctness.

2. **Information-set search.** We describe a game-tree search algorithm called *information-set search* that implements the above formula. We show analytically that with an accurate opponent model, information-set search produces optimal results.

3. **Approximation algorithm.** Information-set search is, of course, intractable for any game of interest as the decision problem in an imperfect-information game is complete in double exponential time [Reif 1984]. To address this intractability problem, we provide a modified version of information-set search that computes an approximation of a move's expected utility by combining Monte Carlo sampling of the belief state with a limited-depth search and a static evaluation function.

4. **Paranoia and overconfidence.** We present two special cases of the expected-utility formula (and hence of the algorithm) that derive from two different opponent models: the *paranoid* model, which assumes the opponent will always make his/her best possible move, and the *overconfident* model, which assumes the opponent will make moves at random.

5. **Experimental results.** We provide experimental evaluations of information-set search in several different imperfect-information games. These include imperfect-information versions of P-games [Pearl 1981; Nau 1982a; Pearl 1984], N-games [Nau 1982a], and kalah [Murray 1952]; and an imperfect-information version of chess called kriegspiel [Li 1994; Li 1995]. Our main experimental results are:

    - Information-set search outperformed HS, the best of our algorithms for kriegspiel in [Parker, Nau, and Subrahmanian 2005].
    - In all of the games, the overconfident opponent model outperformed the paranoid model. The difference in performance became more marked when we decreased the amount of information available to each player.

This work was influenced by Judea Pearl's invention of P-games [Pearl 1981; Pearl 1984], and his suggestion of investigating backup rules other than minimax [Pearl 1984]. We also are grateful for his encouragement of the second author's early work on game-tree search (e.g., [Nau 1982a; Nau 1983]).

2 Basics

Our definitions and notation are based on [Osborne and Rubinstein 1994]. We consider games having the following characteristics: two players, finitely many moves and states, determinism, turn taking, zero-sum utilities, imperfect information expressed via *information sets* (explained in Section 2.1), and *perfect recall* (explained in Section 2.3). We will let G be any such game, and a_1 and a_2 be the two players.

Our techniques are generalizable to stochastic multi-player non-zero-sum games,[1] but that is left for future work.

At each state s, let $a(s)$ be the player to move at s, with $a(s) = \emptyset$ if the game is over in s. Let $M(s)$ be the set of available moves at s, and $m(s)$ be the state produced by making move m in state s. A *history* is a sequence of moves $h = \langle m_1, m_2, \ldots, m_j \rangle$. We let $s(h)$ be the state produced by history h, and when clear from context, will abuse notation and use h to represent $s(h)$ (e.g., $m(h) = m(s(h))$). Histories in which the game has ended are called *terminal*. We let H be the set of all possible histories for game G.

2.1 Information Sets

Intuitively, an information set is a set of histories that are indistinguishable to a player a_i, in the sense that each history h provides a_i with the same sequence of observations. For example, suppose a_1 knows the entire sequence of moves that have been played so far, except for a_2's last move. If there are two possibilities for a_2's last move, then a_1's information set includes two histories, one for each of the two moves.

In formalizing the above notion, we will not bother to give a full formal definition of an "observation." The only properties we need for an observation are the following:[2]

- We assume that each player a_i's sequence of observations is a function $O_i(h)$ of the current history h. The rationale is that if a_1 and a_2 play some game a second time, and if they both make the same moves that they made the first time, then they should be able to observe the same things that they observed the first time.

- We assume that when two histories h, h' produce the same sequence of observations, they also produce the same set of available moves, i.e., if $O_i(h) = O_i(h')$, then $M(s(h)) = M(s(h'))$. The rationale for this is that if the current history is h, a_i's observations won't tell a_i whether the history is h or h', so a_i may attempt to make a move m that is applicable in $s(h')$ but not in $s(h)$. If a_i does so, then m will produce *some* kind of outcome, even if the outcome is just an announcement that a_i must try a different move. Consequently, we can easily make m applicable in $s(h)$, by defining a new state $m(s(h))$ in which this outcome occurs.

[1] Nondeterministic initial states, outcomes, and observations can be modeled by introducing an additional player a_0 who makes a nondeterministic move at the start of the game and after each of the other players' moves. To avoid affecting the other players' payoffs, a_0's payoff in terminal states is always 0.

[2] Some game-theory textbooks define information sets without even using the notion of an "observation." They simply let a player's information sets be the equivalence classes of a partition over the set of possible histories.

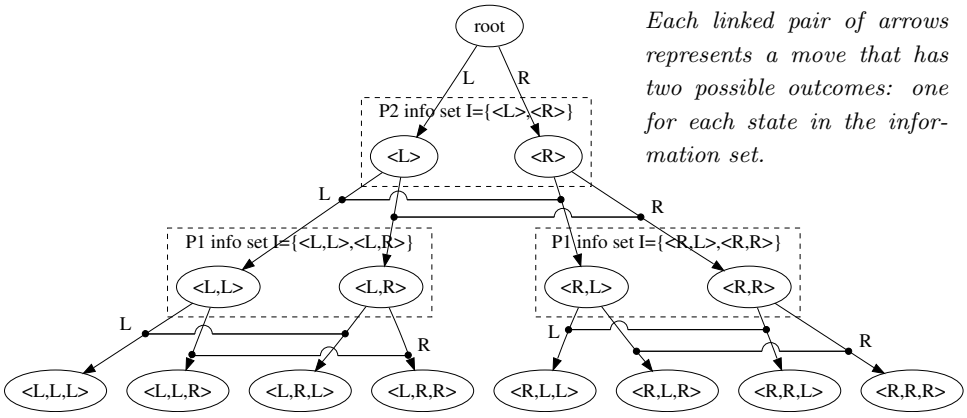


Figure 1. A game tree for a two-player imperfect-information game between two players P1 and P2 who move in alternation. The players may move either left (L) or right (R), and their moves are hidden from each other (e.g., after P1's first move, P2 knows that P1 has moved, but not whether the move was L or R). Each node is labeled with its associated history (e.g., $\langle L \rangle$ and $\langle R \rangle$ for the two children of the root node). The information set of the player to move is indicated by a dotted box (e.g., after P1's first move, P2's information set is $\{\langle L \rangle, \langle R \rangle\}$).

- We assume that terminal histories with distinct utilities always provide distinct observations, i.e., for terminal histories $h, h' \in T$, if $U_i(h) \neq U_i(h')$ then $O_i(h) \neq O_i(h')$.

We define a_i's information set for h to be the set of all histories that give a_i the same observations that h gives, i.e., $[h]_i = \{h' \in H : O_i(h') = O_i(h)\}$. The set of all possible information sets for a_i is $\mathcal{I}_i = \{[h]_i : h \in H\}$. It is easy to show that $\mathcal{I}_i$ is a partition of H.

Figure 1 shows an example game tree illustrating the correspondence between information sets and histories. In that game, player a_1 makes the first move, which is hidden to player a_2. Thus player a_2 knows that the history is either $\langle L \rangle$ or $\langle R \rangle$, which is denoted by putting a dotted box around the nodes for those histories.

2.2 Strategies

In a perfect-information game, a player a_i's *strategy* is a function $\sigma_i(m|s)$ that returns the probability p that a_i will make move m in state s. For imperfect-information games, where a_i will not always know the exact state he/she is in, σ_i is a function of an information set rather than a state; hence $\sigma_i(m|I)$ is the probability that a_i will make move m when their information set is I. We let $M(I)$ be the set of moves available in information set I.

If σ_i is a mixed strategy, then for every information set $I \in \mathcal{I}_i$ where it is a_i's move, there may be more than one move $m \in M(I)$ for which $\sigma_i(m|I) > 0$. But if σ_i is a pure strategy, then there will be a unique move $m_I \in M(I)$ such that

$\sigma_i(m|I) = 0\ \forall m \neq m_I$ and $\sigma_i(m_I|I) = 1$; and in this case we will use the notation $\sigma_i(I)$ to refer to m_I.

If $h = \langle m_1, m_2, \ldots, m_n \rangle$ is a history, then its probability $P(h)$ can be calculated from the players' strategies. Suppose a_1's and a_2's strategies are σ_1 and σ_2. In the special case where a_1 has the first move and the players move in strict alternation,

$$P(h|\sigma_1, \sigma_2) = \sigma_1(m_1|h_0)\sigma_2(m_2|h_1)\ldots\sigma_1(m_j|h_{j-1}), \sigma_2(m_{j+1}|h_j), \ldots, \quad (1)$$

where $h_j = \langle m_1, \ldots, m_j \rangle$ ($h_0 = \langle\rangle$). More generally,

$$P(h|\sigma_1, \sigma_2) = \prod_{j=0}^{n-1} \sigma_{a(h_j)}(m_{j+1}|h_j). \quad (2)$$

Given σ_1, σ_2, and any information set I, the conditional probability of any $h \in I$ is the normalized probability

$$P(h|I, \sigma_1, \sigma_2) = \frac{P(h|\sigma_1, \sigma_2)}{\sum_{h' \in I} P(h'|\sigma_1, \sigma_2)}. \quad (3)$$

2.3 Perfect Recall

Perfect recall means that every player always remembers all the moves they've made – we can't have two histories in player a_i's information set which disagree on what player a_i did at some point in the past. One can get a more detailed explanation of perfect and imperfect recall in perfect information games in [Osborne and Rubinstein 1994].

In a game of perfect recall, it is easy to show that if $I \in \mathcal{I}_1$, then all histories in I have the same sequence of moves for a_1, whence the probability of h given I is conditionally independent of σ_1. If $h = \langle m_1, m_2, \ldots, m_n \rangle$, then

$$P(h|I, \sigma_1, \sigma_2) = P(h|I, \sigma_2) = \frac{\prod_{a(h_j)=a_2} \sigma_2(m_{j+1}|h_j)}{\sum_{h' \in I} \prod_{a(h'_j)=a_2} \sigma_2(m_{j+1}|h'_j)}. \quad (4)$$

An analogous result, with the subscripts 1 and 2 interchanged, holds when $I \in \mathcal{I}_2$.

2.4 Utility and Expected Utility

If a history h takes us to the game's end, then h is *terminal*, and we let $U(h)$ be the *utility* of h for player a_1. Since the game is zero-sum, it follows that a_2's utility is $-U(h)$.

If a_1 and a_2 have strategies σ_1 and σ_2, then the expected utility for a_i is

$$EU(\sigma_1, \sigma_2) = \sum_{h \in T} P(h|\sigma_1, \sigma_2)U(h), \quad (5)$$

where T is the set of all terminal histories, and $P(h|\sigma_1, \sigma_2)$ is as in Eq. (2). Since the game is zero-sum, it follows that a_2's expected utility is $-EU(\sigma_1, \sigma_2)$.

For the expected utility of an individual history h, there are two cases:

Case 1: History h is terminal. Then h's expected utility is just its actual utility, i.e.,

$$EU(h|\sigma_1, \sigma_2) = EU(h) = U(h). \tag{6}$$

Case 2: History h ends at a state where it is a_i's move. Then h's expected utility is a weighted sum of the expected utilities for each of a_i's possible moves, weighted by the probabilities of a_i making those moves:

$$\begin{aligned} EU(h|\sigma_1, \sigma_2) &= \sum_{m \in M(h)} \sigma_i(m|h) \cdot EU(h \circ m|\sigma_1, \sigma_2) \\ &= \sum_{m \in M(h)} \sigma_i(m|[h]_i) \cdot EU(h \circ m|\sigma_1, \sigma_2), \end{aligned} \tag{7}$$

where $\circ$ denotes concatenation.

