# 9 Online Planning

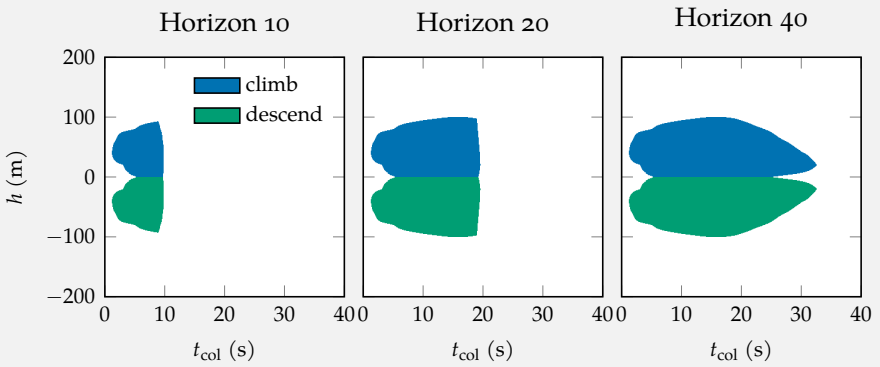
The solution methods we have discussed so far compute policies offline, before any actions are executed in the real problem. Even offline approximation methods can be intractable in many high-dimensional problems. This chapter discusses *online planning* methods that find actions based on reasoning about states reachable from the current state. The *reachable state space* is often orders of magnitude smaller than the full state space, which can significantly reduce storage and computational requirements compared to offline methods. We will discuss a variety of algorithms that aim to make online planning efficient, including pruning the state space, sampling, and planning more deeply along trajectories that appear more promising.

## 9.1 Receding Horizon Planning

In *receding horizon planning*, we plan from the current state to some maximum fixed horizon or depth $d$. We then execute the action from our current state, transition to the next state, and replan. The online planning methods discussed in this chapter follow this receding horizon planning scheme. They differ in how they explore different courses of action.

A challenge in applying receding horizon planning is determining the appropriate depth. Deeper planning generally requires more computation. For some problems, a shallow depth can be quite effective; the fact that we replan at each step can compensate for our lack of longer-term modeling. In other problems, greater planning depths may be necessary so that our planner can be driven towards goals or away from unsafe states as illustrated in example 9.1.

Suppose we want to apply receding horizon planning to aircraft collision avoidance. The objective is to provide descend or climb advisories when necessary to avoid collision. A collision occurs when our altitude relative to the intruder $h$ is within $\pm 50$ m and the time to potential collision $t_{\text{col}} = 0$. We want to plan deeply enough so that we can provide an advisory sufficiently early to avoid collision with a high degree of confidence. The plots below show the actions that would be taken by a receding horizon planner with different depths.




If the depth is $d = 10$, we only provide advisories within 10 s of collision. Due to the limitations of the vehicle dynamics and the uncertainty of the behavior of the other aircraft, providing advisories this late compromises safety. With $d = 20$, we can do better, but there are cases where we would want to alert a little earlier to further reduce collision risk. There is no motivation to plan deeper than $d = 40$ because we do not need to advise any maneuvers that far in advance of potential collision.

Example 9.1. Receding horizon planning for collision avoidance to different planning depths. In this problem, there are four state variables. These plots show slices of the state space under the assumption that the aircraft is currently level and there has not yet been an advisory. The horizontal axis is the time to collision $t_{\text{col}}$ and the vertical axis is our altitude $h$ relative to the intruder. Appendix F.6 provides additional details about the problem.

## 9.2 Lookahead with Rollouts

The previous chapter involved extracting policies that are greedy with respect to an approximate value function $U$ through the use of one-step lookahead.[1] A simple online strategy involves acting greedily with respect to values estimated through simulation to depth $d$. In order to run a simulation, we need a policy to simulate. Of course, we do not know the optimal policy, but we can use what is called a *rollout policy* instead. Rollout policies are typically stochastic, with actions drawn from a distribution $a \sim \pi(s)$. To produce these rollout simulations, we use a *generative model* $s' \sim T(s, a)$ to generate successor states $s'$ from the distribution $T(s' \mid s, a)$. This generative model can be implemented through draws from a random number generator, which can be easier to implement in practice compared to explicitly representing the distribution $T(s' \mid s, a)$.

[1] The lookahead strategy was originally introduced in algorithm 7.2 as part of our discussion of exact solution methods.

Algorithm 9.1 combines one-step lookahead with values estimated through rollout. This approach often results in better behavior than that of the original rollout policy, but optimality is not guaranteed. It can be viewed as an approximate form of policy improvement used in the policy iteration algorithm (section 7.4). A simple variation of this algorithm is to use multiple rollouts to arrive at a better estimate of the expected discounted return. If we run $m$ simulations to estimate the action values, the time complexity is $O(m \times |\mathcal{A}| \times d)$.

## 9.3 Forward Search

*Forward search* determines the best action to take from an initial state $s$ by expanding all possible transitions up to depth $d$. These expansions form a *search tree*.[2] Such search trees have a worst-case branching factor of $|\mathcal{S}| \times |\mathcal{A}|$, yielding a computational complexity of $O((|\mathcal{S}| \times |\mathcal{A}|)^d)$. Figure 9.1 shows a search tree applied to a problem with three states and two actions.

[2] The exploration of the tree occurs as a *depth-first search*. Appendix E reviews depth-first search and other standard search algorithms in the deterministic context.

Algorithm 9.2 calls itself recursively to the specified depth. Once reaching the specified depth, it uses an estimate of the utility provided by the function $U$. If we simply want to plan to the specified horizon, we set $U(s) = 0$. If our problem requires planning beyond the depth we can afford to compute online, we can use an estimate of the value function obtained offline using, for example, one of the value function approximations described in the previous chapter. Combining online and offline approaches in this way is sometimes referred to as *hybrid planning*.

```
struct RolloutLookahead
    𝒫 # problem
    π # rollout policy
    d # depth
end

randstep(𝒫::MDP, s, a) = 𝒫.TR(s, a)

function rollout(𝒫, s, π, d)
    if d ≤ 0
        return 0.0
    end
    a = π(s)
    s′, r = randstep(𝒫, s, a)
    return r + 𝒫.γ*rollout(𝒫, s′, π, d-1)
end

function (π::RolloutLookahead)(s)
    U(s) = rollout(π.𝒫, s, π.π, π.d)
    return greedy(π.𝒫, U, s).a
end
```

Algorithm 9.1. A function that runs a rollout of policy `π` in problem `𝒫` from state `s` to depth `d`. It returns the total discounted reward. This function can be used with the `greedy` function (introduced in algorithm 7.5) to generate an action that is likely to be an improvement over the original rollout policy. We will use this algorithm later for problems other than MDPs, requiring us to only have to modify `randstep` appropriately.

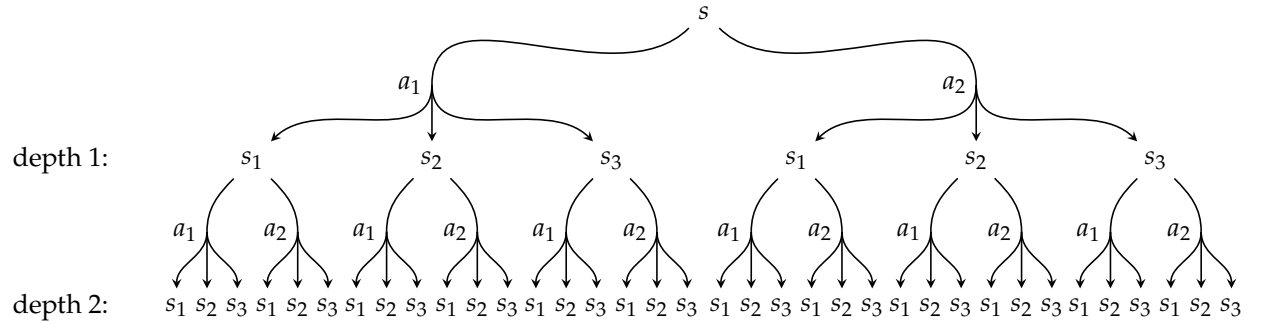



Figure 9.1. A forward search tree for a problem with three states and two actions.

```
struct ForwardSearch
    𝒫 # problem
    d # depth
    U # value function at depth d
end

function forward_search(𝒫, s, d, U)
    if d ≤ 0
        return (a=nothing, u=U(s))
    end
    best = (a=nothing, u=-Inf)
    U′(s) = forward_search(𝒫, s, d-1, U).u
    for a in 𝒫.𝒜
        u = lookahead(𝒫, U′, s, a)
        if u > best.u
            best = (a=a, u=u)
        end
    end
    return best
end

(π::ForwardSearch)(s) = forward_search(π.𝒫, s, π.d, π.U).a
```

Algorithm 9.2. The forward search algorithm for finding an approximately optimal action online for a problem $\mathcal{P}$ 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`. The problem type is not constrained to be an MDP; section 22.2 uses this same algorithm in the context of partially observable problems with a different implementation for `lookahead`.

