E Search Algorithms

A *search problem* is concerned with finding an appropriate sequence of actions to maximize obtained reward over subsequent deterministic transitions. Search problems are Markov decision processes (covered in part II) with deterministic transition functions. Some well known search problems include sliding tile puzzles, the Rubik's Cube, Sokoban, and finding the shortest path to a destination.

E.1 Search Problems

In a search problem, we choose action a_t at time t based on observing state s_t, and then receive a reward r_t. The *action space* $\mathcal{A}$ is the set of possible actions, and the *state space* $\mathcal{S}$ is the set of possible states. Some of the algorithms assume these sets are finite, but this is not required in general. The state evolves deterministically and depends only on the current state and action taken. We use $\mathcal{A}(s)$ to denote the set of valid actions from state s. When there are no valid actions, the state is considered to be *absorbing* and yields zero reward for all future timesteps. Goal states, for example, are typically absorbing.

The deterministic state transition function $T(s, a)$ gives the successor state s'. The reward function $R(s, a)$ gives the reward received when executing action a from state s. Search problems typically do not include a discount factor γ that penalizes future rewards. The objective is to choose a sequence of actions that maximizes the sum of rewards or *return*. Algorithm E.1 provides a data structure for representing search problems.

```
struct Search
    𝒮  # state space
    𝒜  # valid action function
    T  # transition function
    R  # reward function
end
```

Algorithm E.1. The search problem data structure.

E.2 Search Graphs

A search problem with finite state and action spaces can be represented as a *search graph*. The nodes correspond to states, and edges correspond to transitions between states. Associated with each edge from a source to a destination state is both an action that results in that state transition and the expected reward when taking that action from the source state. Figure E.1 depicts a subset of such a search graph for a 3×3 sliding tile puzzle.

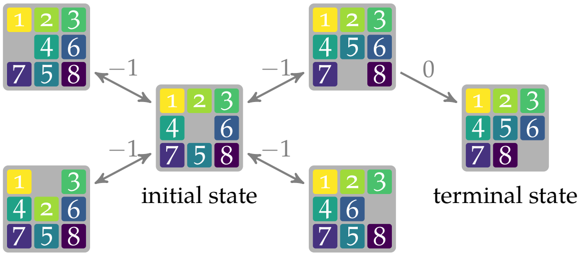


Figure E.1. A few states in a sliding tile puzzle visualized as a graph. Two transitions can be taken from the initial state to arrive at the terminal solution state. The numbers on edges represent rewards.

Many graph search algorithms conduct a search from an initial state and fan out from there. In so doing, these algorithms trace out a *search tree*. The initial state is the root node, and any time we transition from s to s' during search, an edge from s to a new node s' is added to the search tree. A search tree for the same sliding tile puzzle is shown in figure E.2.

E.3 Forward Search

Perhaps the simplest graph search algorithm is *forward search* (algorithm E.2), which determines the best action to take from an initial state s by looking at all possible action-state transitions up to a depth (or horizon) d. At depth d, the algorithm uses an estimate of the value of the state $U(s)$.[1] The algorithm calls

[1] The approximate value functions in this chapter are expected to return 0 when in a state with no available actions.

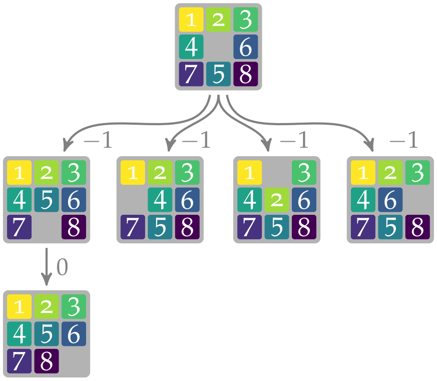


Figure E.2. The graph for the 3×3 sliding tile puzzle in figure E.1 can be represented as a tree search problem. Search begins at the root node and flows down the tree. In this case, a path can be traversed to the desired terminal state.

itself recursively in a depth-first manner, resulting in a search tree and returning a tuple with an optimal action a and its finite-horizon expected value u.

```
function forward_search(𝒫::Search, s, d, U)
    𝒜, T, R = 𝒫.𝒜(s), 𝒫.T, 𝒫.R
    if isempty(𝒜) || d ≤ 0
        return (a=nothing, u=U(s))
    end
    best = (a=nothing, u=-Inf)
    for a in 𝒜
        s′ = T(s,a)
        u = R(s,a) + forward_search(𝒫, s′, d-1, U).u
        if u > best.u
            best = (a=a, u=u)
        end
    end
    return best
end
```

Algorithm E.2. The forward search algorithm for finding an approximately optimal action for a discrete search problem 𝒫 from a current state `s`. The search is performed to depth `d`, at which point the terminal value is estimated with an approximate value function `U`. The returned named tuple consists of the best action `a` and its finite-horizon expected value `u`.

