on ending up in a dead end from a given state, it has to sample many deterministic futures, exacting a steep price in the speed of computing a solution.

- **RFF**, although aware of dead ends' existence, does little to avoid them. When, as a result of expansion, its policy graph includes a dead end, the basic RFF version does not modify the policy to avoid the dead end. Rather, it simply excludes the dead end from the policy graph and marks it in order to prevent its re-inclusion in the future. A more sophisticated RFF version performs policy optimization [228] on the solution graph using Bellman backups, which improves its ability to eschew dead ends when possible.
- **HMDPP** detects dead-end states explicitly with the help of a dedicated heuristic. Its approach appears to strike a good balance between the quality of a solution according to the MAXPROB criterion and the efficiency of finding it.

Our discussion of approximating solutions to MAXPROB$_{s_0}$ MDPs with determinization-based algorithms would be incomplete without considering a natural alternative: why not solve MAXPROB problems *optimally*? After all, perhaps solving MAXPROB is easy enough that it can be done reasonably efficiently without resorting to tricks such as determinization? Unfortunately, too little is known about MAXPROB$_{s_0}$ MDPs to answer these questions conclusively. *MAXPROB* is known not to be a subclass of *SSP* with an initial state [137], since in MAXPROB$_{s_0}$ MDPs, improper policies do not accumulate an infinite cost, as the *SSP* definition requires. Rather, *MAXPROB*$_{s_0}$ is a subclass of *GSSP* [137], a type of MDPs with complicated mathematical properties briefly covered in Chapter 7. The most efficient currently known method of optimally solving MAXPROB$_{s_0}$ MDPs is heuristic search, although of a more sophisticated kind [137] than presented in Chapter 4. However, at present its effectiveness is limited by the lack of admissible heuristics for MAXPROB problems. The only nontrivial (i.e., more involved than setting the value of every state to 1) such heuristic known so far is SixthSense [133], which soundly identifies dead ends in the MDP's state space and sets $h_{6S}(s) = 0$ for them. Curiously, computing SixthSense itself heavily relies on domain determinization. We revisit it in some more detail in Section 7.4.3.

6.2 SAMPLING-BASED TECHNIQUES

Both the optimal and the approximate MDP algorithms discussed so far work well as long as the number of outcomes of each action in different states is, on average, small. For a factored MDP, "small" means constant or polynomial in the number of state variables. There are plenty of scenarios for which this is not the case. To illustrate them, we use a toy problem named Sysadmin, introduced in Section 2.5.3. Recall that it involves a network of n servers, in which every running server has some probability of going down and every server that is down has some probability of restarting at every time step. There are n binary variables indicating the statuses of all servers, and at each time step the system can transition to virtually any of the 2^n states with a positive probability. Some MDP specification languages, e.g., RDDL [204], which we saw in the same section, can easily describe such MDPs in a compact form (polynomial in the number of state variables).

What happens to the algorithms we have explored in this book so far if they are run on an MDP with an exponentially sized transition function, as above? Let us divide these techniques into two (non-mutually exclusive) groups, those that compute Q-values via Definition 3.7 (e.g., RTDP) and those that use determinizations (e.g., FF-Replan). Computing a Q-value requires iterating over all of an action's successors, whose number for the MDPs we are considering is exponential in the problem description size. Thus, the first group of methods will need to perform a prohibitively expensive operation *for every state transition*, rendering them impractical.[1] The second group of methods does not fare much better. In the presence of a large transition function, their preprocessing step, generating the determinization, requires an exponential effort as well. Thus, all the machinery we have seen seems defeated by such MDPs.

6.2.1 UCT

UCT [128] is a planning algorithm that can successfully cope even with exponential transition functions. Exponential transition functions plague other MDP solvers ultimately because these solvers attempt to enumerate the domain of the transition function for various state-action pairs, e.g., when summing over the transition probabilities or when generating action outcomes to build determinizations. As a consequence, they need the transition function to be explicitly known and efficiently enumerable. Instead, being a Monte Carlo planning technique [89], UCT only needs to be able to efficiently sample from the transition function and the cost function. This is often possible using an environment simulator even when the transition function itself is exponentially sized. Moreover, the fact that UCT only needs access to the transition and cost functions in the form of a simulator implies that UCT does *not* need to know the transition probabilities (and action costs) explicitly.

As Algorithm 6.2 shows, UCT works in a similar way to RTDP. It samples a number of trajectories, or *rollouts* in UCT terminology, through the state space, and updates Q-value approximations for the state-action pairs the trajectories visit. The rollouts have the length of at most T_{max}, a user-specified cutoff, as is common for RTDP as well. However, the way UCT selects an action in a state, and the way it updates Q-value approximations is somewhat different from RTDP. For each state s, UCT maintains a counter n_s of the number of times state s has been visited by the algorithm. The n_s counter is incremented every time a rollout passes through s. For each state-action pair $\langle s, a\rangle$, UCT maintains a counter $n_{s,a}$ of how many times UCT selected action a when visiting state s. Clearly, for each s, $n_s = \sum_{a\in\mathcal{A}} n_{s,a}$. Finally, for each $\langle s, a\rangle$, UCT keeps track of $\hat{Q}(s, a)$, an approximation of the Q-value of a in s equal to the average reward accumulated by past rollouts after visiting s and choosing a in s (line 30). In every state s, UCT selects an action

$$a' = \operatorname*{argmin}_{a\in\mathcal{A}} \left\{ \hat{Q}(s, a) - C\sqrt{\frac{\ln(n_s)}{n_{s,a}}} \right\} \tag{6.2}$$

[1]The use of ADDs may alleviate the problem of computing Q-values in some, though not all, such problems.

