CHAPTER 4

Heuristic Search Algorithms

4.1 HEURISTIC SEARCH AND SSP MDPS

The methods we explored in the previous chapter have a serious practical drawback — the amount of memory they require is proportional to the MDP's state space size. Ultimately, the reason for this is that they compute a complete policy, i.e., an optimal action for each state of an MDP.

For factored MDPs, the size of the state space (and hence of a complete policy) is exponential in the MDP description size, limiting the scalability of algorithms such as VI and PI. For instance, a factored MDP with 50 state variables has at least 2^{50} states. Even if each state could be represented with 50 bits, the total amount of memory required to represent such an MDP's complete policy (let alone the time to compute it!) would be astronomical. In the meantime, for many probabilistic planning problems we have two pieces of information that, as we will see in this chapter, can help us drastically reduce the amount of computational resources for solving them.

The first of these pieces of information is the MDP's initial state, denoted as s_0. In the Mars rover example, s_0 may be the configuration of the rover's systems (the amount of energy, the positions of its manipulators, etc.) and its location at the start of the mission. While computing a policy that reaches the goal from states other than s_0 may be useful, for many of the states doing so is unnecessary since they are not reachable from the initial state. In other words, if the initial state is known, it makes sense to compute only a *partial policy* π_{s_0} *closed w.r.t.* s_0.

Definition 4.1 Partial Policy. A stationary deterministic policy $\pi : \mathcal{S}' \to \mathcal{A}$ is *partial* if its domain is some set $\mathcal{S}' \subseteq \mathcal{S}$.

Definition 4.2 Policy Closed with Respect to State s. A stationary deterministic partial policy $\pi_s : \mathcal{S}' \to \mathcal{A}$ is *closed with respect to state* s if any state s' reachable by π_s from s is contained in $\mathcal{S}'$, the domain of π_s.

Put differently, a policy closed w.r.t. a state s must specify an action for any state s' that can be reached via that policy from s. Such a policy can be viewed as a restriction of a complete policy π to a set of states $\mathcal{S}'$ iteratively constructed as follows:

- Let $\mathcal{S}' = \{s\}$.
- Add to $\mathcal{S}'$ all the states that π can reach from the states in $\mathcal{S}'$ in one time step.

- Repeat the above step until $\mathcal{S}'$ does not change.

For many MDPs, a policy closed w.r.t. the initial state may exclude large parts of the state space, thereby requiring less memory to store. In fact, $\pi^*_{s_0}$, an *optimal* such policy, is typically even more compact, since its domain $\mathcal{S}'$ usually does not include all the states that are theoretically reachable from s_0 in *some* way. The advantages of computing a closed partial policy have prompted research into efficient solution techniques for a variant of stochastic shortest path problems where the initial state is assumed to be known. Below, we adapt the definition of SSP MDPs and concepts relevant to it to account for the knowledge of the initial state.

4.2 FIND-AND-REVISE: A SCHEMA FOR HEURISTIC SEARCH

In the previous section we gave a high-level idea of why heuristic search can be efficient, but have not explained how exactly we can employ heuristics for solving MDPs. In the remainder of this chapter we will see many algorithms for doing so, and all of them are instances of a general schema called FIND-and-REVISE (Algorithm 4.1) [31]. To analyze it, we need several new concepts.

Throughout the analysis, we will view the connectivity structure of an MDP M's state space as a *directed hypergraph* $G_{\mathcal{S}}$ which we call the *connectivity graph of* M. A directed hypergraph generalizes the notion of a regular graph by allowing each *hyperedge*, or *k-connector*, to have one source but several destinations. In the case of an MDP, the corresponding hypergraph $G_{\mathcal{S}}$ has $\mathcal{S}$ as the set of vertices, and for each state s and action a pair has a k-connector whose source is s and whose destinations are all states s' s.t. $\mathcal{T}(s, a, s') > 0$. In other words, it has a k-connector for linking each state via an action to the state's possible successors under that action. We will need several notions based on the hypergraph concept.