Figure E.3 shows an example search tree obtained by running forward search on a sliding tile puzzle. Depth-first search can be wasteful; all reachable states for the given depth are visited. Searching to depth d will result in a search tree with $O(|\mathcal{A}|^d)$ nodes for a problem with $|\mathcal{A}|$ actions.

E.4 Branch and Bound

A general method known as *branch and bound* (algorithm E.3) can significantly reduce computation by using domain information about the upper and lower

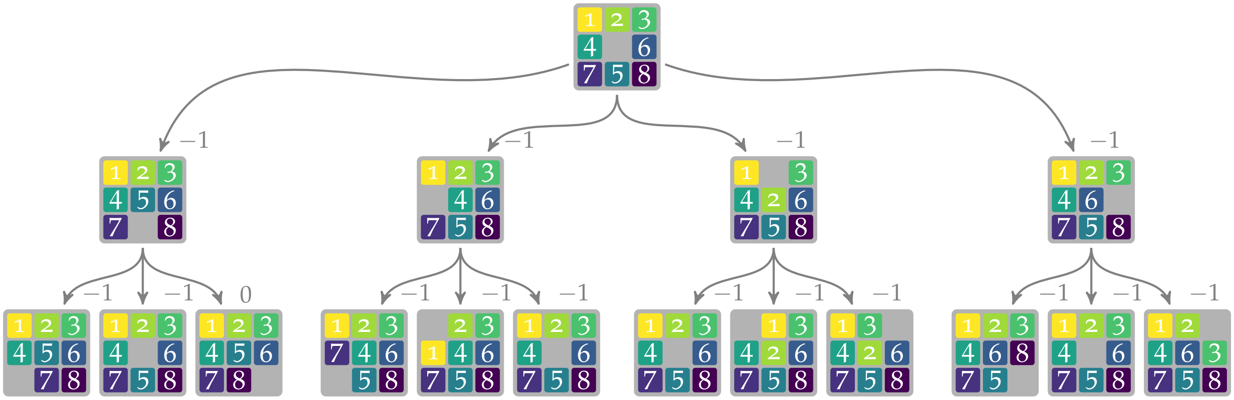


Figure E.3. A search tree arising from running forward search to depth 2 on a sliding tile puzzle. All states reachable in two steps are visited, and some are visited more than once. We find that there is one path to the terminal node. That path has a return of −1, whereas all other paths have a return of −2.

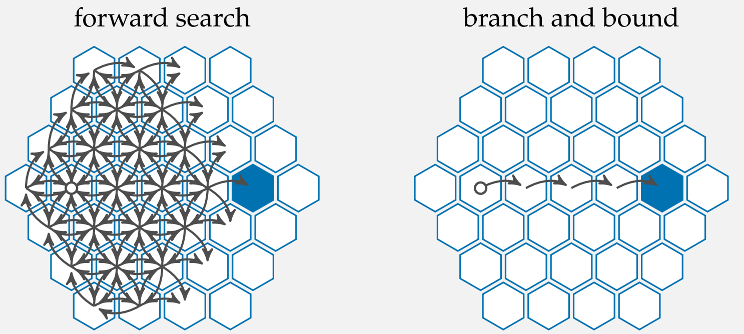
bounds on expected reward. The upper bound on the return from taking action a from state s is $\overline{Q}(s, a)$. The lower bound on the return from state s is $\underline{U}(s)$. Branch and bound follows the same procedure as depth-first-search, but iterates over the actions according to their upper bound, and only proceeds to a successor node if the best possible value it could return is lower than what has already been discovered by following an earlier action. Branch and bound search is compared to forward search in example E.1.

```
function branch_and_bound(𝒫::Search, s, d, Ulo, Qhi)
    𝒜, T, R = 𝒫.𝒜(s), 𝒫.T, 𝒫.R
    if isempty(𝒜) || d ≤ 0
        return (a=nothing, u=Ulo(s))
    end
    best = (a=nothing, u=-Inf)
    for a in sort(𝒜, by=a->Qhi(s,a), rev=true)
        if Qhi(s,a) ≤ best.u
            return best # safe to prune
        end
        u = R(s,a) + branch_and_bound(𝒫,T(s,a),d-1,Ulo,Qhi).u
        if u > best.u
            best = (a=a, u=u)
        end
    end
    return best
end
```

Algorithm E.3. The branch and bound search algorithm for finding of an approximately optimal action for a discrete search problem 𝒫 from a current state s. The search is performed to depth d with value function lower bound Ulo and action value function upper bound Qhi. The returned named tuple consists of the best action a and its finite-horizon expected value u.