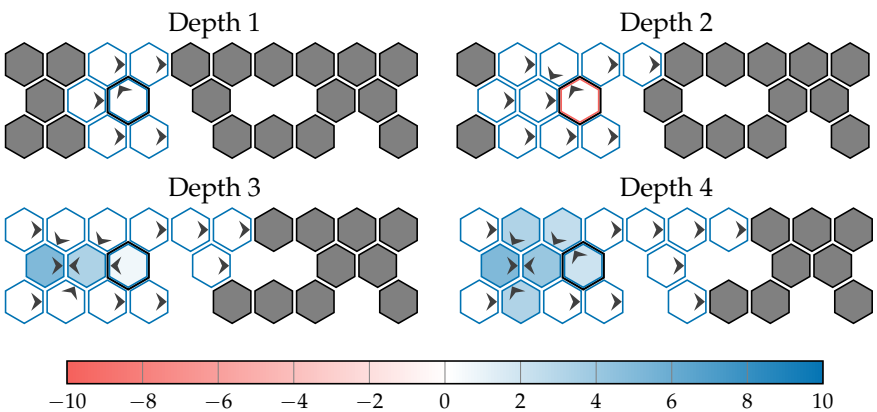



Figure 9.2. Forward search applied to the hex world problem with four different maximum depths. The search can visit a node multiple times. The actions and colors for visited states were chosen according to the shallowest, highest-value node in the search tree for that state. The initial state has an additional black border.

## 9.4 Branch and Bound

*Branch and bound* (algorithm 9.3) attempts to avoid the exponential computational complexity of forward search. It prunes branches by reasoning about bounds on the value function. The algorithm requires knowing a lower bound on the value function $\underline{U}(s)$ and an upper bound on the action value function $\overline{Q}(s,a)$. The lower bound is used to evaluate the states at the maximum depth. This lower bound is propagated upwards through the tree through Bellman updates. If we find that the upper bound of an action at a state is lower than the lower bound of a previously explored action from that state, then we need not explore that action, allowing us to *prune* the associated subtree from consideration.

```
struct BranchAndBound
    𝒫  # problem
    d  # depth
    Ulo # lower bound on value function at depth d
    Qhi # upper bound on action value function
end

function branch_and_bound(𝒫, s, d, Ulo, Qhi)
    if d ≤ 0
        return (a=nothing, u=Ulo(s))
    end
    U′(s) = branch_and_bound(𝒫, s, d-1, Ulo, Qhi).u
    best = (a=nothing, u=-Inf)
    for a in sort(𝒫.𝒜, by=a->Qhi(s,a), rev=true)
        if Qhi(s, a) < best.u
            return best # safe to prune
        end
        u = lookahead(𝒫, U′, s, a)
        if u > best.u
            best = (a=a, u=u)
        end
    end
    return best
end

(π::BranchAndBound)(s) = branch_and_bound(π.𝒫, s, π.d, π.Ulo, π.Qhi).a
```

Algorithm 9.3. The branch and bound algorithm for finding of an approximately optimal action online for a discrete MDP `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`. This algorithm is also used for POMDPs.

Branch and bound will give the same result as forward search, but it can be more efficient depending on how many branches are pruned. The worst-case complexity of branch and bound is still the same as forward search. To facilitate

pruning, actions are traversed in descending order by upper bound. Tighter bounds will generally result in more pruning as shown in example 9.2.

Consider applying branch and bound to the mountain car problem. We can use the value function of a heuristic policy for the lower bound $\underline{U}(s)$, such as a heuristic policy that always accelerates in the direction of motion. For our upper bound $\overline{Q}([x, v], a)$, we can use the return expected when accelerating toward the goal with no hill. Branch and bound visits about a third as many states as forward search.

Example 9.2. An example of branch and bound applied to the mountain car problem (appendix F.4). Branch and bound can achieve a significant speedup over forward search.

## 9.5 Sparse Sampling

A method known as *sparse sampling*[3] (algorithm 9.4) attempts to reduce the branching factor of forward search and branch and bound. Instead of branching on all possible next states, we only consider a limited number of samples of the next state. Although the sampling of the next state results in an approximation, this method can work well in practice and can significantly reduce computation. If we draw $m$ samples of the next state for each action node in the search tree, the computational complexity is $O\left((m \times |\mathcal{A}|)^d\right)$, which is still exponential in the depth but no longer depends on the size of the state space. Figure 9.3 shows an example.

[3] M. J. Kearns, Y. Mansour, and A. Y. Ng, "A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes," *Machine Learning*, vol. 49, no. 2–3, pp. 193–208, 2002.

## 9.6 Monte Carlo Tree Search

*Monte Carlo tree search*[4] (algorithm 9.5) avoids the exponential complexity in the horizon by running $m$ simulations from the current state. During these simulations, the algorithm updates estimates of the action value function $Q(s, a)$ and a record of the number of times a particular state-action pair has been selected, $N(s, a)$. After running these $m$ simulations from our current state $s$, we simply choose the action that maximizes our estimate of $Q(s, a)$.

A simulation (algorithm 9.6) begins by traversing the explored state space, consisting of the states for which we have estimates of $Q$ and $N$. The action we

[4] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, et al., "A Survey of Monte Carlo Tree Search Methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

```
struct SparseSampling
    𝒫 # problem
    d # depth
    m # number of samples
    U # value function at depth d
end

function sparse_sampling(𝒫, s, d, m, U)
    if d ≤ 0
        return (a=nothing, u=U(s))
    end
    best = (a=nothing, u=-Inf)
    for a in 𝒫.𝒜
        u = 0.0
        for i in 1:m
            s′, r = randstep(𝒫, s, a)
            a′, u′ = sparse_sampling(𝒫, s′, d-1, m, U)
            u += (r + 𝒫.γ*u′) / m
        end
        if u > best.u
            best = (a=a, u=u)
        end
    end
    return best
end

(π::SparseSampling)(s) = sparse_sampling(π.𝒫, s, π.d, π.m, π.U).a
```

Algorithm 9.4. The sparse sampling algorithm for finding of an approximately optimal action online for a discrete problem `𝒫` from a current state `s` to depth `d` with `m` samples per action. The returned named tuple consists of the best action `a` and its finite-horizon expected value `u`.

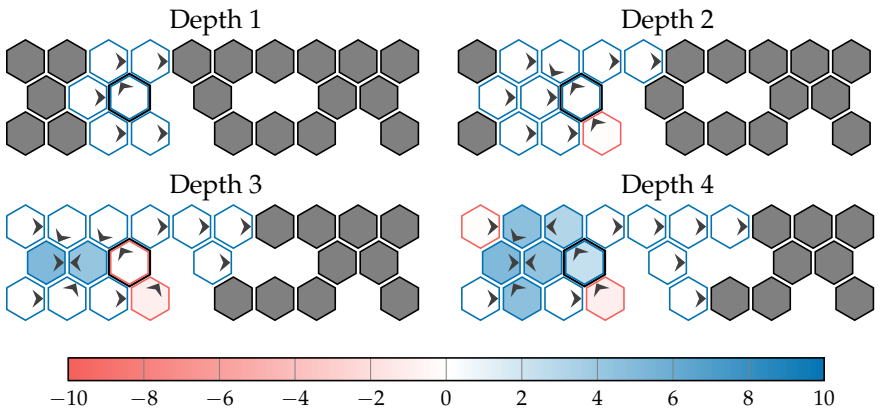



Figure 9.3. Sparse sampling with $m = 10$ applied to the hex world problem. Tiles are colored according to their value if they were visited. The bordered tile is the initial state. Compare to forward search in figure 9.2.

```
struct MonteCarloTreeSearch
    𝒫     # problem
    N     # visit counts
    Q     # action value estimates
    d     # depth
    m     # number of simulations
    c     # exploration constant
    π     # rollout policy
end

function (π::MonteCarloTreeSearch)(s)
    for k in 1:π.m
        simulate!(π, s)
    end
    return _argmax(a→π.Q[(s,a)], π.𝒫.𝒜)
end
```

Algorithm 9.5. The Monte Carlo tree search policy for finding of an approximately optimal action from a current state `s`.

take from state $s$ is the one that maximizes the *upper confidence bound*:

$$Q(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}} \tag{9.1}$$

where $N(s) = \sum_a N(s,a)$ is the total visit count to $s$ and $c$ is an exploration parameter that scales the value of unexplored actions. The second term corresponds to an *exploration bonus*. If $N(s,a) = 0$, the bonus is defined to be infinity. With $N(s,a)$ in the denominator, the exploration bonus is higher for actions that have not been tried as frequently.[5] Algorithm 9.7 implements this exploration strategy.

[5] We will discuss exploration strategies in chapter 15.

As we take actions specified by algorithm 9.7, we step into new states sampled from the generative model $T(s,a)$, similar to the sparse sampling method. We increment the visit count $N(s,a)$ and update $Q(s,a)$ to maintain the mean value.