Definition 4.5 Reachability. A state s_n is *reachable* from s_1 in $G_{\mathcal{S}}$ if there is a sequence of states and actions $s_1, a_1, s_2, \ldots, s_{n-1}, a_{n-1}, s_n$, where for each i, $1 \leq i \leq n-1$, the node for s_i is the source of the k-connector for action a_i and s_{i+1} is one of its destinations.

Definition 4.6 The Transition Graph of an MDP Rooted at a State. *The transition graph of an MDP rooted at state* s is G_s, a subgraph of the MDP's connectivity graph $G_{\mathcal{S}}$. Its vertices are s and only those states s' that are reachable from s in $G_{\mathcal{S}}$. Its hyperedges are only those k-connectors that originate at s or at some state reachable from s in $G_{\mathcal{S}}$.

When dealing with MDPs with an initial state, we will mostly refer to the transition graph G_{s_0} rooted at the initial state. This hypergraph includes only states reachable via some sequence of action outcomes from s_0. Historically, MDP transition graphs are also known as *AND-OR graphs*.

Definition 4.7 The Transition Graph of a Policy. *The transition graph of a partial deterministic Markovian policy* $\pi_s : \mathcal{S}' \to A$ is a subgraph of the MDP's connectivity graph $G_{\mathcal{S}}$ that contains only the states in $\mathcal{S}'$ and, for each state $s \in \mathcal{S}'$, only the k-connector for the action a s.t. $\pi(s) = a$.

Definition 4.8 The Greedy Graph of a Value Function Rooted at a State. *The greedy graph* G_s^V *of value function* V *rooted at state* s is the union of transition graphs of all policies π_s^V greedy w.r.t. V and closed w.r.t. s.

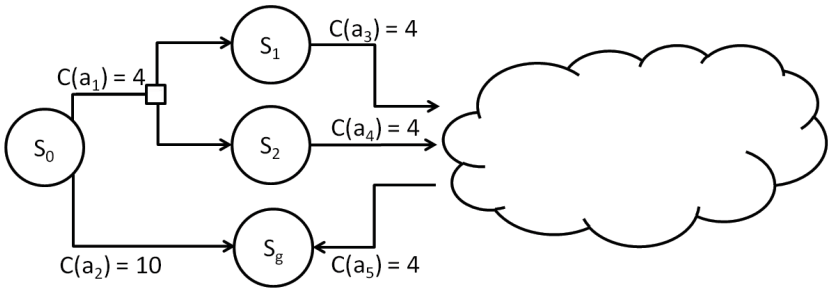

Figure 4.1: MDP showing a possible impact of a heuristic on the efficiency of policy computation via FIND-and-REVISE.

That is, G_s^V contains all states that can be reached via *some* π_s^V from s. As with general transition graphs, we will mostly be interested in greedy graphs of value functions rooted at the initial state s_0.

As the final pieces of terminology before we proceed, recall that the residual $Res^V(s)$ (Definition 3.16) is the magnitude of change in the value of a state as a result of applying a Bellman backup to value function V. A state s is called ϵ-consistent w.r.t. V if $Res^V(s) < \epsilon$ (Definition 3.17) and ϵ-inconsistent otherwise.

Algorithm 4.1: FIND-and-REVISE

```
1 Start with a heuristic value function V ← h
2 while V's greedy graph G^V_{s0} contains a state s with Res^V(s) > ε do
3 |   FIND a state s in G^V_{s0} with Res^V(s) > ε
4 |   REVISE V(s) with a Bellman backup
5 end
6 return a π^V
```

The idea of FIND-and-REVISE is quite simple. It iteratively searches the greedy graph of the current value function for an ϵ-inconsistent state and updates the value of that state and possibly of a few others with a finite number of Bellman backups. This typically changes the greedy graph, and the cycle repeats. FIND-and-REVISE can be viewed as a Prioritized Value Iteration scheme (Section 3.5) over the MDP's transition graph rooted at s_0 — in every iteration it assigns a higher priority to every state in $G_{s_0}^V$ than to any state outside of this greedy graph. Moreover, the FIND step can use values of states within the greedy graph for further prioritization. Recall that Prioritized Value Iteration is itself a kind of Asynchronous Value Iteration (Section 3.4.4), so FIND-and-REVISE is a member of this more general class of approaches as well.