Algorithm 6.2: UCT

```
while there is time left do
    s ← current state
    cumulativeCost[0] ← 0
    maxNumSteps ← 0
    // Sample a rollout of length at most T_max, as specified by the user
    for i = 1 through T_max do
        if s has not been visited before then
            n_s ← 0
            for all a ∈ A do
                n_{s,a} ← 0
                Q̂(s, a) ← 0
            end
        end
        maxNumSteps ← i
        // C ≥ 0 is a user-specified weight of the exploration term
        a' ← argmin_{a∈A} { Q̂(s, a) − C sqrt(ln(n_s) / n_{s,a}) }
        s' ← execute action a' in s
        // cumulativeCost[i] is the cost incurred during the first i steps of the current rollout
        cumulativeCost[i] ← cumulativeCost[i − 1] + C(s, a, s')
        s_i ← s
        a_i ← a'
        s ← s'
        if s ∈ G then
            break
        end
    end
    for i = 1 through maxNumSteps do
        // Update the average Q̂(s_i, a_i) with the total cost incurred in the current rollout
        // after visiting the state-action pair ⟨s_i, a_i⟩
        Q̂(s_i, a_i) ← (n_{s_i,a_i} Q̂(s_i,a_i) + (cumulativeCost[maxNumSteps] − cumulativeCost[i−1])) / (n_{s_i,a_i} + 1)
        n_{s_i} ← n_{s_i} + 1
        n_{s_i,a_i} ← n_{s_i,a_i} + 1
    end
end
return argmin_{a∈A} Q̂(current state, a)
```

The algorithm then samples an outcome of a' and continues the trial.

UCT effectively selects an action in each state based on a combination of two characteristics — an approximation of the action's Q-value ($\hat{Q}(s, a)$) and a measure of how well-explored the action in this state is ($C\sqrt{\frac{\ln(n_s)}{n_{s,a}}}$). Intuitively, UCT can never be "sure" of how well $\hat{Q}(s, a)$ approximates the quality of an action because $\hat{Q}(s, a)$ is computed via sampling and may fail to take into account

some of a's possible effects. Therefore, there is always the danger of $\hat{Q}(s, a)$ failing to reflect some particularly good or bad outcome of a, which could affect the ordering of actions in a state by quality. In other words, UCT needs to always be aware that some actions are not explored well enough. One way to implement this insight is to have UCT from time to time try an action that currently seems suboptimal but has not been chosen for a long time. This is exactly what the exploration term $C\sqrt{\frac{\ln(n_s)}{n_{s,a}}}$ does for UCT. If a state s has been visited many times ($\ln(n_s)$ is large) but a in s has been tried few times ($n_{s,a}$ is small), the exploration term is large and eventually forces UCT to choose a. The constant $C \geq 0$ is a parameter regulating the relative weight of the exploration term and the Q-value approximation [128]. The value of C greatly affects UCT's performance. Setting it too low causes UCT to under-explore state-action pairs. Setting it too high slows down UCT's convergence, since the algorithm spends too much time exploring suboptimal state-action pairs.

The exploration term may appear to make UCT's action selection mechanism somewhat arbitrary. When UCT starts exploring the MDP, this is so indeed; the exploration term encourages UCT to try a lot of actions that, purely based on $\hat{Q}(s, a)$, look suboptimal. However, notice: as the denominator $n_{s,a}$ grows, it is harder and harder for the numerator $\ln(n_s)$ to "keep up the pace," i.e., the growth of the exploration term for $\langle s, a\rangle$ slows down. That is, the more times UCT visits a state-action pair, the smaller this pair's exploration term and the bigger the role of $\hat{Q}(s, a)$ in deciding whether a is selected in s. The longer the algorithm runs, the more its action selection strategy starts resembling RTDP's.

UCT has no termination criterion with solution quality guarantees, but the following result holds regarding its convergence.

Theorem 6.6 In a finite-horizon MDP where all action costs have been scaled to lie in the [0,1] interval and for each augmented state (s, t) the constant C of the exploration term is set to t, the probability of UCT selecting a suboptimal action in the initial state s_0 converges to zero at a polynomial rate as the number of rollouts goes to infinity [128].

Note that this result pertains to finite-horizon MDPs and does not directly apply to the goal-oriented setting. However, if for an SSP_{s_0} MDP it is known that there is a policy that reaches the goal from s_0 within H steps with probability 1, one can set the T_{max} parameter (line 6) to H and expect that UCT will approach a near-optimal policy as the theorem describes.

In an online setting, UCT can be used by giving it some amount of time to select an action in the current state and making it return an action when the time is up. The amount of planning time to allocate for good results is problem-dependent and needs to be determined experimentally. UCT can also greatly benefit from a good initialization of its Q-value approximations $\hat{Q}(s, a)$. Well-tuned UCT versions are responsible for several recent, as of 2012, advances in computer game playing (e.g., bringing AI to a new level in 9×9 Go [93]). It is also the basis of a successful planner for finite-horizon MDPs called PROST [124; 204]. The family of Monte Carlo planning algorithms to which UCT belongs is a vast research area, most of which is beyond the scope of this book, and we encourage the reader to use external sources to explore it [89].