At some point, we will either reach the maximum depth or a state that we have not yet explored. If we reach an unexplored state $s$, we initialize $N(s,a)$ and $Q(s,a)$ to zero for each action $a$. We may modify algorithm 9.6 to initialize these counts and value estimates to some other values based on prior expert knowledge of the problem. After initializing $N$ and $Q$, we then arrive at a value estimate through rollout of some policy using the process outlined in section 9.2.

Examples 9.3 to 9.7 work through an illustration of Monte Carlo tree search applied to the 2048 problem. Figure 9.4 shows a search tree generated by running Monte Carlo tree search on 2048. Example 9.8 discusses the impact of using different strategies for estimating values.

```
function simulate!(π::MonteCarloTreeSearch, s, d=π.d)
    if d ≤ 0
        return 0.0
    end
    𝒫, N, Q, c = π.𝒫, π.N, π.Q, π.c
    𝒜, TR, γ = 𝒫.𝒜, 𝒫.TR, 𝒫.γ
    if !haskey(N, (s, first(𝒜)))
        for a in 𝒜
            N[(s,a)] = 0
            Q[(s,a)] = 0.0
        end
        return rollout(𝒫, s, π.π, d)
    end
    a = explore(π, s)
    s′, r = TR(s,a)
    q = r + γ*simulate!(π, s′, d-1)
    N[(s,a)] += 1
    Q[(s,a)] += (q-Q[(s,a)])/N[(s,a)]
    return q
end
```

Algorithm 9.6. A simulation in Monte Carlo tree search starting from state `s` to depth `d`.

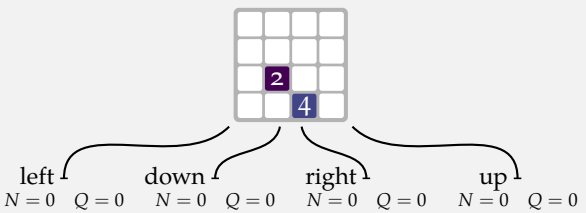
```
bonus(Nsa, Ns) = Nsa == 0 ? Inf : sqrt(log(Ns)/Nsa)

function explore(π::MonteCarloTreeSearch, s)
    𝒜, N, Q, c = π.𝒫.𝒜, π.N, π.Q, π.c
    Ns = sum(N[(s,a)] for a in 𝒜)
    return _argmax(a->Q[(s,a)] + c*bonus(N[(s,a)], Ns), 𝒜)
end
```

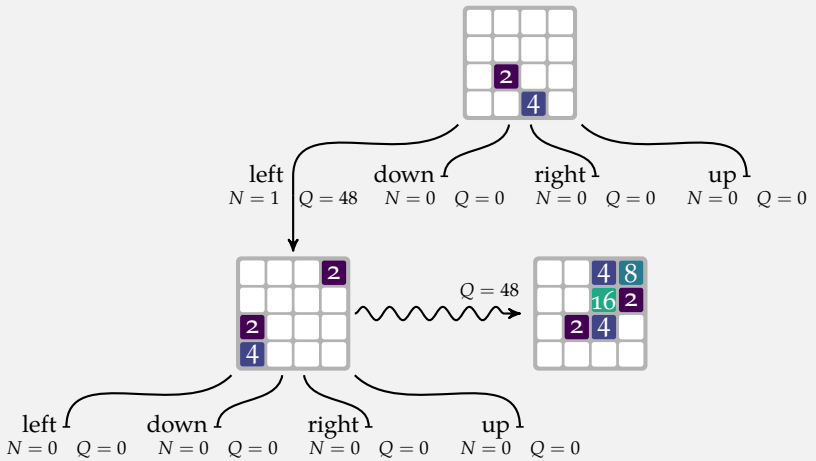
Algorithm 9.7. An exploration policy used in Monte Carlo tree search when determining which nodes to traverse through the search tree. The policy is determined by a dictionary of state-action visitation counts `N` and values `Q`, as well as an exploration parameter `c`. When `N[(s,a)] = 0`, the policy returns infinity.

Example 9.3. An example of solving 2048 with Monte Carlo tree search.

Consider using Monte Carlo tree search to play 2048 (appendix F.2) with a maximum depth $d = 10$, an exploration parameter $c = 100$, and a uniform random rollout policy. Our first simulation expands the starting state. The count and value are initialized for each action from the initial state:



The second simulation begins by selecting the best action from the initial state according to our exploration strategy in equation (9.1). Because all states have the same value, we arbitrarily choose the first action, `left`. We then sample a new successor state and expand it, initializing the associated counts and value estimates. A rollout is run from the successor state and its value is used to update the value of `left`.



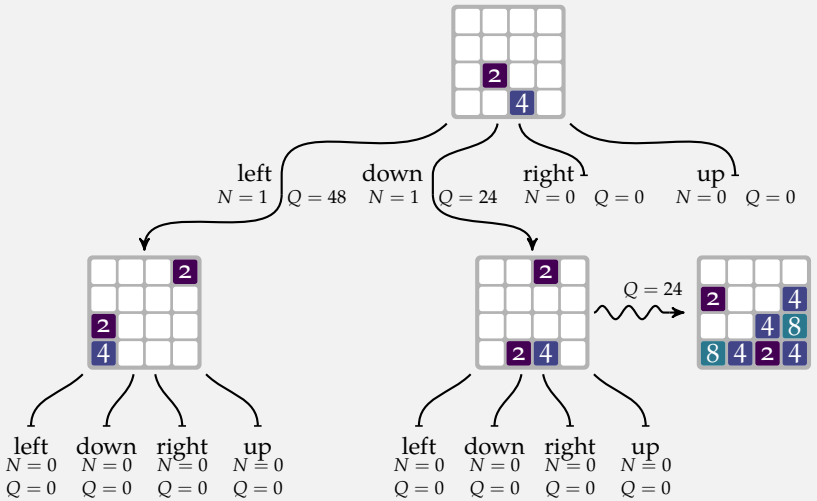

Example 9.4. A (continued) example of solving 2048 with Monte Carlo tree search.

Example 9.5. A (continued) example of solving 2048 with Monte Carlo tree search.

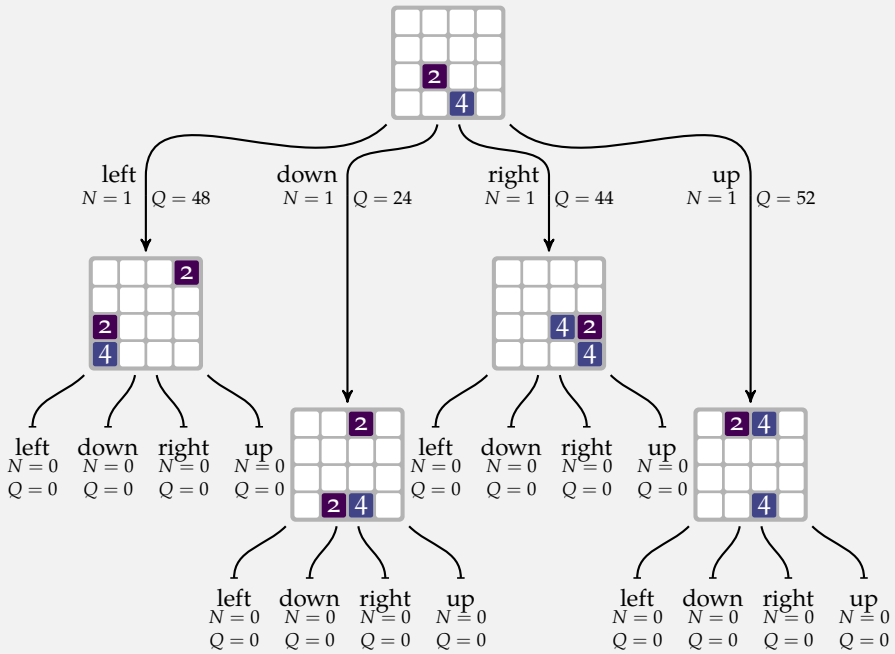
The third simulation begins by selecting the second action, `down`, because it has infinite value due to the exploration bonus given for unexplored actions. The first action has finite value:

$$Q(s_0, \texttt{left}) + c\sqrt{\frac{\log N(s_0)}{N(s_0, \texttt{left})}} = 48 + 100\sqrt{\frac{\log 2}{1}} \approx 117.315$$

We take the `down` action and sample a new successor state, which is expanded. A rollout is run from the successor state and its value is used to update the value of `down`.