Crucially, the greedy graph that FIND-and-REVISE starts with is induced by some heuristic function h. To demonstrate the difference h can make on the number of states a FIND-and-REVISE-like algorithm may have to store, we present the following example.

Example: Consider the transition graph G_{s_0} of the SSP_{s_0} MDP in Figure 4.1. This MDP has many states, four of which (the initial state s_0, the goal state s_g, and two other states s_1 and s_2) are shown, while the rest are denoted by the cloud. Action costs are shown for a subset of the MDP's actions; assume that costs are nonnegative for actions in the cloud. At least one of the actions, a_1, has several probabilistic effects, whose probabilities do not matter for this example and are omitted. Note that $\pi^*_{s_0}$ for this MDP is unique and involves taking action a_2 from s_0 straight to the goal; thus, $V^*(s_0) = 10$. All other policies involve actions a_1, a_5, and a_3 or a_4. Therefore, the cost of reaching the goal from s_0 using them is at least $4 \cdot 3 = 12 > 10$, making these policies suboptimal.

Now, the transition graph G_{s_0} of the MDP in Figure 4.1 can be arbitrarily large, depending on the number of states in the cloud. This is the largest set of states an MDP solution algorithm may *have to* store while searching for $\pi^*_{s_0}$. Compare G_{s_0} to $G^{V^*}_{s_0}$, the greedy graph of the optimal value function. $G^{V^*}_{s_0}$ contains only the states visited by $\pi^*_{s_0}$, i.e., s_0 and s_g, as established above. This is the very smallest set of states we can hope to explore while looking for $\pi^*_{s_0}$. Finally, consider $G^h_{s_0}$ for a heuristic value function h that assigns $h(s_g) = 0$ and, for instance, $h(s_1) = h(s_2) = 7$. These values would induce $Q^h(s_0, a_1) = 4 + 7 = 11 > Q^h(s_0, a_2) = 10$, making a_2 more preferable in s_0 and thus immediately helping discover the optimal policy. Thus, $G^h_{s_0}$ consists of s_0 and s_g, and starting FIND-and-REVISE from such an h allows FIND-and-REVISE to evaluate only four states before finding $\pi^*_{s_0}$. Contrast this with an algorithm such as VI, which, even initialized with a very good h, will still necessarily visit the entire transition graph rooted at the initial state, G_{s_0}. As this example shows, FIND-and-REVISE in combination with a good heuristic can make finding $\pi^*_{s_0}$ arbitrarily more efficient than via VI or PI! □

The fact that FIND-and-REVISE may never touch some of the states, as in the above example, might seem alarming. After all, Prioritized Value Iteration algorithms in general fail to find an optimal policy if they starve some states, and FIND-and-REVISE appears to be doing exactly that. As it turns out, however, if the FIND-and-REVISE's FIND procedure is *systematic* and the heuristic function FIND-and-REVISE is using is *admissible*, FIND-and-REVISE is guaranteed to converge to an optimal solution for a sufficiently small ϵ.

Definition 4.10 Heuristic Admissibility. A heuristic h is *admissible* if for all states s in the transition graph G_{s_0}, $h(s) \leq V^*(s)$. Otherwise, the heuristic is called *inadmissible*.

Theorem 4.11 For an SSP_{s_0} MDP, if FIND-and-REVISE is initialized with an admissible heuristic, terminated when all the states in the value function's greedy graph rooted at s_0 are ϵ-consistent, and its FIND procedure is systematic, the value function computed by FIND-and-REVISE approaches the optimal value function over all states reachable from s_0 by an optimal policy as ϵ goes to 0 [31].