The following lemma shows that the recursive formulation in Eqs. (6–7) matches the notion of expected utility given in Eq. 5.

LEMMA 1. *For any strategies σ_1 and σ_2, $EU(\langle\rangle|\sigma_1, \sigma_2)$ (the expected utility of the empty initial history as computed via the recursive Equations 6 and 7) equals $EU(\sigma_1, \sigma_2)$.*

Sketch of proof. This is shown by showing, by induction on the length of h, the more general statement that

$$EU(h|\sigma_1, \sigma_2) = \sum_{h' \in T, h' = h \circ m_k, \circ \cdots \circ, m_n} P(h'|\sigma_1, \sigma_2) U(h') / P(h|\sigma_1, \sigma_2), \tag{8}$$

where k is one greater than the size of h and n is the size of each h' as appropriate. The base case occurs when h is terminal, and the inductive case assumes Eq. 8 holds for histories of length $m + 1$ to show algebraically that Eq. 8 holds for histories of length m. □

The expected utility of an information set $I \in H$ is the weighted sum of the expected utilities of its histories:

$$EU(I|\sigma_1, \sigma_2) = \sum_{h \in I} P(h|I, \sigma_1, \sigma_2) EU(h|\sigma_1, \sigma_2). \tag{9}$$

COROLLARY 2. *For any strategies σ_1 and σ_2, and player a_i, $EU([\langle\rangle]_i|\sigma_1, \sigma_2)$ (the expected utility of the initial information set for player a_i) equals $EU(\sigma_1, \sigma_2)$.*

3 Finding a Strategy

We now develop the theory for a game-tree search technique that exploits an opponent model.

3.1 Optimal Strategy

Suppose a_1's and a_2's strategies are σ_1 and σ_2, and let I be any information set for a_1. Let $M^*(I|\sigma_1, \sigma_2)$ be the set of all moves in $M(I)$ that maximize a_1's expected utility at I, i.e.,

$$M^*(I|\sigma_1,\sigma_2) = \operatorname*{argmax}_{m\in M(I)} EU(I\circ m|\sigma_1,\sigma_2)$$
$$= \left\{ m^* \in M(I) \;\middle|\; \begin{array}{r} \forall m \in M(I),\ \sum_{h\in I} P(h|I,\sigma_1,\sigma_2)EU(h\circ m^*|\sigma_1,\sigma_2) \\ \geq\ \sum_{h\in I} P(h|I,\sigma_1,\sigma_2)EU(h\circ m|\sigma_1,\sigma_2) \end{array} \right\}. \quad (10)$$

Since we are considering only finite games, every history has finite length. Thus by starting at the terminal states and going backwards up the game tree, applying Eqs. (7) and (9) at each move, one can compute a strategy σ_1^* such that:

$$\sigma_1^*(m|I) = \begin{cases} 1/|M^*(I,\sigma_1^*,\sigma_2)|, & \text{if } m \in M^*(I|\sigma_1^*,\sigma_2), \\ 0, & \text{otherwise.} \end{cases} \quad (11)$$

THEOREM 3. *Let σ_2 be a strategy for a_2, and σ_1^* be as in Eq. (11). Then σ_1^* is σ_2-optimal.*

Sketch of proof. Let $\bar{\sigma}_1$ be any σ_2-optimal strategy. The basic idea is to show, by induction on the lengths of histories in an information set I, that $EU(I|\sigma_1^*,\sigma_2) \geq EU(I|\bar{\sigma}_1,\sigma_2)$.

The induction goes backwards from the end of the game: the base case is where I contains histories of maximal length, while the inductive case assumes the inequality holds when I contains histories of length $k+1$, and shows it holds when I contains histories of length k. The induction suffices to show that $EU([\langle\rangle]_1|\sigma_1^*,\sigma_2) \geq EU([\langle\rangle]_1|\bar{\sigma}_1,\sigma_2)$, whence from Lemma 1, $EU(\sigma_1^*,\sigma_2) \geq EU(\bar{\sigma}_1,\sigma_2)$. □

Computing σ_1^* is more difficult than computing an optimal strategy in a perfect-information game. Reif [Reif 1984] has shown that the problem of finding a strategy with a guaranteed win is doubly exponential for imperfect-information games (this corresponds to finding σ_1 such that for all σ_2, σ_1 wins).

In the minimax game-tree search algorithms used in perfect-information games, one way of dealing with the problem of intractability is to approximate the utility value of a state by searching to some limited depth d, using a *static evaluation function* $\mathcal{E}(\cdot)$ that returns approximations of the expected utilities of the nodes at that depth, and pretending that the values returned by $\mathcal{E}$ are the nodes' actual utility values. In imperfect-information games we can compute approximate values

for EU in a similar fashion:

$$EU_d(h|\sigma_1^*, \sigma_2) = \begin{cases} \mathcal{E}(h), & \text{if } d = 0, \\ U(h), & \text{if } h \text{ is terminal,} \\ \sum_{m \in M(h)} \sigma_2(m|[h]_2) \cdot EU_{d-1}(h \circ m|\sigma_1^*, \sigma_2), & \text{if it's } a_2\text{'s move,} \\ EU_{d-1}(h \circ \text{argmax}_{m \in M(h)}(EU_d([h \circ m]_1|\sigma_1^*, \sigma_2))), & \text{if it's } a_1\text{'s move,} \end{cases} \tag{12}$$

$$EU_d(I|\sigma_1^*, \sigma_2) = \sum_{h \in I} P(h|I, \sigma_1^*, \sigma_2) \cdot EU_d(h|I, \sigma_1^*, \sigma_2). \tag{13}$$

3.2 Opponent Models

Eqs. (11–12) assume that a_1 knows a_2's strategy σ_2, an assumption that is quite unrealistic in practice. A more realistic assumption is that a_1 has a *model* of a_2 that provides an approximation of σ_2. For example, in perfect-information games, the well-known minimax formula corresponds to an opponent model in which the opponent always chooses the move whose utility value is lowest. We now consider two opponent models for imperfect-information games: the *overconfident* and *paranoid* models.

Overconfidence. The overconfident model assumes a_2 is just choosing moves at random from a uniform distribution; i.e., it assumes a_2's strategy is $\sigma_2(m|I) = 1/|M(I)|$ for every $m \in M(I)$, and second, that a_1's strategy is σ_2-optimal. If we let $OU_d(h) = EU_d(h|\sigma_1^*, \sigma_2)$ and $OU_d(I) = EU_d(I|\sigma_1^*, \sigma_2)$ be the expected utilities for histories and information sets under these assumptions, then it follows from Eqs. (12–13) that:

$$OU_d(h) = \begin{cases} \mathcal{E}(h), & \text{if } d = 0, \\ U(h), & \text{if } h \text{ is terminal,} \\ \sum_{m \in M(h)} \frac{OU_{d-1}(h \circ m)}{|M(h)|}, & \text{if it's } a_2\text{'s move,} \\ OU_{d-1}(h \circ \text{argmax}_{m \in M(h)} OU_d([h \circ m]_1)), & \text{if it's } a_1\text{'s move,} \end{cases} \tag{14}$$

$$OU_d(I) = \sum_{h \in I} (1/|I|) \cdot OU_d(h). \tag{15}$$

If the algorithm searches to a limited depth (Eq. 12 with $d < \max_{h \in H} |h|$), we will refer to the resulting strategy as *limited-depth overconfident*. If the algorithm searches to the end of the game (i.e., $d \geq \max_{h \in H} |h|$), we will refer to the resulting strategy as *full-depth overconfident*; and in this case we will usually write $OU(h)$ rather than $OU_d(h)$.

Paranoia. The paranoid model assumes that a_2 will always make the worst possible move for a_1, i.e., the move that will produce the minimum expected utility over all of the histories in a_1's information set. This model replaces the summation in

the third line of Eq. (12) with a minimization:

$$PU_d(h) = \begin{cases} \mathcal{E}(I), & \text{if } d = 0, \\ U(h), & \text{if } h \text{ is terminal}, \\ PU_{d-1}(h \circ \text{argmin}_{m \in M(h)}(\min_{h' \in [h]_1} PU_d([h \circ m]))), & \text{if it's } a_2\text{'s move}, \\ PU_{d-1}(h \circ \text{argmax}_{m \in M(h)}(\min_{h' \in [h]_1} PU_d([h \circ m]))), & \text{if it's } a_1\text{'s move}, \end{cases} \tag{16}$$

$$PU_d(I) = \min_{h \in I} PU_d(h). \tag{17}$$

Like we did for overconfident search, we will use the terms *limited-depth* and *full-depth* to refer to the cases where $d < \max_{h \in H} |h|$ and $d \geq \max_{h \in H} |h|$, respectively; and for a full-depth paranoid search, we will usually write $PU(h)$ rather than $PU_d(h)$.

In perfect-information games, $PU(h)$ equals h's minimax value. But in imperfect-information games, h's minimax value is the minimum Eq. (11) over all possible values of σ_2; and consequently $PU(h)$ may be less than h's minimax value.

3.3 Comparison with the Minimax Theorem

The best known kinds of strategies for zero-sum games are the strategies based on the famous Minimax Theorem [von Neumann and Morgenstern 1944]. These *minimax strategies* tacitly incorporate an opponent model that we will call the *minimax model*. The minimax model, overconfident model, and paranoid model each correspond to differing assumptions about a_2's knowledge and competence, as we will now discuss.

Let Σ_1 and Σ_2 be the sets of all possible pure strategies for a_1 and a_2, respectively. If a_1 and a_2 use mixed strategies, then these are probability distributions P_1 and P_2 over Σ_1 and Σ_2. During game play, a_1 and a_2 will randomly choose pure strategies σ_1 and σ_2 from P_1 and P_2. Generally they will do this piecemeal by choosing moves as the game progresses, but game-theoretically this is equivalent to choosing the entire strategy all at once.

Paranoia: If a_1 uses a paranoid opponent model, this is equivalent to assuming that a_2 knows in advance the pure strategy σ_1 that a_1 will choose from P_1 during the course of the game, and that a_2 can choose the optimal counter-strategy, i.e., a strategy $P_2^{\sigma_1}$ that minimizes σ_1's expected utility. Thus a_1 will want to choose a σ_1 that has the highest possible expected utility given $P_2^{\sigma_1}$. If there is more than one such σ_1, then a_1's strategy can be any one of them or can be an arbitrary probability distribution over all of them.

Minimax: If a_2 uses a minimax opponent model, this is equivalent to assuming that a_2 will know in advance what a_1's mixed strategy P_1 is, and that a_2 will be competent enough to choose the optimal counter-strategy, i.e., a mixed strategy $P_2^{P_1}$ that minimizes P_1's expected utility. Thus a_1 will want to use a mixed strategy P_1

that has the highest possible expected utility given $P_2^{P_1}$.