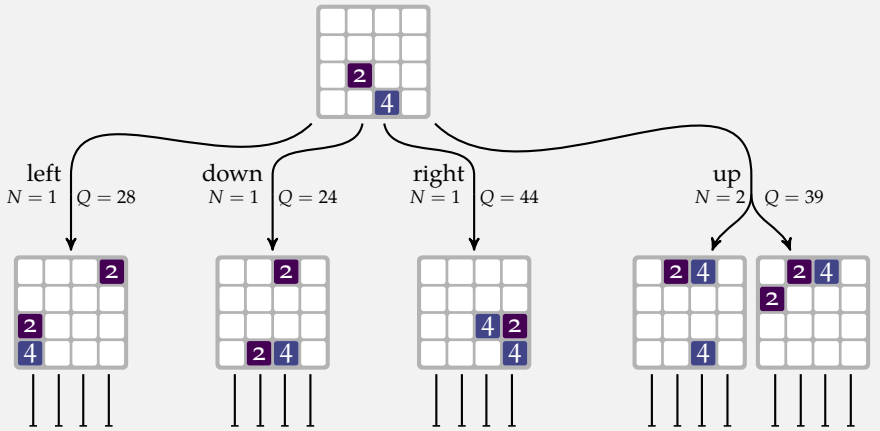
The next two simulations select `right` and `up`, respectively. This results in:




Example 9.6. A (continued) example of solving 2048 with Monte Carlo tree search.

Example 9.7. A (continued) example of solving 2048 with Monte Carlo tree search.

In the fifth simulation, up has the highest value. The successor state after taking up in the source state will not necessarily be the same as the first time up was selected. A new successor node is created.




There are variations of this basic Monte Carlo tree search algorithm to better handle large action and state spaces. Instead of expanding all of the actions, we can use *progressive widening*. The number of actions considered from state $s$ is limited to $\theta_1 N(s)^{\theta_2}$, where $\theta_1$ and $\theta_2$ are hyper-parameters. Similarly, we can limit the number of states that result from taking action $a$ from state $s$ in the same way, using what is called *double progressive widening*. If the number of states that have been simulated from state $s$ after action $a$ is below $\theta_3 N(s,a)^{\theta_4}$, then we sample a new state; otherwise, we sample one of the previously sampled states with probability proportional to the number of times it has been visited. This strategy can be used to handle large as well as continuous action and state spaces.[6]

[6] A. Couëtoux, J.-B. Hoock, N. Sokolovska, O. Teytaud, and N. Bonnard, "Continuous Upper Confidence Trees," in *Learning and Intelligent Optimization (LION)*, 2011.

## 9.7 Heuristic Search

*Heuristic search* (algorithm 9.8) uses $m$ simulations of a greedy policy with respect to a value function $U$ from the current state $s$.[7] The value function $U$ is initialized to an upper bound of the value function $\overline{U}$, which is referred to as a *heuristic*. As we run these simulations, we update our estimate of $U$ through lookahead. After running these simulations, we simply select the greedy action from $s$ with respect

[7] A. G. Barto, S. J. Bradtke, and S. P. Singh, "Learning to Act Using Real-Time Dynamic Programming," *Artificial Intelligence*, vol. 72, no. 1–2, pp. 81–138, 1995. Other forms of heuristic search are discussed by Mausam and A. Kolobov, *Planning with Markov Decision Processes: An AI Perspective*. Morgan & Claypool, 2012.

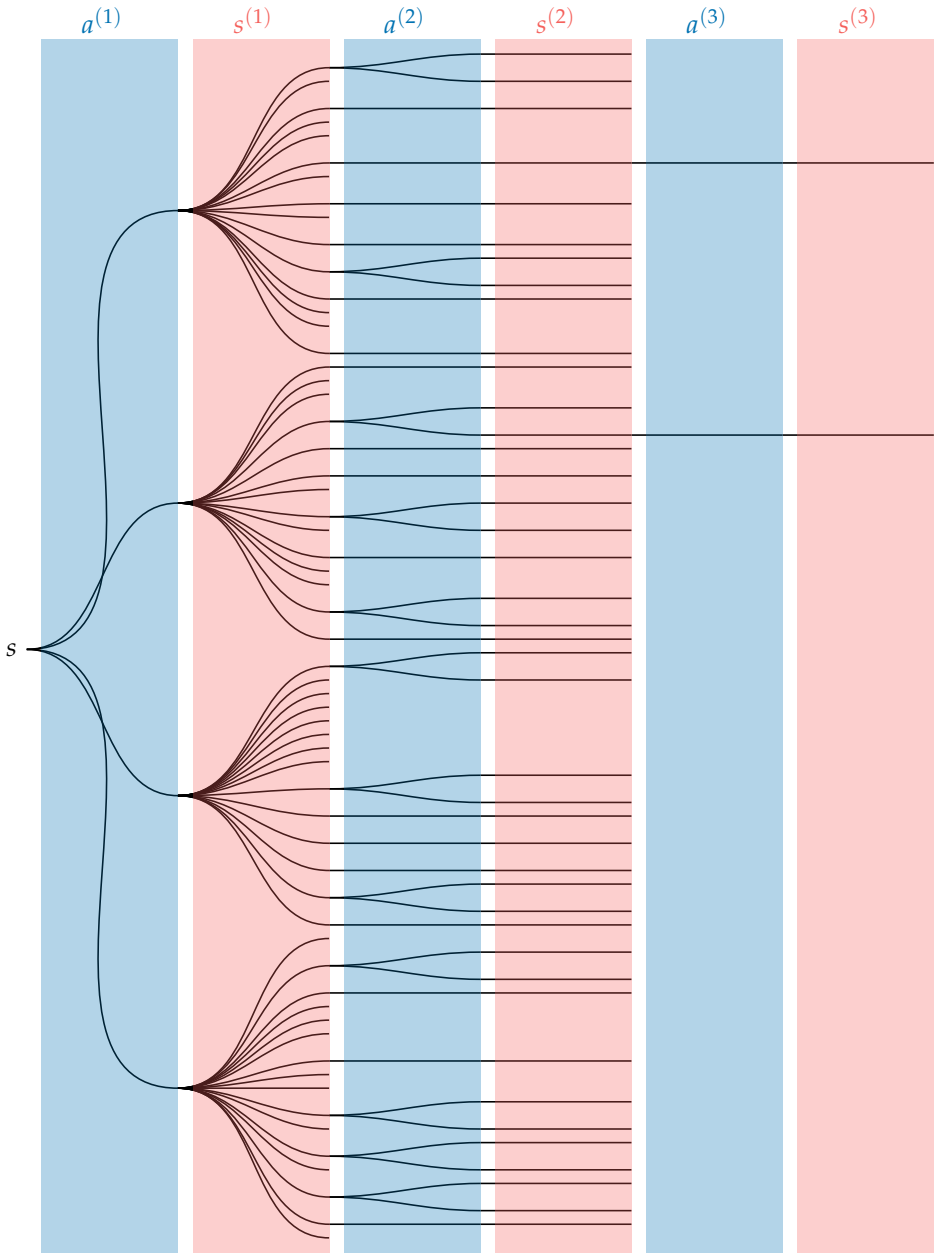



Figure 9.4. A Monte Carlo tree search tree on 2048 after 100 simulations with red states and blue actions. In general, Monte Carlo tree search for MDPs produces a search graph because there can be multiple ways to reach the same state. This tree visualization shows the progression of rollouts. The tree is shallow with a fairly high branching factor.

Example 9.8. The performance of Monte Carlo tree search varies with the number of simulations and as the board evaluation method is changed. Heuristic board evaluations tend to be efficient and can more effectively guide search when run counts are low. Lookahead rollout evaluations take about 18 times longer than heuristic evaluations.

Rollouts are not the only means by which we can estimate utilities in Monte Carlo tree search. Custom evaluation functions can often be crafted for specific problems to help guide the algorithm. For example, we can encourage Monte Carlo tree search to order its tiles in 2048 using evaluation functions that return the weighted sum across tile values:

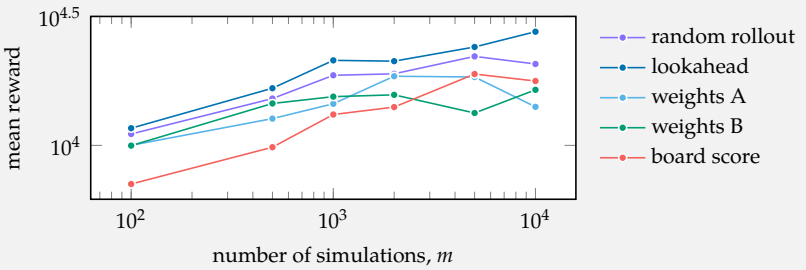
| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 2 | 3 | 4 | 5 |
| 3 | 4 | 5 | 6 |
| 4 | 5 | 6 | 7 |

heuristic A weights

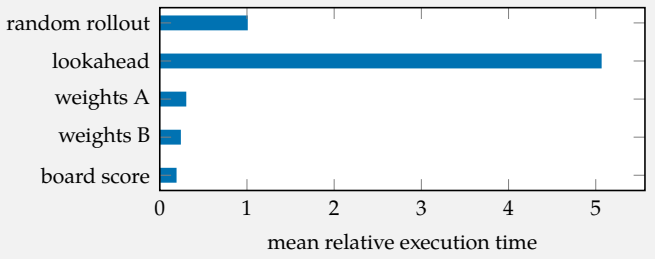
| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 7 | 6 | 5 | 4 |
| 8 | 9 | 10 | 11 |
| 15 | 14 | 13 | 12 |

heuristic B weights

The plot below compares Monte Carlo tree search on 2048 using rollouts with a uniform random policy, rollouts with a one-step lookahead policy, the two evaluation functions above, and using the current board score.