Consider using branch and bound on a hex world search problem. Actions in search problems cause deterministic transitions, so unlike the hex-world MDP, we always correctly transition between neighboring tiles when the corresponding action is taken.

The circle indicates the start state. All transitions incur a reward of -1. The blue colored tile is terminal, and produces reward 5 when entered.



Above we show the search trees for both forward search and branch and bound to depth 4. For branch and bound we used a lower bound $\underline{U}(s) = -6$ and an upper bound $\overline{Q}(s, a) = 5 - \delta(T(s, a))$, where the function $\delta(s)$ is the minimum number of steps from the given state to the terminal reward state. The search tree of branch and bound is a subset of that of forward search because branch and bound can ignore portions it knows are not optimal.

Due to the upper bound, branch and bound evaluates moving right first, and because that happens to be optimal, it is able to immediately identify the optimal sequence of actions and avoid exploring other actions. If the start and goal states were reversed, the search tree would be larger. In the worst case, it can be as large as forward search.

Example E.1. A comparison of the savings that branch and bound can have over forward search. Branch and bound can be significantly more efficient with appropriate bounds.

Branch and bound is not guaranteed to reduce computation over forward search. Both approaches have the same worst-case time complexity. The efficiency of the algorithm greatly depends on the heuristic.

E.5 Dynamic Programming

Neither forward search nor branch and bound remember whether a state has been previously visited; they waste computational resources by evaluating these states multiple times. *Dynamic programming* (algorithm E.4) avoids duplicate effort by remembering when a particular subproblem has been previously solved. Dynamic programming can be applied to problems in which an optimal solution can be constructed from optimal solutions of its subproblems, a property called *optimal substructure*. For example, if the optimal sequence of actions from s_1 to s_3 goes through s_2, then the subpaths from s_1 to s_2 and from s_2 to s_3 are also optimal. This substructure is shown in figure E.4.

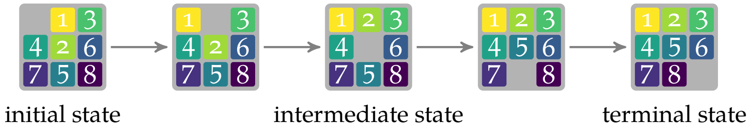


Figure E.4. The sequence of states on the left form an optimal path from the initial state to the terminal state. Shortest path problems have optimal substructure, meaning the sequence from the initial state to the intermediate state is also optimal, as is the sequence from the intermediate state to the terminal state.

In the case of graph search, when evaluating a state we first check a *transposition table* to see whether the state has been previously visited, and if it has, we return its stored value.[2] Otherwise we evaluate the state as normal and store the result in the transposition table. A comparison to forward search is shown in figure E.5.

E.6 Heuristic Search

Heuristic search[3] (algorithm E.5) improves upon branch and bound by ordering its actions based on a provided *heuristic* function $\overline{U}(s)$, which is an upper bound of the return. Like dynamic programming, heuristic search has a mechanism by which state evaluations can be cached in order to avoid redundant computation. Furthermore, heuristic search does not require the lower bound value function needed by branch and bound.[4]

[2] Caching the results of expensive computations so that they can be retrieved rather than being recomputed in the future is called *memoization*.

[3] Heuristic search is also known as *informed search* or *best-first search*.

[4] Our implementation does use two value functions: the heuristic for guiding the search and an approximate value function for evaluating terminal states.

```
function dynamic_programming(𝒫::Search, s, d, U, M=Dict())
    if haskey(M, (d,s))
        return M[(d,s)]
    end
    𝒜, T, R = 𝒫.𝒜(s), 𝒫.T, 𝒫.R
    if isempty(𝒜) || d ≤ 0
        best = (a=nothing, u=U(s))
    else
        best = (a=first(𝒜), u=-Inf)
        for a in 𝒜
            s′ = T(s,a)
            u = R(s,a) + dynamic_programming(𝒫, s′, d-1, U, M).u
            if u > best.u
                best = (a=a, u=u)
            end
        end
    end
    M[(d,s)] = best
    return best
end
```

Algorithm E.4. Dynamic programming applied to forward search, which includes a tranposition table `M`. Here, `M` is a dictionary that stores depth-state tuples from previous evaluations, allowing the method to return previously computed results. The search is performed to depth `d`, at which point the terminal value is estimated with an approximate value function `U`. The returned named tuple consists of the best action `a` and its finite-horizon expected value `u`.