In perfect-information games, the minimax model is equivalent to the paranoid model. But in imperfect-information games, the minimax model assumes a_2 has less information than the paranoid model does: the minimax model assumes that a_2 knows the probability distribution P_1 over a_1's possible strategies, and the paranoid model assumes that a_2 knows which strategy a_1 will choose from P_1.

Overconfidence: If a_1 uses an overconfident opponent model, this equivalent to assuming that a_2 knows nothing about (or is not competent enough to figure out) how good or bad each move is, whence a_2 will use a strategy $P_2^{=}$ in which all moves are equally likely. In this case, a_1 will want to choose a strategy σ_1 that has the highest expected utility given $P_2^{=}$. If there is more than one such σ_1, then a_1's strategy can be any one of them or can be an arbitrary probability distribution over all of them.

In both perfect- and imperfect-information games, the overconfident model assumes a_2 has much less information (and/or competence) than in the minimax and paranoid models.

3.4 Handling Large Information Sets

Information sets can be quite large. When they are too large for techniques like the above to run in a reasonable amount of time, there are several options.

Game simplification reduces the size of the information set by creating an analogous game with smaller information sets. This technique has worked particularly well in poker [Billings, Burch, Davidson, Holte, Schaeffer, Schauenberg, and Szafron 2003; Gilpin and Sandholm 2006a; Gilpin and Sandholm 2006b], as it is possible to create a "simpler" game which preserves win probabilities (within some ϵ). However, these approaches apply only to variants of poker, and the technique is not easily generalizable. Given an arbitrary game G other than poker, we know of no general-purpose way of producing a simpler game whose expected utilities accurately reflect expected utilities in G.

State aggregation was first used in the game of sprouts [Applegate, Jacobson, and Sleator 1991], and subsequently has been used in computer programs for games such as bridge (e.g., [Ginsberg 1999]), in which many of the histories in an information set are similar, and hence can be reasoned about as a group rather than individually. For example, if one of our opponents has an ace of hearts and a low heart, it usually does not matter *which* low heart the opponent has: generally all low hearts will lead to an identical outcome, so we need not consider them separately. The aggregation reduces the computational complexity by handling whole sets of game histories in the information set at the same time. However, just as with game simplification, such aggregation techniques are highly game dependent. Given an arbitrary game G, we do not know of a general-purpose way to aggregate states of G in a way that is useful for computing expected utility values in G.

Unlike the previous two techniques, **statistical sampling** [Corlett and Todd

1985] is general enough to fit any imperfect-information game. It works by selecting a manageable subset of the given, large, information set, and doing our computations based on that.

Since we are examining game playing across several imperfect-information games we use the third technique. Let us suppose Γ is an expected utility function such as OU_d or PU_d. In statistical sampling we pick $I' \subset I$ and compute the value of $\Gamma(I')$ in place of $\Gamma(I)$. There are two basic algorithms for doing the sampling:

1. **Batch:** Pick a random set of histories $I' \subset I$, and compute $\Gamma_s(I')$ using the equations given earlier.

2. **Iterative:** Until the available time runs out, repeatedly pick a random $h \in I$, compute $\Gamma(\{h\})$ and aggregate that result with all previous picks.

The iterative method is preferable because it is a true anytime algorithm: it continues to produce increasingly accurate estimates of $\Gamma(I)$ until no more time is available. In contrast, the batch method requires guessing how many histories we will be able to compute in that time, picking a subset I' of the appropriate size, and hoping that the computation finishes before time is up. For more on the relative advantages of iterative and batch sampling, see [Russell and Wolfe 2005].

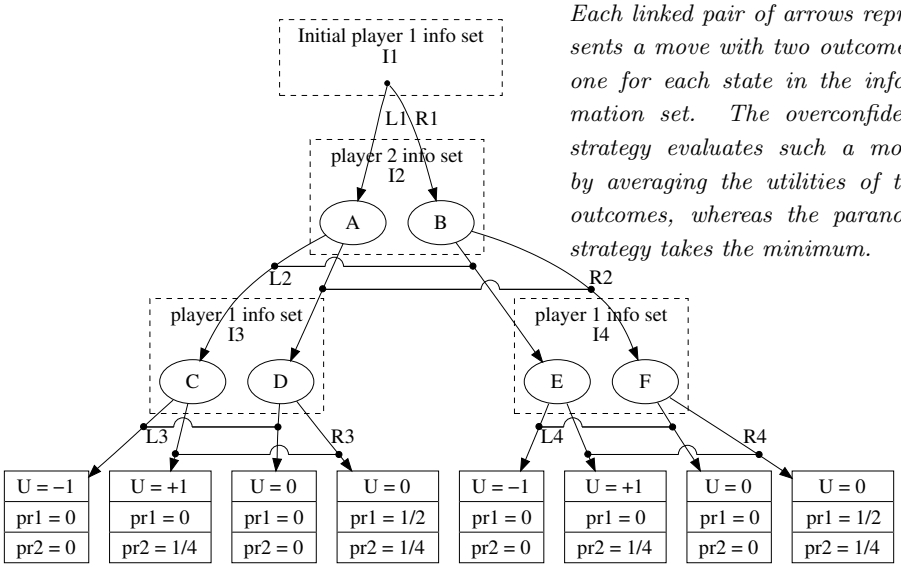
Statistical sampling, unlike game simplification and state aggregation, can be used for arbitrary imperfect-information games rather than just on games that satisfy special properties. Consequently, it is what we use in our experiments in Section 5.

4 Analysis

Since paranoid and overconfident play both depend on opponent models that may be unrealistic, which of them is better in practice? The answer is not completely obvious. Even in games where each player's moves are completely hidden from the other player, it is not hard to create games in which the paranoid strategy outplays the overconfident strategy and vice-versa. We now give examples of games with these properties.

Figures 2 and 3, respectively, are examples of situations in which paranoid play outperforms overconfident play and vice versa. As in Figure 1, the games are shown in tree form in which each dotted box represents an information set. At each leaf node, U is the payoff for player 1. Based on these values of U, the table gives, the probabilities of moving left (L) and right (R) at each information set in the tree, for both the overconfident and paranoid strategies. At each leaf node, pr1 is the probability of reaching that node when player 1 is overconfident and player 2 is paranoid, and pr2 is the probability of reaching that node when player 2 is overconfident and player 1 is paranoid.

In Figure 2, the paranoid strategy outperforms the overconfident strategy, because of the differing choices the strategies will make at the information set I2:



Each linked pair of arrows represents a move with two outcomes: one for each state in the information set. The overconfident strategy evaluates such a move by averaging the utilities of the outcomes, whereas the paranoid strategy takes the minimum.

Info set	Overconfident strategy		Paranoid strategy	
I1	$P(\text{L1}) = 1/2$	$P(\text{R1}) = 1/2$	$P(\text{L1}) = 1/2$	$P(\text{R1}) = 1/2$
I2	$P(\text{L2}) = 1/2$	$P(\text{R2}) = 1/2$	$P(\text{L2}) = 0$	$P(\text{R2}) = 1$
I3	$P(\text{L3}) = 0$	$P(\text{R3}) = 1$	$P(\text{L3}) = 0$	$P(\text{R3}) = 1$
I4	$P(\text{L4}) = 0$	$P(\text{R4}) = 1$	$P(\text{L4}) = 0$	$P(\text{R4}) = 1$

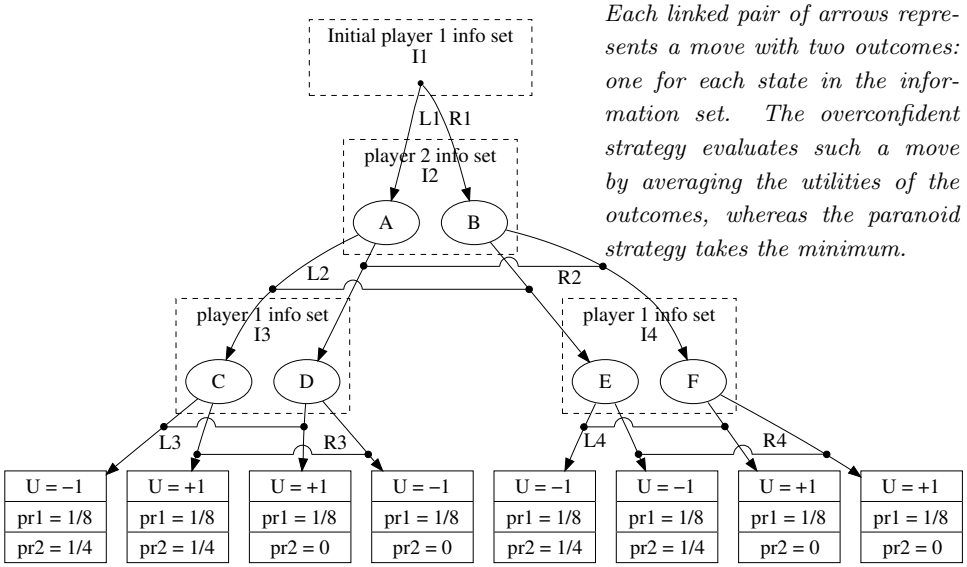
Figure 2. An imperfect-information game in which paranoid play beats overconfident play. If an overconfident player plays against a paranoid player and each player has an equal chance of moving first, the expected utilities are -0.25 for the overconfident player and 0.25 for the paranoid player.

- Suppose player 1 is overconfident and player 2 is paranoid. Then at information set I2, player 2 assumes its opponent will always choose the worst possible response. Hence when choosing a move at I2, player 2 thinks it will lose if it chooses L2 and will tie if it chooses R2, so it chooses R2 to avoid the anticipated loss.

- Suppose player 1 is paranoid and player 2 is overconfident. Then at information set I2, player 2 assumes its opponent is equally likely to move left or right. Hence when choosing a move at I2, player 2 thinks that both moves have the same expected utility, so it will choose between them at random—which is a mistake, because its paranoid opponent will win the game by moving right in both information sets I3 and I4.

Figure 3 shows a game in which the overconfident strategy outperforms the paranoid strategy. Again, the pertinent information set is I2:

- Suppose overconfident play is player 1 and paranoid play is player 2. Then



Each linked pair of arrows represents a move with two outcomes: one for each state in the information set. The overconfident strategy evaluates such a move by averaging the utilities of the outcomes, whereas the paranoid strategy takes the minimum.

Info set	Overconfident strategy		Paranoid strategy	
I1	$P(\text{L1}) = 1/2$	$P(\text{R1}) = 1/2$	$P(\text{L1}) = 1/2$	$P(\text{R1}) = 1/2$
I2	$P(\text{L2}) = 1$	$P(\text{R2}) = 0$	$P(\text{L2}) = 1/2$	$P(\text{R2}) = 1/2$
I3	$P(\text{L3}) = 1/2$	$P(\text{R3}) = 1/2$	$P(\text{L3}) = 1/2$	$P(\text{R3}) = 1/2$
I4	$P(\text{L4}) = 1/2$	$P(\text{R4}) = 1/2$	$P(\text{L4}) = 1/2$	$P(\text{R4}) = 1/2$

Figure 3. An imperfect-information game where overconfident play beats paranoid play. If an overconfident player plays against a paranoid player and each player has an equal chance of moving first, the expected utilities are 0.25 for the overconfident player and −0.25 for the paranoid player.

paranoid play, assuming the worst, believes both move L2 and R2 are losses. R2 is a loss because the opponent may have made move R1 resulting in a forced loss for player 2 at node F, and L2 is a loss because the opponent may have made move L1 and then may make move R4 resulting in a loss for player 2. Since there is a potential loss in all cases, paranoid play chooses both cases with equal probability.