Rollouts perform well, but require more execution time. Below we plot the average execution time relative to random rollouts for $m = 100$ from a starting state.



to $U$. Figure 9.5 shows how $U$ and the greedy policy changes with the number of simulations.

Heuristic search is guaranteed to converge to the optimal utility function so long as the heuristic $\overline{U}$ is indeed an upper bound on the value function.[8] The efficiency of the search depends on the tightness of the upper bound. Unfortunately, tight bounds can be difficult to obtain in practice. While a heuristic that is not a true upper bound may not converge to the optimal policy, it may still converge to a policy that performs well. The time complexity is $O(m \times d \times |S| \times |A|)$.

[8] Such a heuristic is referred to as an *admissable heuristic*.

```
struct HeuristicSearch
    𝒫     # problem
    Uhi   # upper bound on value function
    d     # depth
    m     # number of simulations
end

function simulate!(π::HeuristicSearch, U, s)
    𝒫, d = π.𝒫, π.d
    for d in 1:d
        a, u = greedy(𝒫, U, s)
        U[s] = u
        s = rand(𝒫.T(s, a))
    end
end

function (π::HeuristicSearch)(s)
    U = [π.Uhi(s) for s in π.𝒫.𝒮]
    for i in 1:m
        simulate!(π, U, s)
    end
    return greedy(π.𝒫, U, s).a
end
```

Algorithm 9.8. Heuristic search runs `m` simulations starting from an initial state `s` to a depth `d`. The search is guided by a heuristic initial value function `Uhi`, which leads to optimality in the limit of simulations if it is an upper bound on the optimal value function.

## 9.8 Labeled Heuristic Search

*Labeled heuristic search* (algorithm 9.9) is a variation of heuristic search that runs simulations with value updates while labeling states based on whether their value is solved.[9] We say that a state $s$ is solved if their utility residual falls below some threshold $\delta > 0$:

$$|U_{k+1}(s) - U_k(s)| < \delta \tag{9.2}$$

[9] B. Bonet and H. Geffner, "Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2003.

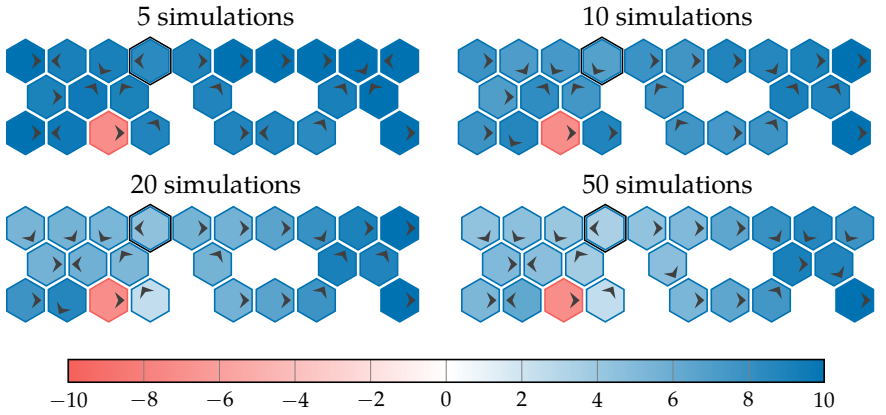



Figure 9.5. Heuristic search runs simulations with Bellman updates to improve a value function on the hex world problem to obtain a policy from an initial state, shown here with an additional black border. These simulations are run to depth 8 with heuristic $\overline{U}(s) = 10$. Each hex is colored according to the utility function value in that iteration. We see that the algorithm eventually finds an optimal policy.

We run simulations with value updates until the current state is solved. In contrast with the heuristic search in the previous section that runs a fixed number of iterations, this labeling process focuses computational effort in the most important areas of the state space.

```
struct LabeledHeuristicSearch
    𝒫     # problem
    Uhi   # upper bound on value function
    d     # depth
    δ     # gap threshold
end

function (π::LabeledHeuristicSearch)(s)
    U, solved = [π.Uhi(s) for s in 𝒫.𝒮], Set()
    while s ∉ solved
        simulate!(π, U, solved, s)
    end
    return greedy(π.𝒫, U, s).a
end
```

Algorithm 9.9. Labeled heuristic search, which runs simulations starting from the current state to a depth `d` until the current state is solved. The search is guided by a heuristic upper bound on the value function `Uhi` and maintains a growing set of solved states `solved`. States are considered solved when their utility residuals fall below `δ`. A value function policy is returned.

Simulations in labeled heuristic search (algorithm 9.10) begin by running to a maximum depth of $d$ by following a policy that is greedy with respect to our estimated value function $U$, similar to the heuristic search in the previous section. We may stop a simulation before a depth of $d$ if we reach a state that has been labeled as solved in a prior simulation.

```
function simulate!(π::LabeledHeuristicSearch, U, solved, s)
    visited = []
    for d in 1:π.d
        if s ∈ solved
            break
        end
        push!(visited, s)
        a, u = greedy(π.𝒫, U, s)
        U[s] = u
        s = rand(π.𝒫.T(s, a))
    end
    while !isempty(visited)
        if label!(π, U, solved, pop!(visited))
            break
        end
    end
end
```

Algorithm 9.10. Simulations are run from the current state to a maximum depth `d`. We stop a simulation at depth `d` or if we encounter a state that is in the set `solved`. After a simulation, we call `label!` on the states we visited in reverse order.

After each simulation, we iterate over the states we visited during that simulation in reverse order, performing a labeling routine on each state and stopping if a state is found that is not solved. The labeling routine (algorithm 9.11) searches the states in the *greedy envelope* of $s$, which is defined to be the states reachable from $s$ under a greedy policy with respect to $U$. The state $s$ is considered not solved if there is a state in the greedy envelope of $s$ whose utility residual is greater than the threshold $\delta$. If no such state is found, then $s$ is marked as solved—as well as all states in the greedy envelope of $s$ because they must have converged as well. If a state with a sufficiently large utility residual is found, then the utilities all of states traversed during the search of the greedy enveloped are updated.

Figure 9.6 shows several different greedy envelopes. Figure 9.7 shows the states traversed in a single iteration of labeled heuristic search. Figure 9.8 shows the progression of heuristic search on the hex world problem.

## 9.9 Open-Loop Planning

The online methods discussed in this chapter as well as the offline methods discussed in the previous chapters are examples of *closed-loop planning*, which involves accounting for future state information in the planning process.[10] Often *open-loop planning* can provide a satisfactory approximation of an optimal closed-loop plan while greatly enhancing computational efficiency by avoiding having

[10] The loop in this context refers to the observe-act loop introduced in section 1.1.

```
function expand(π::LabeledHeuristicSearch, U, solved, s)
    𝒫, δ = π.𝒫, π.δ
    𝒮, 𝒜, T = 𝒫.𝒮, 𝒫.𝒜, 𝒫.T
    found, toexpand, envelope = false, Set(s), []
    while !isempty(toexpand)
        s = pop!(toexpand)
        push!(envelope, s)
        a, u = greedy(𝒫, U, s)
        if abs(U[s] - u) > δ
            found = true
        else
            for s′ in 𝒮
                if T(s,a,s′) > 0 && s′ ∉ (solved ∪ envelope)
                    push!(toexpand, s′)
                end
            end
        end
    end
    return (found, envelope)
end

function label!(π::LabeledHeuristicSearch, U, solved, s)
    if s ∈ solved
        return false
    end
    found, envelope = expand(π, U, solved, s)
    if found
        for s ∈ reverse(envelope)
            U[s] = greedy(π.𝒫, U, s).u
        end
    else
        union!(solved, envelope)
    end
    return found
end
```

Algorithm 9.11. The `label!` function will attempt to find a state in the greedy envelope of `s` whose utility residual exceeds some threshold `δ`. The function `expand` computes the greedy envelope of `s` and determines whether any of those states have utility residuals above the threshold. If a state has a residual that exceeds some threshold, then we update utilities of the states in the envelope. Otherwise, we add that envelope to the set of solved states.

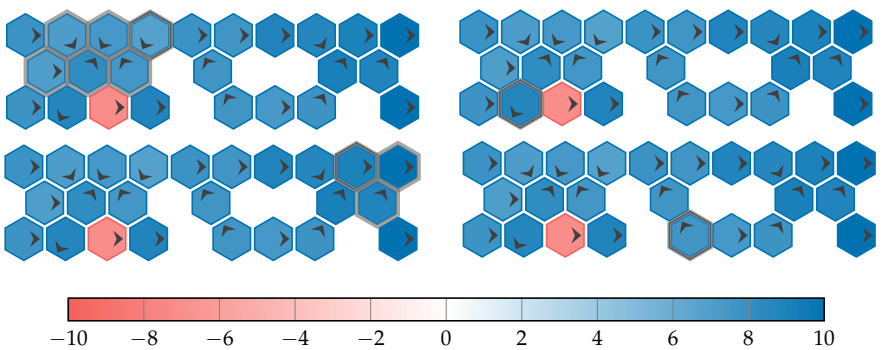



Figure 9.6. The greedy envelope for $\delta = 1$ for several different states visualized for a value function on the hex world problem. The value function was obtained by running basic heuristic search for 10 iterations from the first darkly bordered state to a maximum depth of 8. We find that the size of the greedy envelope can vary widely depending on the state.