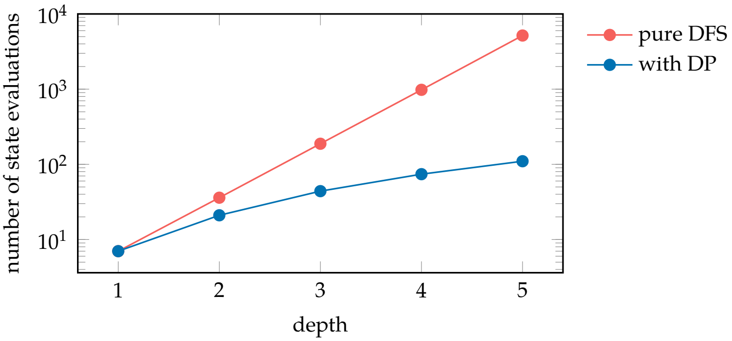


Figure E.5. A comparison of the number of state evaluations for pure forward search and forward search augmented with dynamic programming on the hex-world search problem of example E.1. Dynamic programming is able to avoid the exponential growth in state visitation by caching results.

Actions are sorted based on the immediate reward plus a heuristic estimate of the future return:

$$R(s,a) + \overline{U}(T(s,a)) \tag{E.1}$$

In order to guarantee optimality, the heuristic must be both *admissible* and *consistent*. An admissible heuristic is an upper bound of the true value function. A consistent heuristic is never less than the expected reward gained by transitioning to a neighboring state:

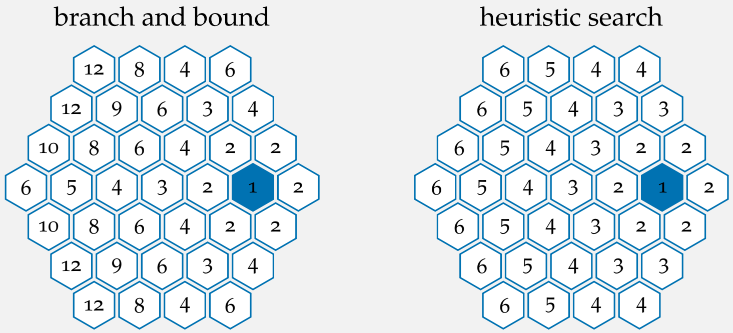
$$\overline{U}(s) \geq R(s,a) + \overline{U}(T(s,a)) \tag{E.2}$$

The method is compared to branch and bound search in example E.2.

```
function heuristic_search(𝒫::Search, s, d, Uhi, U, M)
    if haskey(M, (d,s))
        return M[(d,s)]
    end
    𝒜, T, R = 𝒫.𝒜(s), 𝒫.T, 𝒫.R
    if isempty(𝒜) || d ≤ 0
        best = (a=nothing, u=U(s))
    else
        best = (a=first(𝒜), u=-Inf)
        for a in sort(𝒜, by=a->R(s,a) + Uhi(T(s,a)), rev=true)
            if R(s,a) + Uhi(T(s,a)) ≤ best.u
                break
            end
            s′ = T(s,a)
            u = R(s,a) + heuristic_search(𝒫, s′, d-1, Uhi, U, M).u
            if u > best.u
                best = (a=a, u=u)
            end
        end
    end
    M[(d,s)] = best
    return best
end
```

Algorithm E.5. The heuristic search algorithm for solving a search problem 𝒫 starting from state `s` and searching to a maximum depth `d`. A heuristic `Uhi` is used to guide the search, the approximate vlaue function `U` is evaluated at terminal states, and a transposition table `M` in the form of a dictionary containing depth-state tuples allows the algorithm to cache values from previously explored states.

We can apply heuristic search to the same hex world search problem as in example E.1. We use the heuristic $\overline{U}(s) = 5 - \delta(s)$, where $\delta(s)$ is the number of steps from the given state to the terminal reward state. Below we show the number of states visited when running either branch and bound (left) or heuristic search (right) from each starting state. Branch and bound is just as efficient in states near and to the left of the goal state, whereas heuristic search is able to efficiently search from any intitial state.



branch and bound

12 8 4 6
12 9 6 3 4
10 8 6 4 2 2
6 5 4 3 2 1 2
10 8 6 4 2 2
12 9 6 3 4
12 8 4 6

heuristic search

6 5 4 4
6 5 4 3 3
6 5 4 3 2 2
6 5 4 3 2 1 2
6 5 4 3 2 2
6 5 4 3 3
6 5 4 4

Example E.2. A comparison of the savings that heuristic search can have over branch and bound search. Heuristic search automatically orders actions according to their lookahead heuristic value.