- When overconfident play is player 2, it makes move L2 at I2, on the theory that the opponent was equally likely to make moves L1 and R1 and therefore giving it a 50% probability of ending up in node E, which is a forced win for player 2. Against paranoid play as player 1, this is a good move, since paranoid play actually does make moves L1 and R1 with 50% probability.

These two examples show that neither strategy is guaranteed to be better in all cases: sometimes paranoid play outperforms overconfident play, and sometimes vice versa. So to determine their relative worth, deeper analysis is necessary.

4.1 Analysis of Overconfidence Performance in Perfect Information Games

Let s be a state in a perfect-information zero-sum game. We will say that a child s' of s is *minimax-optimal* if $\mu(s') \geq \mu(s'')$ for every child s'' of s, where $\mu(s)$ is the minimax value for the player to move at s. A *minimax strategy* is any strategy that will always move to a minimax-optimal node. In the game-tree search literature, minimax strategies have often been called "perfect play" because they produce the highest possible value against an opponent who is also using a minimax strategy.

In perfect-information zero-sum games, $PU(s) = \mu(s)$ at every state s, hence full-depth paranoid play is a minimax strategy. Surprisingly, if the only outcomes are wins and losses (or equivalently, utility values of 1 and -1), full-depth *overconfident* play is also a minimax strategy. To prove this result, we first need a lemma:

LEMMA 4. *Let G be any finite two-player perfect-information game whose outcomes all have utility 1 or -1. At every state s, if $\mu(s) = 1$ then $OC(s) = 1$, and if $\mu(s) = -1$ then $OC(s) \in [-1, 1)$.*

Sketch of proof. This is proven by induction on the height of the state s under consideration. The base case occurs for with terminal nodes of height 0 for which the lemma follows trivially. The inductive case supposes the lemma holds for all states of height k and shows algebraically for states s of height $k+1$ in each of four possible cases: (1) if it is a_1's move and $\mu(s) = -1$ then $OC(s) \in [-1, 1)$, (2) if it is a_1's move and $\mu(s) = 1$ then $OC(s) = 1$, (3) if it is a_2's move and $\mu(s) = -1$ then $OC(s) \in [-1, 1)$, and (4) if it is a_2's move and $\mu(s) = 1$ then $OC(s) = 1$. Since the game allows only wins and losses (so that $\mu(s)$ is 1 or -1), these are all the possibilities. □

THEOREM 5. *Let G be any finite two-player perfect-information game whose outcomes all have utility 1 or -1. At every nonterminal state s, the overconfident strategy, σ_O, will move to a state s' that is minimax-optimal.*

Proof. Immediate from the lemma. □

This theorem says that in head-to-head play in perfect-information games allowing only wins or losses, the full-depth overconfident and full-depth paranoid strategies will be evenly matched. In the experimental section, we will see this to hold in practice.

4.2 Discussion

Paranoid play. When using paranoid play a_1 assumes that a_2 has always and will always make the worst move possible for a_1, but a_1 does this *given only a_1's information set*. This means that for any given information set, the paranoid player will find the history in the information set that is least advantageous to itself and make moves as though that were the game's actual history *even when the game's actual*

history is any other member of the information set. There is a certain intuitively appealing protectionism occurring here: an opponent that happens to have made the perfect moves cannot trap the paranoid player. However, it really is not clear exactly how well a paranoid player will do in an imperfect-information game, for the following reasons:

- There is no reason to necessarily believe that the opponent has made those "perfect" moves. In imperfect-information games, the opponent has different information than the paranoid player, which may not give the opponent enough information to make the perfect moves paranoid play expects.

- Against non-perfect players, the paranoid player may lose a lot of potentially winnable games. The information set could contain thousands of histories in which a particular move m is a win; if that move is a loss on just one history, and there is another move m' which admits no losses (and no wins), then m will not be chosen.[3]

- In games such as kriegspiel, in which there are large and diverse information sets, usually every information set will contain histories that are losses, hence paranoid play will evaluate all of the information sets as losses. In this case, all moves will look equally terrible to the paranoid player, and paranoid play becomes equivalent to random play.[4]

We should also note the relationship paranoid play has to the "imperfection" of the information in the game. A game with large amounts of information and small information sets should see better play from a paranoid player than a game with large information sets. The reason for this is that as we get more information about the actual game state, we can be more confident that the move the paranoid player designates as "worst" is a move the opponent can discover and make in the actual game. The extreme of this is a perfect information game, where paranoid play has proven quite effective: it is minimax search. But without some experimentation, it is not clear to what extent smaller amounts of information degrade paranoid play.

Overconfident play. Overconfident play assumes that a_2 will, with equal probability, make all available moves regardless of what the available information tells a_2 about each move's expected utility. The effect this has on game play depends on the extent to which a_2's moves diverge from random play. Unfortunately for overconfidence, many interesting imperfect-information games implicitly encourage

[3] This argument assumes that the paranoid player examines the entire information set rather than a statistical sample as discussed in Section 3.4. If the paranoid player examines a statistical sample of the information set, there is a good chance that the statistical sample will not contain the history for which m is a loss. Hence in this case, statistical sampling would actually *improve* the paranoid player's play.

[4] We have verified this experimentally in several of the games in the following section, but omit these experiments due to lack of space.

non-random play. In these games the overconfident player will not adequately consider the risks of its moves. The overconfident player, acting under the theory that the opponent is unlikely to make a particular move, will many times not protect itself from a potential loss.

However, depending on the amount of information in the imperfect-information game, the above problem may not be as bad as it seems. For example, consider a situation where a_1, playing overconfidently, assumes the opponent is equally likely to make each of the ten moves available in a_1's current information set. Suppose that each move is clearly the best move in exactly one tenth of the available histories. Then, despite the fact that the opponent is playing a deterministic strategy, random play is a good opponent model given the information set. This sort of situation, where the model of random play is reasonable despite it being not at all related to the opponent's actual mixed strategy, is more likely to occur in games where there is less information. The larger the information set, the more likely it is that every move is best in enough histories to make that move as likely to occur as any other. Thus in games where players have little information, there may be a slight advantage to overconfidence.

Comparative performance. The above discussion suggests that (1) paranoid play should do better in games with "large" amounts of information, and (2) overconfident play might do better in games with "small" amounts of information. But will overconfident play do better than paranoid play? Suppose we choose a game with small amounts of information and play a paranoid player against an overconfident player: what should the outcome be? Overconfident play has the advantage of probably not diverging as drastically from the theoretically correct expected utility of a move, while paranoid play has the advantage of actually detecting and avoiding bad situations – situations to which the overconfident player will not give adequate weight.

Overall, it is not at all clear from our analysis how well a paranoid player and an overconfident player will do relative to each other in a real imperfect-information game. Instead, experimentation is needed.

5 Experiments

In this section we report on our experimental comparisons of overconfident versus paranoid play in several imperfect-information games.

One of the games we used was kriegspiel, an imperfect-information version of chess [Li 1994; Li 1995; Ciancarini, DallaLibera, and Maran 1997; Sakuta and Iida 2000; Parker, Nau, and Subrahmanian 2005; Russell and Wolfe 2005]. In kriegspiel, neither player can observe anything about the other player's moves, except in cases where the players directly interact with each other. For example, if a_1 captures one of a_2's pieces, a_2 now knows that a_1 has a piece where a_2's piece used to be. For more detail, see Section 5.2.

In addition, we created imperfect-information versions of three perfect-

information games: P-games [Pearl 1984], N-games [Nau 1982a], and a simplified version of kalah [Murray 1952]. We did this by hiding some fraction $0 \leq h \leq 1$ of each player's moves from the other player. We will call h the *hidden factor*, because it is the fraction of information that we hide from each player: when $h = 0$, each player can see all of the other player's moves; when $h = 1$, neither player can see any of the other player's moves; when $h = 0.2$, each player can see 20% of the other player's moves; and so forth.

In each experiment, we played two players head-to-head for some number of trials, and averaged the results. Each player went first on half of the trials.

5.1 Experiments with Move-Hiding

We did experiments in move-hiding variants of simple perfect information games. These experiments were run on 3.4 GHz Xeon processors with at least 2 GB of RAM per core. The programs were written in OCaml. All games were 10-ply long, and each player searched all the way to the end of the game.

Hidden-move P-game experiments. P-games were invented by Judea Pearl [Pearl 1981], and have been used in many studies of game-tree search (e.g., [Nau 1982a; Pearl 1984]). They are two-player zero-sum games in which the game tree has a constant branching factor b, fixed game length d, and fixed probability P_0 that the first player wins at any given leaf node.[5] One creates a P-game by randomly assigning "win" and "loss" values to the b^d leaf nodes.

We did a set of experiments with P-games with $P_0 = 0.38$, which is the value of P_0 most likely to produce a nontrivial P-game [Nau 1982b]. We used depth $d = 10$, and varied the branching factor b. We varied the hidden factor h from 0 to 1 by increments of 0.2, so that the number of hidden moves varied from 0 to 10. In particular, we hid a player's m^{th} move if $\lfloor m \cdot h \rfloor > \lfloor (m-1) \cdot h \rfloor$. For instance, in a game where each player makes 5 moves and the hidden factor is 0.6, then the 2^{nd}, 4^{th}, and 5^{th} moves of both players are hidden.

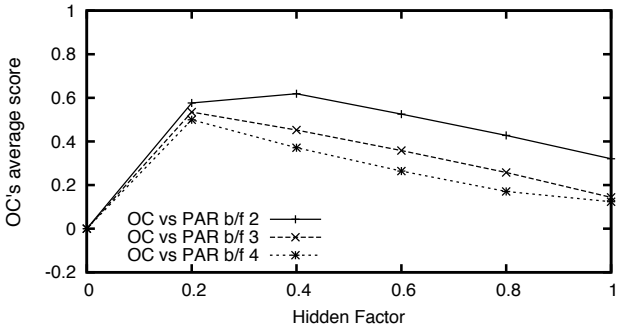
For each combination of parameters, we played 2000 games: 1000 in which one of the players moved first, and 1000 in which the other player moved first. Thus in each of our figures, each data point is the average of 2000 runs.

Figure 4(a) shows the results of head-to-head play between the overconfident and paranoid strategies. These results show that in hidden-move P-games, paranoid play does indeed perform worse than overconfident play with hidden factors greater than 0. The results also confirm theorem 5, since overconfident play and paranoid play did equally well with hidden factor 0. From these experiments, it seems that paranoid play may not be as effective in imperfect-information games as it is in perfect information games.