Figure 9.7. A single iteration of labeled heuristic search conducts an exploratory run (arrows) followed by labeling (hexagonal border). Only two states are labeled in this iteration: the hidden terminal state and the state with a hexagonal border. Both the exploratory run and the labeling step update the value function.

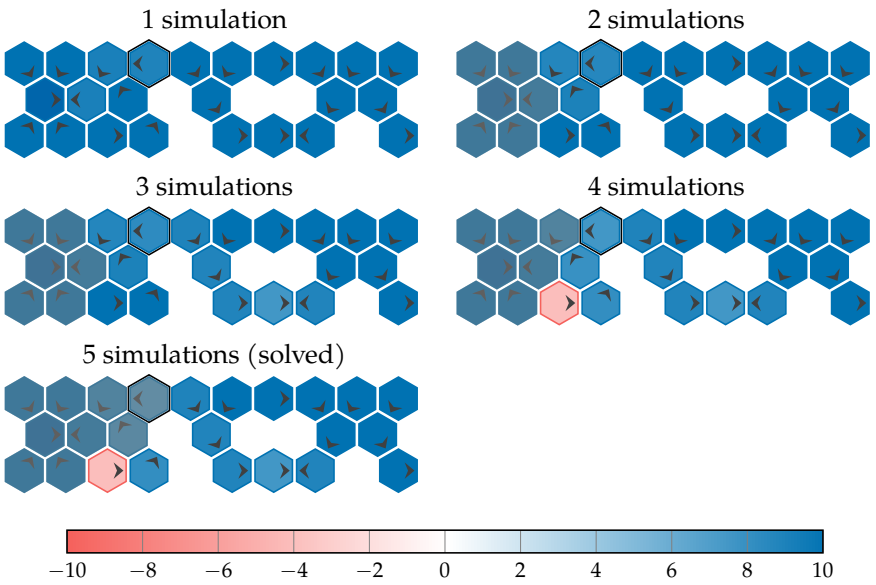



Figure 9.8. A progression of heuristic search on the hex world problem using $\delta = 1$ and a heuristic $\overline{U}(s) = 10$. The solved states in each iterations are covered in an opaque gray wash. The set of solved states grows from the terminal reward state back toward the initial state with the dark border.

to reason about the acquisition of future information. Sometimes this open-loop planning approach is referred to as *model predictive control*.[11] As with receding horizon control, model predictive control solves the open-loop problem, executes the action from our current state, transition to the next state, and then replans.

[11] F. Borrelli, A. Bemporad, and M. Morari, *Predictive Control for Linear and Hybrid Systems*. Cambridge University Press, 2019, 446 pp..

Open-loop plans can be represented as a sequence of actions up to some depth $d$. The planning process reduces to an optimization problem

$$\underset{a_{1:d}}{\text{maximize}} \quad U(a_{1:d}) \tag{9.3}$$

where $U(a_{1:d})$ is the expected return when executing the sequence of actions $a_{1:d}$. Depending on the application, this optimization problem may be convex or lends itself to a convex approximation, meaning that it can be solved quickly using a variety of different algorithms.[12] Later in this section, we will discuss a few different formulations that can be used to transform equation (9.3) into a convex problem.

[12] Appendix A.6 reviews convexity. An introduction to convex optimization is provided by S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

Open-loop planning can often allow us to devise effective decision strategies in high-dimensional spaces where closed-loop planning is computationally infeasible. Open-loop planning gains this efficiency by not accounting for future information. Example 9.9 provides a simple example of where open-loop planning can result in poor decisions, even when we account for stochasticity.

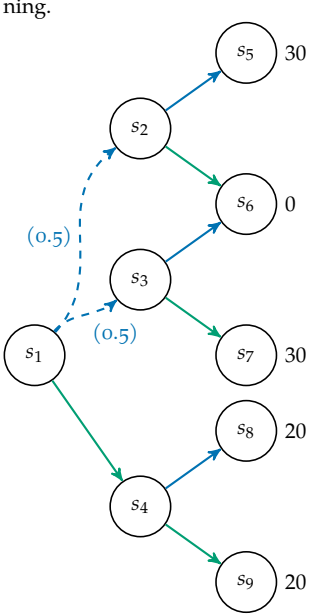
### 9.9.1 Deterministic Model Predictive Control

A common approximation to make $U(a_{1:d})$ convex is to assume deterministic dynamics. We can use the following formulation

$$\begin{aligned} \underset{a_{1:d}, s_{2:d}}{\text{maximize}} \quad & \sum_{t=1}^{d} \gamma^t R(s_t, a_t) \\ \text{subject to} \quad & s_{t+1} = T(s_t, a_t), \quad t \in 1:d-1 \end{aligned} \tag{9.4}$$

where $s_1$ is the current state and $T(s, a)$ is a deterministic transition function that returns the state that results from taking action $a$ from state $s$. Common strategies for producing a suitable deterministic transition function from a stochastic transition function include using the most likely transition for each action and sampling a random transition once for each state-action input pair.

Example 9.9. Example illustrating suboptimality of open-loop planning.




Consider a problem with nine states, as shown in the margin, with two decision steps starting from the initial state $s_1$. In our decisions, we must decide between going up (blue arrows) and going down (green arrows). The effects of these actions are deterministic, except that if we go up from $s_1$, then we end up in state $s_2$ half the time and in state $s_3$ half the time. We receive a reward of 30 in states $s_5$ and $s_7$ and a reward of 20 in states $s_8$ and $s_9$, as indicated in the figure.

There are exactly four open-loop plans: (up, up), (up, down), (down, up), and (down, down). In this simple example, it is easy to compute their expected utilities:

- $U(\text{up}, \text{up}) = 0.5 \times 30 + 0.5 \times 0 = 15$
- $U(\text{up}, \text{down}) = 0.5 \times 0 + 0.5 \times 30 = 15$
- $U(\text{down}, \text{up}) = 20$
- $U(\text{down}, \text{down}) = 20$

According to the set of open-loop plans, it is best to choose down from $s_1$ because our expected reward is 20 instead of 15.

Closed-loop planning, in contrast, takes into account the fact that we can base our next decision on the observed outcome of our first action. If we choose to go up from $s_1$, then we can choose to go down or up depending on whether we end up in $s_2$ or $s_3$, thereby guaranteeing a reward of 30.

Example 9.10 provides an example involving navigating to a goal state while avoiding an obstacle and minimizing acceleration effort. Both the state space and action space are continuous, and we can find a solution in well under a second. Replanning after every step can help compensate for stochasticity or unexpected events. For example, if the obstacle moves, we can readjust our plan as illustrated in figure 9.9.

In this problem, our state $\mathbf{s}$ represents our two-dimensional position concatenated with our two-dimensional velocity vector, with $\mathbf{s}$ initially set to $[0, 0, 0, 0]$. Our action $\mathbf{a}$ is an acceleration vector, where each component must be between $\pm 1$. At each step, we use our action to update our velocity, and we use our velocity to update our position. Our objective is to reach a goal state of $\mathbf{s}_{\text{goal}} = [10, 10, 0, 0]$. We plan to $d = 10$ steps with no discounting. With each step, we accumulate a cost of $\|\mathbf{a}_t\|_2^2$ to minimize acceleration effort. At the last step, we want to be as close to the goal state as possible with a penalty of $100\|\mathbf{s}_d - \mathbf{s}_{\text{goal}}\|_2^2$. We also have to ensure that we avoid a circular obstacle with radius 2 centered at $[3, 4]$. We can formulate this problem as follows and extract the first action from the plan.

```
model = Model(Ipopt.Optimizer)
d = 10
current_state = zeros(4)
goal = [10,10,0,0]
obstacle = [3,4]
@variables model begin
    s[1:4, 1:d]
    -1 ≤ a[1:2,1:d] ≤ 1
end
# velocity update
@constraint(model, [i=2:d,j=1:2], s[2+j,i] == s[2+j,i-1] + a[j,i-1])
# position update
@constraint(model, [i=2:d,j=1:2], s[j,i] == s[j,i-1] + s[2+j,i-1])
# initial condition
@constraint(model, s[:,1] .== current_state)
# obstacle
@constraint(model, [i=1:d], sum((s[1:2,i] - obstacle).^2) ≥ 4)
@objective(model, Min, 100*sum((s[:,d] - goal).^2) + sum(a.^2))
optimize!(model)
action = value.(a[:,1])
```

Example 9.10. An example of open-loop planning in a deterministic environment. We attempt to find a path around a circular obstacle. This implementation uses the `JuMP.jl` interface to the Ipopt solver. A. Wächter and L. T. Biegler, "On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming," *Mathematical Programming*, vol. 106, no. 1, pp. 25–57, 2005.