Theorem 4.11 contains a small caveat. As with VI, although *in the limit* a vanishingly small residual implies optimality, for finite values of ϵ FIND-and-REVISE can return a significantly suboptimal policy. Nonetheless, in practice this rarely happens. In this book we will assume the chosen ϵ to be small enough to let FIND-and-REVISE halt with an optimal policy.

We will examine several techniques for computing admissible heuristics automatically in Section 4.6, and until then will simply assume the SSP MDP heuristic we are given to be admissible, i.e., to be a lower bound on V^*. This assumption has a very important implication. Recall from Chapter 3 that Bellman backups on SSP MDPs are monotonic (Theorem 3.20); in particular, if for some V, $V \leq V^*$, then after V is updated with a Bellman backup it is still true that $V \leq V^*$. Therefore, if FIND-and-REVISE starts with an admissible heuristic, after every state value update the resulting value function is still a lower bound on V^*. In effect, FIND-and-REVISE generates a sequence of value functions that approaches V^* from below for all states visited by $\pi^*_{s_0}$. To stress this fact, we will refer to the value function V FIND-and-REVISE maintains as V_l in the rest of this chapter (l stands for "lower bound").

FIND-and-REVISE's pseudocode intentionally leaves the FIND and REVISE procedures unspecified — it is in their implementations that various heuristic search algorithms differ from each other. Their REVISE methods tend to resemble one another, as they are all based on the Bellman backup operator. Their FIND methods, however, can be vastly distinct. An obvious approach to finding ϵ-inconsistent states is via a simple systematic search strategy such as depth-first search. Indeed, depth-first search is used in this manner by several proposed FIND-and-REVISE algorithms, e.g., HDP [31] and LDFS [33]. Employing depth-first search for the purpose of identifying weakly explored states also echoes similar approaches in solving games, non-deterministic planning problems, and other related fields [33]. However, in tackling MDPs this strategy is usually outperformed by more sophisticated search methods, which we are about to explore.

4.3 LAO* AND EXTENSIONS

Historically, the first FIND-and-REVISE algorithm that could deal with arbitrary SSP_{s_0} MDPs was LAO* [105]. We start our presentation of heuristic search techniques with this algorithm as well and then describe its derivatives.

4.3.1 LAO*

The basic approach of LAO* is to look for states with high residuals only in a restricted region of the current value function's greedy policy graph (AND-OR graph, in LAO* terminology) reachable from the initial state, called the *solution graph*. Besides the solution graph, LAO* also keeps track of a superset of the solution graph's nodes called the *envelope*, which contains all the states that have ever been part of the solution graph in the past. By updating values of states in the envelope, LAO* constructs a new greedy policy rooted at s_0 (and the solution graph corresponding to it) over the envelope's states. Sometimes, this policy, "attracted" by the low heuristic values of states in the unexplored area of state space, takes an action leading the agent to a state beyond the envelope. This causes LAO* to gradually expand and modify the envelope. At some point, LAO* ends up with a greedy policy over the envelope that never exits the envelope and that cannot be improved. This policy is optimal.

Pseudocode for LAO* is presented in Algorithm 4.2. Starting with the initial state, it gradually expands $\hat{G}_{s_0}$, the *explicit graph* of the envelope, by adding new states to it. $\hat{G}_{s_0}$ represents the connectivity structure of the states in the envelope and is a subgraph of the transition graph G_{s_0} explored by LAO* so far. Thus, at each iteration of LAO*, the current envelope is simply the set of $\hat{G}_{s_0}$'s nodes. This set can be subdivided into the set of the envelope's *fringe* states F and the set of *interior* states I. Each fringe state has at least one successor under some action that lies outside of $\hat{G}_{s_0}$. In contrast, all successors of all interior states belong to $\hat{G}_{s_0}$.