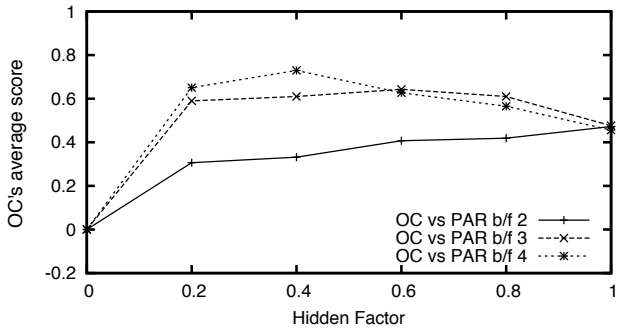
Hidden-move N-game experiments. P-games are known to have a property called *game-tree pathology* that does not occur in "natural" games such as chess [Nau

[5] Hence [Pearl 1984] calls P-games (d, b, P_0)-games.

(a) *Hidden-move P-games.* Each data point is an average of at least 72 trials.



(b) *Hidden-move N-games.* Each data point is an average of at least 39 trials.



(c) *Hidden-move kalah.* Each data point is an average of at least 125 randomly generated initial states.

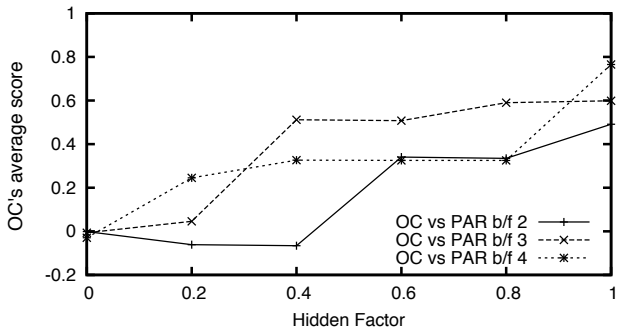


Figure 4. Average scores for overconfident (OC) play against paranoid (PAR) play.

1982a], and we wanted to ascertain whether this property might have influenced our experimental results on hidden-move P-games. N-games are similar to P-games but do not exhibit game-tree pathology, so we did a similar set of experiments on hidden-move N-games.

An N-game is specified by a triple (d, b, P_0), where d is the game length, b is the branching factor, and P_0 is a probability. An N-game specified by this triple has a game tree of height d and branching factor b, and each arc in the game tree is randomly assigned a value of $+1$ with probability P_0, or -1 otherwise. A leaf node is a win for player 1 (and a loss for player 2) if the sum of the values on the arcs between the root and the leaf node is greater than zero; otherwise the leaf node is

a loss for player 1 (and a win for player 2).

Figure 4(b) shows our experimental results for hidden-move N-games. Just as before, overconfident and paranoid play did equally well with hidden factor 0, and overconfident play outperformed paranoid play with hidden factors greater than 0.

Kalah experiments. Kalah [Murray 1952] is also called mankalah, mancala, warri, and other names. It is an ancient African game played on a board with a number of pits that contain seeds, in which the objective is to acquire more seeds than the opponent, either by moving them to a special pit (called a kalah) or by capturing them from the opponent's pits.

In kalah, there are two rows of 6 pits. Flanking the rows of pits on both sides are the larger kalahs. Players sit on opposite sides of the board with one of the rows of pits nearer to each player. Each player owns the kalah on their left. The game starts with 6 stones in each of the pits except the kalahs. The player moves by picking up all the stones from one of the pits in the near row and placing one stone in each pit clockwise around the board including their kalah but excluding the opponent's kalah. If the last stone is placed in their kalah, the player moves again. If the last stone is placed in an empty pit, the player moves all stones from the opposite pit to their kalah. The game ends when the player to move has no moves because all pits on their side are empty. At that point, all stones in pits on the other player's side are placed in the player to move's kalah and the player with the most stones wins; ties occur when both plays own the same number of stones.

Because of the computation requirements of playing a full game of kalah, our experiments were on a simplified version of kalah that we call randomized kalah. The game differs from kalah in several ways:

- We vary the number of pits on the board. This varies the branching factor.
- To ensure a constant branching factor, we allow players to "move" from a pit that contains no stones. These are null moves that have no effect on the board.
- We end the game after 10 ply, to ensure that the algorithms can search the entire tree.
- We eliminate the move-again rule, to ensure alternating moves by the players.
- We start with a random number of stones in each pit to ensure that at each branching factor there will be games with non-trivial decisions.

Since randomized kalah is directly motivated by a very old game that people still play, its game trees are arguably much less "artificial" than those of P-games or N-games.

The results of playing overconfidence versus paranoia in hidden-move versions of randomized kalah are shown in Figure 4(c). The results are roughly similar to the

P-game and N-game results, in the sense that overconfidence generally outperforms paranoia; but the results also differ from the P-game and N-game results in several ways. First, overconfidence generally does better at high hidden factors than at low ones. Second, paranoia does slightly better than overconfidence at hidden factor 0 (which does not conflict with Theorem 5, since kalah allows ties). Third, paranoia does better than overconfidence when the branching factor is 2 and the hidden factor is 0.2 or 0.4. These are the only results we saw where paranoia outperformed overconfidence.

The fact that with the same branching factor, overconfidence outperforms paranoia with hidden factor 0.6, supports the hypothesis that as the amount of information in the game decreases, paranoid play performs worse with respect to overconfident play. The rest of the results support that hypothesis as well: overconfidence generally increases in performance against paranoia as the hidden factor increases.

5.2 Kriegspiel Experiments

For experimental tests in an imperfect-information game people actually play, we used kriegspiel, an imperfect-information version of chess in which the players cannot see their opponent's pieces. Kriegspiel is useful for this study because (i) it is clearly a game where each player has only a small amount of information about the current state, and (ii) due to its relationship to chess, it is complicated enough strategically to allow for all sorts of subtle and interesting play. A further advantage to kriegspiel is that it is played competitively by humans even today [Li 1994; Li 1995; Ciancarini, DallaLibera, and Maran 1997].

Kriegspiel is a chess variant played with a chess board. When played in person, it requires three chess kits: one for each player and one for the referee. All boards are set up as in normal chess, but neither player is allowed to see their opponent's or the referee's board. The players then move in alternation as in standard chess, keeping their moves hidden from the other player. All player's moves are also played by the referee on the referee's board. Since neither player can see the referee's board, the referee acts as a mediator, telling the players if the move they made is legal or illegal, and giving them various other observations about the move made. We use the ICC's kriegspiel observations, described at `http://www.chessclub.com/help/Kriegspiel`. Observations define the information sets. Any two histories that have the same observations at each move and all the same moves for one of the players are in the same information set.

When played on the internet, the referee's job can be automated by a computer program. For instance, on the Internet Chess Club one can play kriegspiel, and there have been thousands of kriegspiel games played on that server.

We ran our experiments on a cluster of computers runing linux, with between 900 MB and 1.5 GB RAM available to each process. The processors were Xeons, Athlons, and Pentiums, ranging in clockspeed from 2 GHz to 3.2 GHz. We used

Table 1. Average scores for overconfident play against paranoid play, in 500 kriegspiel games using the ICC ruleset. d is the search depth.

Over-	Paranoid		
confident	$d = 1$	$d = 2$	$d = 3$
$d = 1$	+0.084	+0.186	+0.19
$d = 2$	+0.140	+0.120	+0.156
$d = 3$	+0.170	+0.278	+0.154

Table 2. Average scores for overconfident and paranoid play against HS, with 95% confidence intervals. d is the search depth.

d	Paranoid	Overconfident
1	–0.066 ± 0.02	+0.194 ± 0.038
2	+0.032 ± 0.035	+0.122 ± 0.04
3	+0.024 ± 0.038	+0.012 ± 0.042

time controls and always forced players in the same game to ensure the results were not biased by different hardware. The algorithms were written in C++. The code used for overconfident and paranoid play is the same, with the exception of the opponent model. We used a static evaluation function that was developed to reward conservative kriegspiel play, as our experience suggests such play is generally better. It uses position, material, protection and threats as features.

The algorithms used for kriegspiel are depth-limited versions of the paranoid and overconfident players. To handle the immense information-set sizes in kriegspiel, we used iterative statistical sampling (see Section 3.4). To get a good sample with time control requires limiting the search depth to at most three ply. Because time controls remain constant, the lower search depths are able to sample many more histories than the higher search depths.

Head-to-head overconfident vs. paranoid play. We did experiments comparing overconfident play to paranoid play by playing the two against each other. We gave the algorithms 30 seconds per move and played each of depths one, two, and three searches against each other. The results are in Table 1. In these results, we notice that overconfident play consistently beats paranoid play, regardless of the depth of either search. This is consistent with our earlier results for hidden-move games (Section 5.1); and, in addition, it shows overconfident play doing better than paranoid play in a game that people actually play.

HS versus overconfidence and paranoia. We also compared overconfident and paranoid play to the hybrid sampling (HS) algorithm from our previous work [Parker, Nau, and Subrahmanian 2005]. Table 2 presents the results of the experiments, which show overconfidence playing better than paranoia except in depth three search, where the results are inconclusive. The inconclusive results at depth three (which are an average over 500 games) may be due to the sample sizes achieved via iterative sampling. We measured an average of 67 histories in each sample at depth three, which might be compared to an average of 321 histories in each sample at depth two and an average of 1683 histories at depth one. Since both algorithms use iterative sampling, it could be that at depth three, both algorithms examine

insufficient samples to do much better than play randomly.

In every case, overconfidence does better than paranoia against HS. Further, overconfidence outperforms HS in every case (though sometimes without statistical significance), suggesting that information-set search is an improvement over the techniques used in HS.

6 Related Work

There are several imperfect-information game-playing algorithms that work by treating an imperfect-information game as if it were a collection of perfect-information games [Smith, Nau, and Throop 1998; Ginsberg 1999; Parker, Nau, and Subrahmanian 2005]. This approach is useful in imperfect-information games such as bridge, where it is not the players' moves that are hidden, but instead some information about the initial state of the game. The basic idea is to choose at random a collection of states from the current information set, do conventional minimax searches on those states as if they were the real state, then aggregate the minimax values returned by those searches to get an approximation of the utility of the current information set. This approach has some basic theoretical flaws [Frank and Basin 1998; Frank and Basin 2001], but has worked well in games such as bridge.

Poker-playing computer programs can be divided into two major classes. The first are programs which attempt to approximate a Nash equilibrium. The best examples of these are PsOpti [Billings, Burch, Davidson, Holte, Schaeffer, Schauenberg, and Szafron 2003] and GS1 [Gilpin and Sandholm 2006b]. The algorithms use an intuitive approximation technique to create a simplified version of the poker game that is small enough to make it feasible to find a Nash equilibrium. The equilibrium can then be translated back into the original game, to get an approximate Nash equilibrium for that game. These algorithms have had much success but differ from the approach in this paper: unlike any attempt to find a Nash equilibrium, information-set search simply tries to find the optimal strategy against a given opponent model. The second class of poker-playing programs includes Poki [Billings, Davidson, Schaeffer, and Szafron 2002] which uses expected value approximations and opponent modeling to estimate the value of a given move and Vexbot [Billings, Davidson, Schauenberg, Burch, Bowling, Holte, Schaeffer, and Szafron 2004] which uses search and adaptive opponent modeling.

The above works have focused specifically on creating successful programs for card games (bridge and poker) in which the opponents' moves (card plays, bets) are observable. In these games, the hidden information is which cards went to which players when the cards were dealt. Consequently, the search techniques are less general than information-set search, and are not directly applicable to hidden-move games such as kriegspiel and the other games we have considered in this paper.