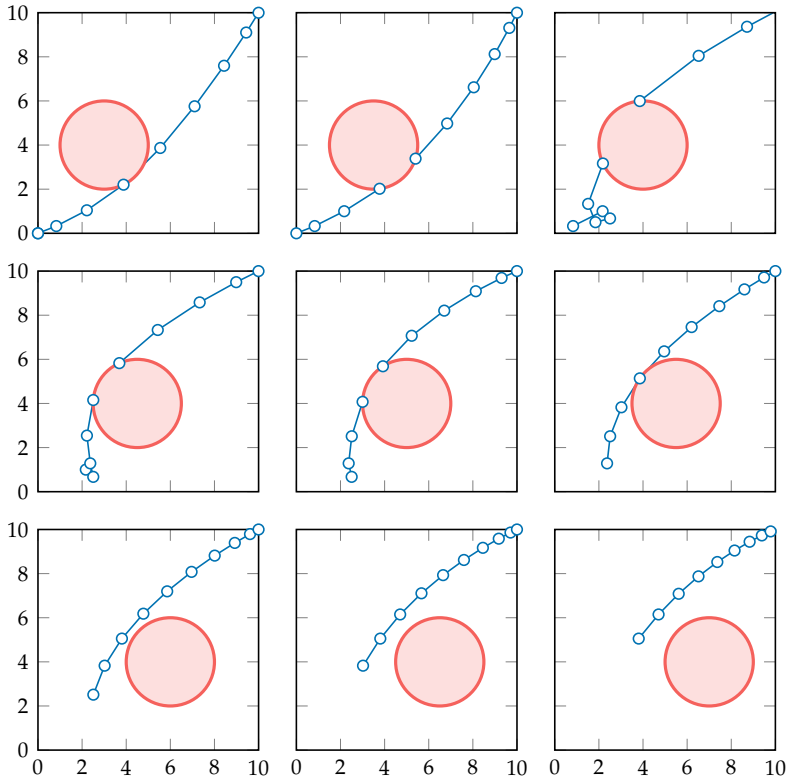



Figure 9.9. Model predictive control applied to the problem in example 9.10 with the addition of a moving obstacle. The sequence progresses left-to-right, top-to-bottom. Initially, we have a plan that passes to the right of the obstacle, but in the third cell, we see that we must change our mind and pass to the left. We have to maneuver around a little to adjust our velocity vector appropriately with minimal effort. Of course, we could have created a better path (in terms of our utility function) if our planning process knew that the obstacle was moving in a particular direction.

### 9.9.2 Robust Model Predictive Control

We can change the problem formulation to provide robustness to outcome uncertainty. There are many different *robust model predictive control* formulations,[13] but one involves choosing the best open-loop plan given the worst-case state transitions. This formulation defines $T(s, a)$ to be an *uncertainty set* consisting of all possible states that can result from taking action $a$ in state $s$. In other words, the uncertainty set is the support of the distribution $T(\cdot \mid s, a)$. Optimizing with respect to worse-case state transitions requires transforming the optimization problem in equation (9.4) into a *minimax* or *min-max* problem:

$$\begin{aligned} &\underset{a_{1:d}}{\text{maximize}} \quad \underset{s_{2:d}}{\text{minimize}} \sum_{t=1}^{d} \gamma^t R(s_t, a_t) \\ &\text{subject to} \quad s_{t+1} \in T(s_t, a_t), \quad t \in 1 : d-1 \end{aligned} \tag{9.5}$$

Unfortunately, this formulation can result in extremely conservative behavior. If we adapt example 9.10 to model the uncertainty in the motion of the obstacle, the accumulation of uncertainty may become quite large even when planning with a relatively short horizon. One way to reduce the accumulation of uncertainty is to restrict the uncertainty set output by $T(s, a)$ to contain only, say, 95 % of the probability mass. Another issue with this approach is that the minimax optimization problem is often not convex and difficult to solve.

[13] A. Bemporad and M. Morari, "Robust Model Predictive Control: A Survey," in *Robustness in Identification and Control*, A. Garulli, A. Tesi, and A. Vicino, eds., Springer, 1999, pp. 207–226.

### 9.9.3 Multi-Forecast Model Predictive Control

One way to address the computational challenge within the minimax problem in equation (9.5) is to use $m$ forecast scenarios, each of which follows its own deterministic transition function.[14] There are different formulations of this kind of *multi-forecast model predictive control*, which is a type of *hindsight optimization*[15] approach. One common approach is to have the deterministic transition functions depend on the step $k$, $T_i(s, a, k)$, which is the same as augmenting the state space to include depth. Example 9.11 demonstrates how this might be done for a linear–Gaussian model.

We try to find the best sequence of actions against the worst sampled scenario:

$$\begin{aligned} &\underset{a_{1:d}}{\text{maximize}} \quad \underset{i,\, s_{2:d}}{\text{minimize}} \sum_{k=1}^{d} \gamma^k R(s_k, a_k) \\ &\text{subject to} \quad s_{k+1} = T_i(s_k, a_k, k), \quad k \in 1 : d-1 \end{aligned} \tag{9.6}$$

[14] S. Garatti and M. C. Campi, "Modulating Robustness in Control Design: Principles and Algorithms," *IEEE Control Systems Magazine*, vol. 33, no. 2, pp. 36–51, 2013.

[15] It is called hindsight optimization because it represents a solution that is optimizing using knowledge about action outcomes than can only be known in hindsight. E. K. P. Chong, R. L. Givan, and H. S. Chang, "A Framework for Simulation-Based Network Control via Hindsight Optimization," in *IEEE Conference on Decision and Control (CDC)*, 2000.

Example 9.11. Modeling linear-Gaussian transition dynamics in multi-forecast model predictive control.

Suppose we have a problem with linear–Gaussian dynamics:

$$T(\mathbf{s}' \mid \mathbf{s}, \mathbf{a}) = \mathcal{N}(\mathbf{T}_s\mathbf{s} + \mathbf{T}_a\mathbf{a}, \mathbf{\Sigma})$$

The problem in figure 9.9 is linear with no uncertainty, but if we allow the obstacle to move according to a Gaussian distribution at each step, then the dynamics become linear–Gaussian. We can approximate the dynamics using a set of $m$ forecast scenarios, each consisting of $d$ steps. We can pull $m \times d$ samples $\epsilon_{ik} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$ and define the deterministic transition functions:

$$T_i(\mathbf{s}, \mathbf{a}, k) = \mathbf{T}_s\mathbf{s} + \mathbf{T}_a\mathbf{a} + \epsilon_{ik}$$

This problem can be much easier to solve than the original robust problem.

We can also use a multi-forecast approach to optimize the average case.[16] The formulation is similar to equation (9.6), except that we replace the minimization with an expectation and allow different action sequences to be taken for different scenarios with the constraint that the first action must agree:

$$\begin{aligned} \underset{a_{1:d}^{(1:n)}, s_{2:d}^{(i)}}{\text{maximize}} \quad & \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{d}\gamma^k R(s_k^{(i)}, a_k^{(i)}) \\ \text{subject to} \quad & s_{k+1}^{(i)} = T_i(s_k^{(i)}, a_k^{(i)}, k), \quad k \in 1:d-1, i \in 1:m \\ & a_1^{(i)} = a_1^{(j)}, \qquad\qquad i \in 1:m, j \in 1:m \end{aligned} \tag{9.7}$$

This formulation can result in robust behavior without being overly conservative while still maintaining computational tractability. Both formulations in equations (9.6) and (9.7) can be made more robust by increasing the number of forecast scenarios $m$ at the expense of additional computation.

[16] This approach was applied to optimizing power flow policies by N. Moehle, E. Busseti, S. Boyd, and M. Wytock, ''Dynamic Energy Management,'' in *Large Scale Optimization in Supply Chains and Smart Manufacturing*, Springer, 2019, pp. 69–126.

## 9.10 Summary

- Online methods plan from the current state, focusing computation on states that are reachable.

- Receding horizon planning involves planning to a certain horizon and then replanning with each step.
- Lookahead with rollouts involves acting greedily with respect to values estimated using simulations of a rollout policy; it is computationally efficient compared to other algorithms, but there are no guarantees on performance.
- Forward search considers all state-action transitions up to a certain depth, resulting in computational complexity that grows exponentially in both the number of states and the number of actions.
- Branch and bound uses upper and lower bound functions to prune portions of the search tree that will not lead to a better outcome in expectation.
- Sparse sampling avoids the exponential complexity in the number of states by limiting the number of sampled transitions from every search node.
- Monte Carlo tree search guides search to promising areas of the search space by taking actions that balance exploration with exploitation.
- Heuristic search runs simulations of a policy that is greedy with respect to a value function that is updated along the way using lookahead.
- Labeled heuristic search reduces computation by not reevaluating states whose values have converged.
- Open-loop planning aims to find the best possible sequence of actions and can be computationally efficient if the optimization problem is convex.