Using $\hat{G}_{s_0}$, LAO* also maintains the solution graph $\hat{G}_{s_0}^{V_l}$ — a subgraph of $\hat{G}_{s_0}$ representing the current best partial policy over the current envelope, rooted at s_0. That is, $\hat{G}_{s_0}^{V_l}$ is a graph of all states in $\hat{G}_{s_0}$ that a policy greedy w.r.t V_l can reach from the initial state.

When LAO* starts running, $\hat{G}_{s_0}$ contains only s_0, and this state is $\hat{G}_{s_0}$'s fringe state (lines 4-6 of Algorithm 4.2). In subsequent iterations, LAO* chooses a fringe state s of $\hat{G}_{s_0}$ that also belongs to the solution graph $\hat{G}_{s_0}^{V_l}$ and expands it (lines 10-14), i.e., adds all the successors of s under all possible actions to $\hat{G}_{s_0}$, except for those that are in $\hat{G}_{s_0}$ already. Such a state can be chosen arbitrarily among all fringe states reachable from s_0 in the solution graph. However, a clever way of choosing the state to expand can greatly improve LAO*'s performance. Once expanded, a fringe state becomes an internal state. Next, using Bellman backups, LAO* updates the values of the expanded state and any state of $\hat{G}_{s_0}$ from which the expanded state can be reached within $\hat{G}_{s_0}$ via a greedy policy (lines 16-17). This set of states, denoted as Z in the pseudocode, is the set of all states in the envelope whose values could have changed as a result of envelope expansion. The state value updates over Z can be performed by running either PI or VI. The pseudocode in Algorithm 4.2 uses PI; however, LAO* can also be implemented using VI, and we refer the reader to the original paper for the details of that version [105]. It is at this step that the heuristic comes into play — the heuristic initializes values of the successors of the newly expanded state. PI/VI propagates these values into the interior of $\hat{G}_{s_0}$, potentially letting them affect the current greedy policy.

Finally, LAO* rebuilds $\hat{G}_{s_0}^{V_l}$ (line 18), the solution graph rooted at s_0, which could have changed because V_l was updated in the previous step. The algorithm then proceeds to the next

Algorithm 4.2: LAO*

```
1  // Initialize V_l, the fringe set F, the interior set I, the explicit graph of the envelope Ĝ_{s0},
2  // and the solution graph Ĝ_{s0}^{V_l} over states in the explicit graph.
3  V_l ← h
4  F ← {s0}
5  I ← ∅
6  Ĝ_{s0} ← {nodes: I ∪ F, hyperedges: ∅}
7  Ĝ_{s0}^{V_l} ← {nodes: {s0}, hyperedges: ∅}
8  while F ∩ Ĝ_{s0}^{V_l} has some non-goal states do
9      // Expand a fringe state of the best partial policy
10     s ← some non-goal state in F ∩ Ĝ_{s0}^{V_l}
11     F ← F \ {s}
12     F ← F ∪ {all successors s' of s under all actions, s' ∉ I}
13     I ← I ∪ {s}
14     Ĝ_{s0} ← {nodes: I ∪ F, hyperedges: all actions in all states of I}
15     // Update state values and mark greedy actions
16     Z ← {s and all states in Ĝ_{s0} from which s can be reached via the current greedy policy}
17     Run PI on Z until convergence to determine the current greedy actions in all states in Z
18     Rebuild Ĝ_{s0}^{V_l} over states in Ĝ_{s0}
19 end
20 return π_{s0}^{V_l}
```

iteration. This process continues until $\hat{G}_{s_0}^{V_l}$ has no more nonterminal fringe states, i.e., states that are not goals.

Viewed in terms of the FIND-and-REVISE framework, LAO* in essence lazily builds the transition and the greedy graph. In each iteration, it adds to both graphs' partial representations $\hat{G}_{s_0}$ and $\hat{G}_{s_0}^{V_l}$, respectively, by expanding a fringe node. LAO* then updates all the states of $\hat{G}_{s_0}$ whose values the expansion could have possibly changed, modifies $\hat{G}_{s_0}^{V_l}$ if necessary, and so on. Identifying the states whose values could have changed corresponds to the FIND-and-REVISE's FIND procedure, since the residuals at these states are likely to be greater than ϵ. Updating these states' values corresponds to the REVISE step.