7 Conclusion

We have introduced a recursive formulation of the expected value of an information set in an imperfect information game. We have provided analytical results showing that this expected utility formulation plays optimally against any opponent if we have an accurate model of the opponent's strategy.

Since it is generally not the case that the opponent's strategy is known, the question then arises as to what the recursive search should assume about an opponent. We have studied two opponent models, a "paranoid" model that assumes the opponent will choose the moves that are best for them, hence worst for us; and an "overconfident" model that assumes the opponent is making moves purely at random.

We have compared the overconfident and paranoid models in kriegspiel, in an imperfect-information version of kalah, and in imperfect-information versions of P-games [Pearl 1984] and N-games [Nau 1982a]. In each of these games, the overconfident strategy consistently outperformed the paranoid strategy. The overconfident strategy even outperformed the best of the kriegspiel algorithms in [Parker, Nau, and Subrahmanian 2005].

These results suggest that the usual assumption in perfect-information game tree search—that the opponent will choose the best move possible—is not as effective in imperfect-information games.

Acknowledgments: This work was supported in part by AFOSR grant FA95500610405, NAVAIR contract N6133906C0149, DARPA IPTO grant FA8650-06-C-7606, and NSF grant IIS0412812. The opinions in this paper are those of the authors and do not necessarily reflect the opinions of the funders.

References

Applegate, D., G. Jacobson, and D. Sleator (1991). Computer analysis of sprouts. Technical report, Carnegie Mellon University.

Billings, D., N. Burch, A. Davidson, R. Holte, J. Schaeffer, T. Schauenberg, and D. Szafron (2003). Approximating game-theoretic optimal strategies for full-scale poker. In *IJCAI*, pp. 661–668.

Billings, D., A. Davidson, J. Schaeffer, and D. Szafron (2002). The challenge of poker. *Artif. Intell. 134*, 201–240.

Billings, D., A. Davidson, T. Schauenberg, N. Burch, M. Bowling, R. Holte, J. Schaeffer, and D. Szafron (2004). Game tree search with adaptation in stochastic imperfect information games. *Computers and Games 1*, 21–34.

Ciancarini, P., F. DallaLibera, and F. Maran (1997). Decision Making under Uncertainty: A Rational Approach to Kriegspiel. In J. van den Herik and J. Uiterwijk (Eds.), *Advances in Computer Chess 8*, pp. 277–298.

Corlett, R. A. and S. J. Todd (1985). A monte-carlo approach to uncertain inference. In *AISB-85*, pp. 28–34.

Frank, I. and D. Basin (2001). A theoretical and empirical investigation of search in imperfect information games. *Theoretical Comp. Sci. 252*, 217–256.

Frank, I. and D. A. Basin (1998). Search in games with incomplete information: A case study using bridge card play. *Artif. Intell. 100*(1-2), 87–123.

Gilpin, A. and T. Sandholm (2006a). Finding equilibria in large sequential games of imperfect information. In *EC '06*, pp. 160–169.

Gilpin, A. and T. Sandholm (2006b). A texas hold'em poker player based on automated abstraction and real-time equilibrium computation. In *AAMAS '06*, pp. 1453–1454.

Ginsberg, M. L. (1999). GIB: Steps toward an expert-level bridge-playing program. In *IJCAI-99*, pp. 584–589.

Li, D. (1994). *Kriegspiel: Chess Under Uncertainty*. Premier.

Li, D. (1995). *Chess Detective: Kriegspiel Strategies, Endgames and Problems*. Premier.

Murray, H. J. R. (1952). *A History of Board Games other than Chess*. London, UK: Oxford at the Clarendon Press.

Nau, D. S. (1982a). An investigation of the causes of pathology in games. *Artif. Intell. 19*(3), 257–278.

Nau, D. S. (1982b). The last player theorem. *Artif. Intell. 18*(1), 53–65.

Nau, D. S. (1983). Decision quality as a function of search depth on game trees. *JACM 30*(4), 687–708.

Osborne, M. J. and A. Rubinstein (1994). *A Course In Game Theory*. MIT Press.

Parker, A., D. Nau, and V. Subrahmanian (2005, August). Game-tree search with combinatorially large belief states. In *IJCAI*, pp. 254–259.

Pearl, J. (1981, August). Heuristic search theory: Survey of recent results. In *Proc. Seventh Internat. Joint Conf. Artif. Intel.*, Vancouver, Canada, pp. 554–562.

Pearl, J. (1984). *Heuristics: intelligent search strategies for computer problem solving*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Reif, J. (1984). The complexity of two-player games of incomplete information. *Jour. Computer and Systems Sciences 29*, 274–301.

Russell, S. and J. Wolfe (2005, August). Efficient belief-state and-or search, with application to kriegspiel. In *IJCAI*, pp. 278–285.

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

Smith, S. J. J., D. S. Nau, and T. Throop (1998). Computer bridge: A big win for AI planning. *AI Magazine 19*(2), 93–105.

von Neumann, J. and O. Morgenstern (1944). *Theory of Games and Economic Behavior*. Princeton University Press.

6

Heuristic Search: Pearl's Significance from a Personal Perspective

Ira Pohl

1 Introduction

This paper is about heuristics, and the significance of Judea Pearl's work to the field. The impact of Pearl's monograph was transformative. It heralded a third wave in the practice and theory of heuristic search. First there were the pioneering search algorithms without a theoretical basis, such as GPS or GT. Second came the Nilsson [1980] work that formulated a basic theory of A*. Third, in 1984, was Judea Pearl adding depth and breadth to this theory and adding a more sophisticated probabilistic context. Judea Pearl's book, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, was a tour de force summarizing the work of three decades.

Heuristic Search is a holy grail of Artificial Intelligence. It attempts to be a universal methodology to achieve AI, demonstrating early success in an era when AI was largely experimental. Pre-1970 programs that were notable include the Doran-Michie Graph Traverser [Doran and Michie 1966], the Art Samuel checker program [Samuel 1959], and the Newell, Simon, and Shaw General Problem Solver [Newell and Simon 1972]. These programs showed what could be called *intelligence* across a gamut of puzzles, games, and logic problems. Having no theoretical basis for predicting success, they were tested and compared to human performance.

The lack of theory for heuristic search changed in 1968 with the publication by Hart, Nilsson, and Raphael [1968] of their A* algorithm and its analysis. A* was provably optimum under some theoretical assumptions. The outcome of this work at the SRI robotics group fueled a series of primary results including my own, and later Pearl's. Pearl's book [Pearl 1984] captured and synthesized much of the A* work, including my work from the late 1960's [Pohl 1967; Pohl 1969] through 1977 [Pohl 1970a; Pohl 1970b; Pohl 1971; Pohl 1973; Pohl 1977]. It built a solid theoretical structure for heuristic search, and inspired much of my own and others subsequent work [Ratner and Pohl 1986; Ratner and Warmuth 1986; Kaindl and Kaintz 1997; Politowski 1984].

2 Early Experimentation

In the 1960's there were three premiere AI labs at US universities: CMU, MIT and Stanford; there was one such lab in England: the Machine Intelligence group at

Edinburgh; and there was the AI group at SRI. Each had particular strengths and visions. The CMU group led by Allen Newell and Herb Simon [Newell and Simon 1972] took a cognitive simulation approach. Their primary algorithmic framework was GPS, the General Problem Solver. This algorithm could be viewed as an elementary divide and conquer strategy. Guidance was based on detecting differences between partial solution and goal states. To make progress this algorithm attempted to apply operators to partial solutions, and reduce the difference with a goal state. It demonstrated that a heuristic search strategy could be applied to a wide array of problems that were associated with human intelligence. These included combinatorial puzzles such as crypt-arithmetic and towers of Hanoi.

The Graph Traverser [Doran and Michie 1966] reduced AI questions to the task of heuristic search. It was deployed on the 8 puzzle, a classic combinatorial puzzle typical of mildly challenging human amusements. "The objective of the 8-Puzzle is to rearrange a given initial configuration of eight numbered tiles arranged on a 3 x 3 board into a given final configuration called the goal state [Pearl 1984, p. 6].". Michie and Doran tried to obtain efficient search by discovering useful and computationally simple heuristics that measured perceived effort toward a solution. An example was "how many tiles are out of their goal space." The Graph Traverser demonstrated that a graph representation was a useful general perspective in problem solving, and that computationally knowledgeable heuristics could efficiently guide search.

The MIT AI lab pioneered many projects in both robotics and advanced problem solving. Marvin Minsky and his collaborators solved relatively difficult mathematical problems such as word algebra problems and calculus problems [Slagle 1963]. They used context, mathematical models and search to solve these problems. Ultimately this work led to programs such as Mathematica. They gave a plausible argument for what could be described as $knowledge + deduction = intelligence$.

The Stanford AI lab was led by John McCarthy, who was important in two regards: computational environment and logical representations. McCarthy pioneered with LISP and time sharing: tools and schemes that would profoundly impact the entire computational community. He championed predicate logic as a uniform and complete representation of what was needed to express and reason about the world.

The SRI lab, originally affiliated with Stanford, but later independent, was engaged in robotics. A concern was how an autonomous mobile robot could efficiently navigate harsh terrain, such as the moon. Here the emphasis was more engineering and algorithmic. Here the question was how something could be made to work efficiently, not whether it simulated human intelligence or generated a comprehensive theory of inference.

3 Early Theory

The Graph Traverser was a heuristic path finding algorithm attempting to minimize search steps to a goal node. This approach was experimental and explored how to

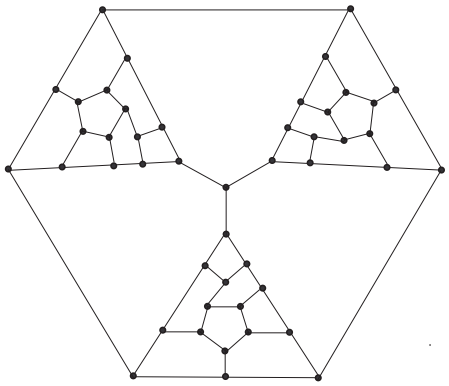


Figure 1. Tutte's graph

formulate and test for useful heuristics. The Dijkstra shortest path algorithm [Dijkstra 1959] was a combinatorial algorithm. It was an improvement on earlier graph theoretic and operations research algorithms for the pure shortest path optimization problem. A* developed in Hart, Nilsson, and Raphael [1968], is the adaptation of Dijkstra's shortest path algorithm to incorporate admissible heuristics. It preserved the need to find an optimal path, while using heuristics that attempted to minimize search.

In the period 1966-1969, I worked on graph theory algorithms, such as the Dijkstra algorithm. The Stanford Computer Science department was dominated by numerical analysts with a strong algorithmic approach. The AI group was metamathematical and inference oriented. Don Knuth had yet to come to Stanford, so there was not yet any systematic work or courses offered on combinatorial algorithms. I was asked to evaluate PL1 for use at Stanford and SLAC and decided to test it out by building a graph algorithm library.