## 9.11 Exercises

**Exercise 9.1.** Why does branch and bound have the same worst-case computational complexity as forward search?

*Solution:* In the worst case, branch and bound will never prune, resulting in a traversal of the same search tree as forward search with the same complexity.

**Exercise 9.2.** Given two admissible heuristics $h_1$ and $h_2$, how can we use both of them in heuristic search?

*Solution:* Create a new heuristic $h(s) = \min\{h_1(s), h_2(s)\}$ and use it instead. This strictly improves a heuristic search algorithm. Moreover, this new heuristic is guaranteed to be admissible. Both $h_1(s) \geq U^*(s)$ and $h_2(s) \geq U^*(s)$ imply that $h(s) \geq U^*(s)$.

**Exercise 9.3.** Given two inadmissible heuristics $h_1$ and $h_2$, describe a way we can use both of them in heuristic search.

*Solution:* Would could define a new heuristic $h_3(s) = \max(h_1(s), h_2(s))$ to get a potentially admissible or a ''less-inadmissible'' heuristic. It may be slower to converge, but it may be more likely to not miss out on a better solution.

**Exercise 9.4.** Suppose we have a discrete MDP with state space $\mathcal{S}$ and action space $\mathcal{A}$ and we want to perform forward search to depth $d$. Due to computational constraints and the requirement that we must simulate to depth $d$, we decide to generate new, smaller state and action spaces by re-discretizing the original state and action spaces on a coarser scale with $|\mathcal{S}'| < |\mathcal{S}|$ and $|\mathcal{A}'| < |\mathcal{A}|$. In terms of the original state and action spaces, what would the size of the new state and action spaces need to be in order to make the computational complexity of forward search approximately depth-invariant with respect to the size of our original state and action spaces, i.e. $O(|\mathcal{S}||\mathcal{A}|)$?

*Solution:* We need

$$|\mathcal{S}'| = |\mathcal{S}|^{\frac{1}{d}} \qquad \text{and} \qquad |\mathcal{A}'| = |\mathcal{A}|^{\frac{1}{d}}$$

This results in the following complexity:

$$O\left(|\mathcal{S}'|^d |\mathcal{A}'|^d\right) = O\left(\left(|\mathcal{S}|^{\frac{1}{d}}\right)^d \left(|\mathcal{A}|^{\frac{1}{d}}\right)^d\right) = O(|\mathcal{S}||\mathcal{A}|)$$

**Exercise 9.5.** Building upon the previous exercise, suppose now that we want to keep all of the original actions in our action space and only re-discretize the state space. What would the size of the new state space need to be to make the computational complexity of forward search approximately depth-invariant with respect to the size of our original state and action spaces?

*Solution:* The computational complexity of forward search is given by $O\left((|\mathcal{S}||\mathcal{A}|)^d\right)$, which can also be written as $O\left(|\mathcal{S}|^d |\mathcal{A}|^d\right)$. Thus, in order for our coarser state space to lead to forward search that is approximately depth-invariant with respect to the size of our original state and action spaces, we need

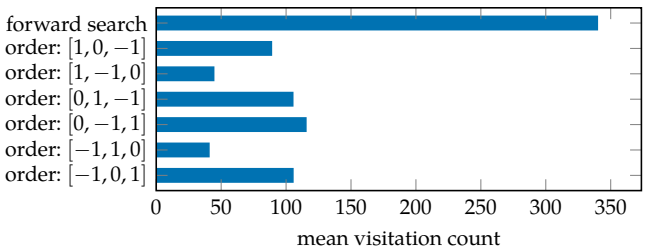
$$|\mathcal{S}'| = \left(\frac{|\mathcal{S}|}{|\mathcal{A}|^{d-1}}\right)^{\frac{1}{d}}$$

This gives us:

$$O\left(|\mathcal{S}'|^d |\mathcal{A}'|^d\right) = O\left(\left[\left(\frac{|\mathcal{S}|}{|\mathcal{A}|^{d-1}}\right)^{\frac{1}{d}}\right]^d |\mathcal{A}|^d\right) = O\left(|\mathcal{S}| \frac{|\mathcal{A}|^d}{|\mathcal{A}|^{d-1}}\right) = O(|\mathcal{S}||\mathcal{A}|)$$

**Exercise 9.6.** Will changing the ordering of the action space cause forward search to take different actions? Will changing the ordering of the action space cause branch and bound to take different actions? Can the ordering of the action space affect how many states are visited by branch and bound?

*Solution:* Forward search enumerates over all possible future actions. It may return different actions if there are ties in their expected utilities. Branch and bound maintains the same optimality guarantee over the horizon as forward search by sorting by upper bound. The ordering of the action space can affect branch and bound's visitation rate when the upper bound produces the same expected value for two or more actions. Below we show this effect on the modified mountain car problem from example 9.2. The plot compares the number of states visited in forward search to that of branch and bound for different action orderings to depth 6. Branch and bound consistently visits far fewer states than forward search, but action ordering can still affect state visitation.




**Exercise 9.7.** Is sparse sampling with $m = |\mathcal{S}|$ equivalent to forward search?

*Solution:* No. While the computational complexities are identical at $O\left(|\mathcal{S}|^d |\mathcal{A}|^d\right)$, forward search will branch on all states in the state space, while sparse sampling will branch on $|\mathcal{S}|$ randomly sampled states.

**Exercise 9.8.** Given an MDP with $|\mathcal{S}| = 10$, $|\mathcal{A}| = 3$, and a uniform transition distribution $T(s' \mid s, a) = 1/|\mathcal{S}|$ for all $s$ and $a$, what is probability that sparse sampling with $m = |\mathcal{S}|$ samples and depth $d = 1$ yields the exact same search tree produced by forward search with depth $d = 1$?

*Solution:* For both forward search and sparse sampling, we branch on all actions from the current state node. For forward search, at each of these action nodes, we branch on all states, while for sparse sampling, we will branch on $m = |\mathcal{S}|$ sampled states. If these sampled states are exactly equal to the state space, that action branch is equivalent to the branch produced in forward search. Thus, for a single action branch we have:

| | |
|---|---|
| the probability the first state is unique | $\frac{10}{10}$ |
| the probability the second state is unique (not equal to the first state) | $\frac{9}{10}$ |
| the probability the third state is unique (not equal to the first or second state) | $\frac{8}{10}$ |
| $\vdots$ | $\vdots$ |

Since each of these sampled states is independent, this leads to the probability of all unique states in the state space being selected with probability

$$\frac{10 \times 9 \times 8 \times \cdots}{10 \times 10 \times 10 \times \cdots} = \frac{10!}{10^{10}} \approx 0.000363$$

Since each of the sampled states across different action branches is independent, the probability that all three action branches sample the unique states in the state space is

$$\left(\frac{10!}{10^{10}}\right)^3 \approx (0.000363)^3 \approx 4.78 \times 10^{-11}$$

**Exercise 9.9.** Given the following tables of $Q(s,a)$ and $N(s,a)$, use the upper confidence bound in equation (9.1) to compute the optimal action for each state with an exploration parameter of $c_1 = 10$ and again for $c_2 = 20$.

| | $Q(s,a_1)$ | $Q(s,a_2)$ |
|---|---|---|
| $s_1$ | 10 | $-5$ |
| $s_2$ | 12 | 10 |

| | $N(s,a_1)$ | $N(s,a_2)$ |
|---|---|---|
| $s_1$ | 27 | 4 |
| $s_2$ | 32 | 18 |

*Solution:* For the first exploration parameter $c_1 = 10$, we tabulate the upper confidence bound of each state-action pair and select the action maximizing the bound for each state:

| | $UCB(s,a_1)$ | $UCB(s,a_2)$ | $\arg\max_a UCB(s,a)$ |
|---|---|---|---|
| $s_1$ | $10 + 10\sqrt{\frac{\log 31}{27}} \approx 13.566$ | $-5 + 10\sqrt{\frac{\log 31}{4}} \approx 4.266$ | $a_1$ |
| $s_2$ | $12 + 10\sqrt{\frac{\log 50}{32}} \approx 15.496$ | $10 + 10\sqrt{\frac{\log 50}{18}} \approx 14.662$ | $a_1$ |

And for $c_2 = 20$, we have:

| | $UCB(s,a_1)$ | $UCB(s,a_2)$ | $\arg\max_a UCB(s,a)$ |
|---|---|---|---|
| $s_1$ | $10 + 20\sqrt{\frac{\log 31}{27}} \approx 17.133$ | $-5 + 20\sqrt{\frac{\log 31}{4}} \approx 13.531$ | $a_1$ |
| $s_2$ | $12 + 20\sqrt{\frac{\log 50}{32}} \approx 18.993$ | $10 + 20\sqrt{\frac{\log 50}{18}} \approx 19.324$ | $a_2$ |