The readers familiar with heuristic search in regular graphs, as exemplified by the A* algorithm [106], may have noticed some similarity in the operation of A* and LAO*. Both algorithms gradually construct a partial solution. A* does this in a regular graph, and its partial solutions are regions of the graph corresponding to *partial linear plans* originating at s_0. LAO* constructs a solution in a hypergraph, not in a regular graph. Accordingly, the partial solutions it deals with are regions of the hypergraph corresponding to *partial policies* rooted at s_0. Indeed, LAO* is a generalization of A*'s heuristic search idea to probabilistic planning problems. Each of these algorithms employs

a *heuristic* to guide the modification of the current partial policy/plan, i.e., to *search* the space of solutions.

4.4 RTDP AND EXTENSIONS

An alternative to the LAO*-style gradual, systematic expansion of the solution graph is a scheme called Real-Time Dynamic Programming (RTDP) [12]. The motivation for RTDP comes from scenarios where an agent (e.g., a robot) is acting in the real world and only occasionally has time for planning between its actions. Such an agent needs a (preferably distributed) algorithm capable of quickly exploring various areas of the state space and coming up with a reasonable action to execute in the agent's current state before planning time has run out. Thanks to its sampling-based nature, RTDP satisfies these requirements, and improvements upon it that we will also examine endow the basic approach with other useful properties.

4.4.1 RTDP

At a high level, RTDP-based algorithms operate by simulating the current greedy policy to sample "paths," or *trajectories*, through the state space, and performing Bellman backups only on the states in those trajectories. These updates change the greedy policy and make way for further state value improvements.

The process of sampling a trajectory is called a *trial*. As shown in Algorithm 4.3, each trial consists of repeatedly selecting a greedy action a_{best} in the current state s (line 15), performing a Bellman backup on the value of s (line 16), and transitioning to a successor of s under a_{best} (line 17). A heuristic plays the same role in RTDP as in LAO* — it provides initial state values in order to guide action selection during early state space exploration.

Each sampling of a successor during a trial that does not result in a termination of the trial corresponds to a FIND operation in FIND-and-REVISE, as it identifies a possibly ϵ-inconsistent state to update next. Each Bellman backup maps to a REVISE instance.

The original RTDP version [12] has two related weaknesses, the main one being the lack of a principled termination condition. Although RTDP is guaranteed to converge asymptotically to V^* *over the states in the domain of an optimal policy* $\pi^*_{s_0}$, it does not provide any mechanisms to detect when it gets near the optimal value function or policy. The lack of a stopping criterion, although unfortunate, is not surprising. RTDP was designed for operating under time pressure, and would almost never have the luxury of planning for long enough to arrive at an optimal policy. In these circumstances, a convergence detection condition is not necessary.

Algorithm 4.3: RTDP

```
1  RTDP(s_0)
2  begin
3      V_l ← h
4      while there is time left do
5          TRIAL(s_0)
6      end
7      return π*_{s_0}
8  end
9
10
11 TRIAL(s_0)
12 begin
13     s ← s_0
14     while s ∉ G do
15         a_best ← argmin_{a∈A} Q^{V_l}(s, a)
16         V_l(s) ← Q^{V_l}(s, a_best)
17         s ← execute action a_best in s
18     end
19 end
```

The lack of convergence detection leads to RTDP's other drawback. As RTDP runs longer, V_l at many states starts to converge. Visiting these states again and again becomes a waste of resources, yet this is what RTDP keeps doing because it has no way of detecting convergence. An extension of RTDP we will look at next addresses both of these problems by endowing RTDP with a method of recognizing proximity to the optimal value function over relevant states.