I enjoyed puzzle-like problems and had an idea for improving Warnsdorf's rule for finding a Knight's tour. A Knight's tour is a Hamiltonian path on the 64 square chessboard whose edge connectivity is determined by Knight moves. In graph theory terms the rule was equivalent to going to an unvisited node of minimum out-degree. Warnsdorf's rule is an example of a classic greedy heuristic. I modified it to find a Hamiltonian in the 46 node Tutte's graph [Pohl 1967; Tutte 1946], as well as showed that it worked well on Knight's Tours.

At that time, the elegance of Dijkstra's work had impressed me, and I sought to improve his shortest path algorithm by implementing it bidirectionally. An early attempt by Nicholson [1966] had proved incorrect because of an error in the stopping criteria. These ideas conspired to lead me to test my methods further as variations on A* [Pohl 1971].

The first half of Pearl [1984], Chapter 3 is a sophisticated and succinct summation of search results through 1978. Three theorems principally due to Hart, Nilsson, and Raphael [1968] are central to early theory:

- A* is complete even on infinite graphs [Pearl 1984, Theorem 1, p. 77].
- A* is admissible [Pearl 1984, Theorem 2, p. 78].
- If A^*_2 is more informed than A^*_1, then A^*_2 dominates A^*_1 [Pearl 1984, Theorem 7, p. 81].

Much of this early work relied on the heuristic being consistent. But as Pearl [1984, p. 111] notes: "The property of monotonicity was introduced by Pohl [1977] to replace that of consistency. Surprisingly, the equivalence of the two have not been previously noted in the literature." These theorems and monotonicity provide a first attempt at a mathematical foundation for heuristic search.

These theorems suggest that A* is robust and that heuristics that are more informed lead to more efficient searches. What is missing is how to relate the effort and accuracy in computing the heuristic to its computational benefit.

4 Algorithms, and Numerical Methods, as a model

In AI search is a weak but universal method. To make it efficient we have to give it powerful guidance mechanisms. A well developed theory of search exists in numerical analysis for finding the roots of equations. It represented to me a possible conceptual model for heuristic search.

There are diverse algorithms to solve the root search problem. A sample of methods could include bisection search, Monte Carlo sampling, and Newton-Raphson. Bisection search for root finding is robust. Newton-Raphson (NR) for root finding is efficient. Monte-Carlo root finding is highly robust and very inefficient. How to decide what to use involves efficiency and error concerns. Often efficient methods of root finding require a function be well behaved. So NR converges quadratically, but requires differentiability. Bisection converges linearly but requires continuity. Monte-Carlo in most circumstances works very slowly, but works on discontinuous and non-differentiable functions

In numerical methods error analysis is critical to understanding a method's utility and its efficiency. Also in algorithmic methods one needs to use adversaries to stress test robustness and efficiency. These techniques can be applied to heuristic search algorithms.

Heuristic search theory can be investigated analogously to the theory of root finding. To subsume the various heuristic algorithms, I formulated the following generalization of the search function f, as a linear combination of g and h. Furthermore this combination could be weighted dynamically.

The node selection function is : $f(x) = (1-w(x))g(x)+w(x)h(x)$, $0 \leq w(x) \leq 1$.

Moore maze/path: $w = 0$, edge costs are 1

Dijkstra: $w = 0$, edge costs are 1

Michie-Doran GT: $w = 1$

HNR A*: $w = 0.5$, h is admissible

This generalization is also adopted in Pearl [1984], Section 3.2.1. Once you have this generalization, one can ask questions that are similar to those studied by numerical analysts. How reliably the heuristic function estimates effort, leads to a notion of error. This error then effects the convergence rate to a solution. This question was first taken up by Pohl [1970a] and later by Gaschnig [1979]. Pearl's results as summarized in [Pearl 1984, Chapters 6–7], provide a detailed treatment of the effects of error on search.

5 Variations: Weighted and Bidirectional search

5.1 Weighted dynamic search

The four standard weightings for HPA were all static, for example $w = 1/2$ for A*. Dynamic weighting is proposed in [Pohl 1973], where $w(x)$ is dependent on the character of the state. This was inspired by the observation that accuracy of heuristics improved as the search neared its goal. An admissible heuristic is an underestimate. Dynamic weighting can overestimate (see discussion in [Pearl 1984, Chapter 7]) and be not admissible. This technique remains controversial and is underutilized and not extensively researched.

5.2 Bidirectional Search

Bidirectional search was originally proposed in optimzation [Nicholson 1966] to improve on unidirectional shortest path algorithms. These implementations presumed a naive termination condition - namely that when an intersection of two paths occurred the resulting path was optimal. Finding this an error and implementing the correct terminating condition led me to consider both practically and theoretically how more efficient bidirectional search was.

Bidirectional search is an attractive approach for several reasons. Searches are normally combinatorially explosive in their depth. Two searches of half the depth ideally save exponential time and space. When run on a parallel architecture they can essentially be done simultaneously. Furthermore, this leads to a natural recursive process of further divide and conquer searches.

The cardinality comparison rule tells us to expand in the sparser hemi-tree and can be important in improving these searches. There is the following intuitive justification as to why such a rule makes more progress than simple alternation. Consider the problem of picking a black ball out of either of two urns. Each urn contains a single black ball and some white balls. The probability of finding a black ball is 1/n where n is the number of balls in the urn. In finding a next black ball

it is best to pick from the urn with fewest balls. Think of the urn as the collection of open nodes and selection as finding the next node along an optimum path. This leads to the cardinality comparison rule.

Bidirectional search works well for the standard graph shortest path problem. Here, bidirectional search, exclusive of memory constraints, dominates unidirectional search when the metric is nodes expanded. But when search is guided by highly selected heuristics there can be a "wandering in the desert" problem when the two frontiers do not meet in the middle.

To address this problem, I first proposed a parallel computation of all front-to-front node values in 1975. Two of my student's implemented and tested this method [De Champeaux and Sint 1977]. It had some important theoretical advantages, such as retaining admissibility, but it used an expensive front-to-front computation that involved the square of the nodes in the open set. By only looking at nodes expanded as a measure of efficiency it was misleading as to its true computational effort [Davis et al. 1984].

6 Judea and Heuristics

In *Heuristics* [Pearl 1984], Judea Pearl insightfully presents the work on heuristic search from 1950-1984. He contextualized and showed it as a mature theory. The book is the defining third generation document that immensely broadens and deepens the mathematical character of the field.

Pearl's book is very important in emphasizing the need for a sophisticated view of computational efficiency. It summarized, systematized and extended the theory of heuristic search. It extensively analyzed A* using both worst-case analysis and expected case analysis. It looked at the case of using nonadmissible heuristics.

Chapter 4 gives pointers to how heuristics can be discovered that will work on difficult problems including NP-Complete problems. An example would be the "out-of-place heuristic" in the sliding blocks puzzles. Here a person would be in one move allowed to swap an out-of-place tile to its final location. The number of out-of-place tiles would be a lower bound on a solution length and easy to compute. A more sophisticated heuristic occurs with respect to the traveling salesman problem (TSP) where the minimum spanning tree is a relaxed constraint version for visiting all nodes in a graph and is $nlog(n)$ in its computation.

In Pearl [1984] Chapter 6, we have a summary of results on complexity versus the precision of the heuristic. Here the work of Pearl and his students Rina Decter and Nam Huyn is presented. Pearl, Dechter, and Huyn [Dechter and Pearl 1985], [Huyn et al. 1980] developed the theory of optimality for A* for the expected case as well as the worst case.

"Theorem 1 [Huyn et al. 1980] For any error distribution, if A^*_2 is stochastically more informed than A^*_1, then A^*_2 is stochastically more efficient than A^*_1 [Pearl 1984, p. 177]."

This leads to a result by Pearl [1983] that "the exponential relationship estab-

lished in this section implies that precision-complexity exchange for A* is fairly 'inelastic'."Namely, unless error in the heuristic is better than logarithmic, search branching rates remain exponential.

In Pearl [1984, Chapter 7], there are results on search without admissibility. He provides a formal probabilistic framework to analyze questions of non-admissible heuristics and search efficiency. These results provide one view of dynamic-weighting search.

The impact of the Pearl monograph was transformative. It heralded a third wave of sophistication in the theory of heuristic search. First we had inventive pioneering search algorithms without a theoretical basis, such as GPS or GT. Second we had the Nilsson [1980] work that formulated a basic theory of A* and my work that formulated a very rough theory of efficiency. Third, in 1984, we had Pearl adding depth and breadth to this theory and embedding it in a sophisticated probabilistic reality.

7 D-nodes, NP, LPA*

7.1 What Pearl inspired

Pearl's synthesis recharged my batteries and led me with several students and colleagues to further examine these problems and algorithms.

NP [Garey and Johnson 1979] complexity is the sine quo non for testing search on known hard problems. The Hamiltonian problem was already known in 1980 as NP, but not sliding tile puzzles. I conjectured that the generalized sliding tile problem was NP in private communications to Ratner and Warmuth [1986]. They in turn produced a remarkable proof that it indeed was NP. This is important in the sense that it puts what some have called "the drosophila of AI" on firm footing as a proper exemplar. In effect it furthers the Nilsson, Pohl, Pearl agenda of having a mathematical foundation for search.

7.2 LPA* search

In work with Ratner [Ratner and Pohl 1986], we proposed Local Path A* (LPA*), an algorithm that combines an initial efficient approximation algorithm with local A* improvements. Such an approach retains the computational efficiency of finding the initial solution by carefully constraining improvements to a small computational cost.

The idea behind these algorithms is to combine a fast approximation algorithm with a search method. This idea was first suggested by S. Lin [Lin, 1965], when he used it to find an effective algorithm for the Traveling-Salesman problem (TSP). Our goal was to develop a problem independent approximation method and combine it with search.

An advantage of approximation algorithms is that they execute in polynomial time, where many other algorithms have no such upper bound. The test domain is the 15 puzzle and the approximation algorithm is based on macro-problem-solving

[Korf 1985a]. The empirical results, which come from a test on a standard set of 50 problems [Politowski and Pohl, 1984], show that the algorithms outperform other then published methods within stated time limits.

In order to bound the effort of local search by a constant, each local search will have the start and goal nodes reside on the path, with the distance between them bounded by d_{max}, a constant independent of n and the nodes. Then we will apply A * with admissible heuristics to find a shortest path between the two nodes. The above two conditions generally guarantee that each A * use requires less than some constant time. More precisely, if the branching degrees of all the nodes in G are bounded by a constant c which is independent of n then A * will generate at most $c(c-1)^{d_{max}-1}$ nodes.

Theoretically $c(c-1)^{d_{max}-1}$ is a constant, but it can be very large. Nevertheless, most heuristics prune most of the nodes [Pearl 1984]. The fact that not many nodes are generated, is supported by experiments.

The results in [Ratner and Pohl 1986] demonstrate the effectiveness of using LPA * . When applicable, this algorithm achieves a good solution with small execution time. This method require an approximation algorithm as a starting point. Typically, when one has a heuristic function, one has adequate knowledge about the problem to be able to construct an approximation algorithm. Therefore, this method should be preferred in most cases to earlier heuristic search algorithms.

7.3 D-node Bidirectional search

Bidirectional heuristic search is potentially more efficient than unidirectional heuristic search. A basic difficulty is that the two search trees do not meet in the middle. This can result in two unidirectional searches and poorer performance. To work around this George Politowski and I implemented a retargeted bidirectional search.

De Champeaux describes a Bidirectional, Heuristic Front-to-Front Algorithm (BHFFA) [De Champeaux, Sint 1977] which is intended to remedy the "meet in the middle" problem. Data is included from a set of sample problems corresponding to those of [Pohl 1971]. The data shows that BHFFA found shorter paths and expanded fewer nodes than Pohl's bidirectional algorithm. However, there are several problems with the data. One is that most of the problems are too easy to constitute a representative sample of the 15-puzzle state space, and this may bias the results. Another is that the overall computational cost of the BHFFA is not adequately measured, although it is of critical importance in evaluating or selecting a search algorithm. A third problem concerns admissibility. Although the algorithm as formally presented is admissible, the heuristics, weightings, termination condition, and pruning involved in the implemented version all violate admissibility. This makes it difficult to determine whether the results which were obtained are a product of the algorithm itself or of the particular implementation. It is also difficult to be sure that the results would hold in the context of admissible search.

The main problem in bidirectional heuristic search is to make the two partial

paths meet in the middle. The problem with Pohl's bidirectional algorithm is that each search tree is 'aimed' at the root of the opposite tree. What is needed is some way of aiming at the front of the opposite tree rather than at its root. There are two advantages to this. First, there is a better chance of meeting the opposite front if you are aiming at it. Second, for most heuristics the aim is better when the target is closer. However, aiming at a front rather than a single node is somewhat troublesome since the heuristic function is only designed to estimate the distance between two nodes. One way to overcome this difficulty is to choose from each front a representative node which will be used as a target for nodes in the opposite tree. We call such nodes d-nodes.

Consider a partially developed search tree. The growth of the tree is guided by the heuristic function used in the search, and thus the whole tree is inclined, at least to some degree, towards the goal. This means that one can expect that on the average those nodes furthest from the root will also be closest to the goal. These nodes are the best candidates for the target to be aimed at from the opposite tree. In particular, the very farthest node out from the root should be the one chosen. D-node selection based on this criterion costs only one comparison per node generated.

We incorporated this idea into a bidirectional version of HPA in the following fashion:

1. Let the root node be the initial d-node in each tree.

2. Advance the search n moves in either the forward or backward direction, aiming at the d-node in the opposite tree. At the same time, keep track of the furthest node out, i.e. the one with the highest g value.

3. After n moves, if the g value of the furthest node out is greater than the g value of the last d-node in this tree, then the furthest node out becomes the new d-node. Each time this occurs, all of the nodes in the opposite front should be re-aimed at the new d-node.

4. Repeat steps 2 and 3 in the opposite direction.

The above algorithm does not specify a value for n. Sufficient analysis may enable one to choose a good value based on other search parameters such as branching rate, quality of heuristic, etc. Otherwise, an empirical choice can be made on the basis of some sample problems. In our work good results were obtained with values of n ranging from 25 to 125.

It is instructive to consider what happens when n is too large or too small, because it provides insight into the behavior of the d-node algorithm. A value of n which is too large will lead to performance similar to unidirectional search. This is not surprising since for a sufficiently large n, a path will be found unidirectionally, before any reversal occurs. A value of n which is too small will lead to poor performance

in two respects. First, the runtime will be high because the overhead to re-aim the opposite tree is incurred too often. Second, the path quality will be lower (i.e. the The evaluation function used by the d-node search algorithm is the same as that used by HPA, namely $f = (1 - w)g + w * h$, except that h is now the heuristic estimate of the distance from a particular node to the d-node of the opposite tree. This is in contrast to the original bidirectional search algorithm, where h estimates the distance to the root of the opposite tree, and to unidirectional heuristic search, where h estimates the distance to the goal. The d-node algorithm's aim is to perform well for a variety of heuristics and over a range of w values.

The exponential nature of the problem space makes it highly probable that randomly generated puzzles will be relatively hard, i.e. their shortest solution paths will be relatively long with respect to the diameter of the state space. The four functions used to compute h are listed below. These functions were originally developed by Doran and Michie [Doran, Michie 1966], and they are the same functions as those used by Pohl and de Champeaux.

1. h = P
2. h = P +20*R
3. h = S
4. h = S +20*R

The three basic terms P, S, and R have the following definitions.

1. where P is the sum for all tile i of Manhattan distance between the position of tile i in the current state and in the goal.
2. S, a relationship between tile and the blank square defined in [Doran, Michie 1966].
3. R is the number of reversals in the current state with respect to goal. For example if tile 2 is first and tile 1 is next, this is a reversal.

Finally, the w values which we used were 0.5, 0.75, and 1.0. This covers the entire 'interesting' range from w = 0.5, which will result in admissible search with a suitable heuristic, to w = 1.0, which is pure heuristic search.

The detailed results of our test of the d-node algorithm are found in [Politowski 1986]. The most significant result is that the d-node method dominates both previously published bidirectional techniques, regardless of heuristic or weighting. In comparison to de Champeaux's BHFFA, the d-node method is typically 10 to 20 times faster. This is chiefly because the front-to-front calculations required by BHFFA are computationally expensive, even though the number of nodes expanded is roughly comparable for both methods. In comparison to Pohl's bidirectional algorithm, the d-node method typically solves far more problems, and when solving the same problems it expands approximately half as many nodes.

8 Next Steps

Improving problem decomposition remains an important underutilized strategy within heuristic search. Divide and conquer remains a central instrument of intelligent deduction in many formats and arenas. Here the bidirectional search theme and the LPA* search are instances of naive but effective first steps. Planning is a further manifestation of divide and conquer. Intuitively planning amounts to a good choice of lemma's when attempting to construct a difficult proof. Korf's [Korf 1985a, Korf 1985b] macro operators can be seen as a first step, or the Pohl-Politowski d-node selection criteria as a an automated attempt at problem decomposition.

Expanding problem selection to hard domains is vital to demonstrating the relevance of heuristic search. Historically these techniques were developed on puzzle domains. Here Korf's [Korf, Zhang 2000] recent work in applying search to genomics problems is welcome. Computational genomics is an unambiguously complex and non-trivial domain that almost certainly requires heuristics search.

Finally, what seems under exploited is the use of a theory of error linked to efficiency. Here my early work and the experimental work of Gaschnig contributed to the deeper studies of Pearl and Davis [1990]. Questions of worst case analysis and average case complexity and its relation to complexity theory while followed up by Pearl, and by Chenoweth and Davis [1992], is only a beginning. A deeper theory that applies both adversary techniques and metric space analysis is needed.

References

Chenoweth, S., and Davis, H. (1992). New Approaches for Understanding the Asymptotic Complexity of A* Tree Searching. *Annals of Mathematics and AI 5*, 133–162.

Davis, H. (1990). Cost-error Relationships in A* Tree Searching. *Journal of the ACM 37*, 195–199.

Davis, H., Pollack, R., and Sudkamp, T. (1984). Toward a Better Understanding of Bidirectional Search. *Proceedings AAAI*, pp. 68–72.

De Champeaux, D., and Sint, L. (1977). An improved bi-directional heuristic search algorithm. *Journal of the ACM 24*, 177–191.

Dechter, R., and Pearl, J. (1985). Generalized Best-First Search Strategies and the Optimality of A*. *Journal of the ACM 32*, 505–536.

Dijkstra, E. (1959). A Note on Two Problems in Connection with Graphs *Numerische Mathematik 1*, 269–271, 1959.

Doran, J., and Michie, D. (1966). Experiments with the Graph Traverser Program. *Proc.of the Royal Society of London, 294A*, 235–259.

Field, R., Mohyeldin-Said, K., and Pohl, I. (1984). An Investigation of Dynamic Weighting in Heuristic Search, *Proc. 6th ECAI*, pp. 277–278.

Gaschnig, J. (1979). Performance Measurement and Analysis of Certain Search Algorithms. *Ph.D. Dissertation CMU CS 79-124*.

Hart, P. E., Nilsson, N., and Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimal Cost Paths Search Reconsidered. *IEEE Trans. Systems Science and Cybernetics SSC-4 2*, 100–07.

Huyn, N., Dechter, R., and Pearl, J. (1980). Probabilistic analysis of the complexity of A*. *Artificial Intelligence:15*, 241–254.

Kaindl, H., and Kainz, G. (1997). Bidirectional Heuristic Search Reconsidered. *Journal of Artificial Intelligence Research 7*, 283–317.

Knuth, D. E. (1994). Leaper graphs. *Mathematical Gazette 78*, 274–297.

Korf, R., E. (1985a). *Learning to Solve problems by Searching Macro-Operators*. Pitman.

Korf, R.,E. (1985b). Iterative Deepening A*. *Proceedings 9th IJCAI, vol. 2*, pp. 1034–1035.

Korf, R.,E., and Zhang, W. (2000). Divide and Conquer Frontier Search Applied to Optimal Sequence Alignment. *AAAI-00*, pp. 910–916.

Korf, R. E. (2004). Best-First Frontier Search with Delayed Duplicate Detection. *AAAI-04*, pp. 650–657.

Lin, S. (1965). Computer solutions of the traveling salesman problem. *Bell Systems Tech. J. 44(10)*, 2245–2269.

Newell, A., and Simon, H. (1972). *Human Problem Solving*. Prentice Hall.

Nilsson, N. (1980). *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga.

Pearl, J. (1983). Knowledge versus search: A quantitative analysis using A*. *Artificial Intelligence:20*, 1–13.

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, Massachusetts: Addison-Wesley.

Pohl, I. (1967). A method for finding Hamilton paths and knight's tours. *CACM 10*, 446–449.

Pohl, I. (1970a). First Results on the Effect of Error in Heuristic Search. In B. Meltzer and D. Michie (Eds.), *Machine Intelligence 5*, pp. 219–236. Edinburgh University Press.

Pohl, I. (1970b). Heuristic Search Viewed as Path Finding in A Graph. *Artificial Intelligence 1*, 193–204.

Pohl, I. (1971). Bi-directional search. In B. Meltzer and D. Michie (Eds.), *Machine Intelligence 6*, pp. 127–140. Edinburgh University Press.

Pohl, I. (1973). The Avoidance of (Relative) Catastrophe, Heuristic Competence, Genuine Dynamic Weighting and Computational Issues in Heuristic Problem Solving. *IJCAII 3*, pp. 20–23.

Pohl, I. (1977). Practical and Theoretical Considerations in Heuristic Search Algorithms, In E. Elcock and D. Michie (Eds.), *Machine Intelligence 8*, pp. 55–72. New York: Wiley.

Ratner, D., and Pohl, I. (1986). Joint and LPA*: Combination of Approximation and Search. *Proceedings AAAI-86, vol. 1*, pp. 173–177.

Ratner, D., and Warmuth, M. (1986). Finding a Shortest Solution for the $N \times N$ Extension of the 15-Puzzle is Intractable. *Proceedings AAAI-86, vol. 1*, pp. 168–172.

Samuel, A. (1959). Some Studies in Machine Learning Using Checkers. *IBM Research Journal of Research and Development 3*, 211–229.

Slagle, J. R. (1963). A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus. *Journal of the ACM 10*, 507–520.

Tutte, W. T. (1946). On Hamiltonian Circuits. *J. London Math. Soc. 21*, 98–101.