IEOR 265 - Lecture 6
Approximation on Value Space

1 Approximate Dynamic Programming

As always, we start be stating the DP problem with disturbances:

$$J^*(x_0) = \min_{\pi \in \Pi} \mathbb{E}_w \big[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k) \big]$$

$$x_{k+1} = f_k(x_k, u_k(x_k), w_k), \forall k \in \{0, 1, ..., N-1\} \tag{1}$$

where the expectation on the right-hand side is taken w.r.t. the joint probability distribution of $(w_0, ..., w_{N-1})$.

As well as the DP recursion:

$$J_N(x_n) = g_N(x_N)$$

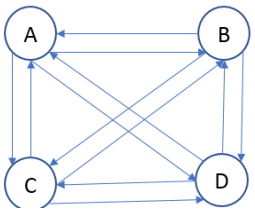
$$J_k(x_k) = \min_{u_k \in U_k(x_k)} \big\{ \mathbb{E}_{w_k} \big[g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k)) \big] \big\}, \forall k \in \{0, ..., N-1\} \tag{2}$$

Practical DP problems are very challenging due to the explosive nature of DP recursion Eq(2): We need to perform both expectation and minimization for **every** possible state x_k and stage k; In addition if the control set is given by finite elements, then the minimization can be performed only by exhaustive enumeration of all possible state-control pairs (x_k, u_k); In addition, the expectation may not be able to be obtained in close form if the disturbances follow some complex distribution; Lastly the horizon length N may be prohibitively large, or even infinite.

And yet, problems like: Autonomous driving, Robotic flying, Chess/Go playing softwares, Gaming AI, robotic object manipulation are just a few examples of hard and complex DP problems where a significant amount of success (although several challenges still lay ahead) has been observed. Most if not all of such problems, bypass the **curse of dimensionality** of DP by relying on Approximations.

As an illustrative example let's consider the Traveling Salesman Problem (TSP), a known "hard" problem (called NP-Hard in the literature): A salesman wish to visit every city from a given set of cities such each city is visited exactly once and the total tour length is minimized. This problem can be formulated as a graph problem. For instance consider following graph and the associated arc length matrix presented in figure 1.

We only have four nodes, so it is easy to solve the problem by visual inspection, leading to the smallest tour: ABDC. We can frame the TSP problem in as



Distances: $\begin{bmatrix} 0 & 5 & 1 & 15 \\ 5 & 0 & 20 & 4 \\ 1 & 20 & 0 & 3 \\ 15 & 4 & 3 & \end{bmatrix}$

Figure 1: TSP problem with each node representing a city and the associated arc length matrix.

a deterministic DP (so as we saw, a Shortest Path problem): We let the states x_k be the partial tours containing k cities (so for example $x_k = (c_0, c_1, ..., c_k)$, so it is a partial tour starting from the initial city c_0 and visiting $k-1$ more cities). At state x_k, we let u_k be all the available cities to be visited from x_k that have not been visited yet. Then we write the associated "trellis diagram":

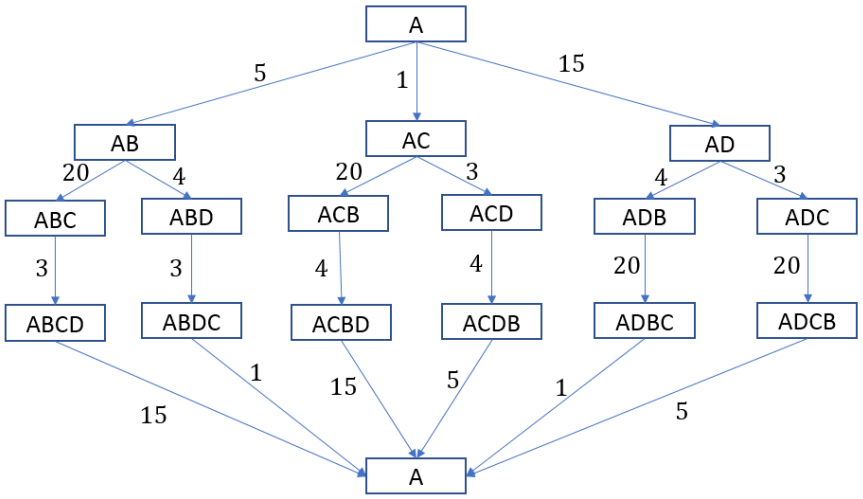


Figure 2: Schematic figure the Shortest Path representation of the TSP problem.

Note how for 4 cities, we have a total of 16 states (excluding the "dummy" terminal node). Now try to imagine if we had 40000 cities, how many states would this DP formulation require? The same line of argument applies from problems like Chess, Go, etc.

1.1 Q-factor reformulation

As we study Approximate DP methods, it is often convenient to re-write the DP recursion in the convenient form of **Q-factors** (or Q-functions). Namely instead of expressing the cost-to-go function $J_k(x_k)$ as a function solely of the state x_k, we look at the pairs os state-controls (x_k, u_k). We can define the Q-factors as follows:

$$Q_k^*(x_k, u_k) = \mathbb{E}_{w_k}\left[g_k(x_k, u_k, w_k) + J_{k+1}^*(f_k(x_k, u_k, w_k))\right], \forall k \in \{0, ..., N-1\} \tag{3}$$

The Q-factors have a nice intuitive definition: They are the total cost at state x_k given that we select the control u_k in the *current* time period, and *then* we proceed to follow the optimal policy starting from period $k+1$.

Given this definition, it follows directly, that the optimal value function $J_k^*(x_k)$ at state x_k and time period k, is given by:

$$J_k^*(x_k) = \min_{u_k \in U_k(x_k)} \left\{ Q_k^*(x_k, u_k) \right\}, \forall k \in \{0, ..., N-1\} \tag{4}$$

And we can write a similar backwards recursion defining each Q-factor:

$$Q_k^*(x_k, u_k) = \mathbb{E}_{w_k} \left[g_k(x_k, u_k, w_k) + \min_{u_{k+1} \in U_{k+1}(f_k(x_k, u_k, w_k))} \left\{ Q_{k+1}^*(f_k(x_k, u_k, w_k), u_{k+1}) \right\} \right] \quad \forall k \in \{0, ..., N-1\} \tag{5}$$

As we study different approximation methods, sometimes it will be more suitable to work with the value function themselves $J_k(x_k)$, and other times with the Q-factors $Q_k^*(x_k, u_k)$. Our presentation will follow closely the text [1].

2 Approximate DP and Lookahead

2.1 1-step Lookahead

The first type of approximation we will analyze is the approximation on value space where we replace the cost-to-go function (also called the value function) $J_{k+1}(x_{k+1})$ by some approximate function $\tilde{J}_{k+1}(x_{k+1})$. Therefore we can define a suboptimal admissible policy $\tilde{\pi} = (\tilde{\mu}_k(x_0), ..., \tilde{\mu}_k(x_{N-1}))$ as follows:

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left\{ \mathbb{E}_{w_k} \left[g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \right] \right\}, \forall k \in \{0, ..., N-1\} \tag{6}$$

We can use the following as the approximation:

$$\tilde{J}_{k+1}(x_{k+1}) = \min_{u_{k+1} \in U_{k+1}(x_{k+1})} \mathbb{E}_{w_{k+1}} \left[g_{k+1}(x_{k+1}, \mu_{k+1}(x_{k+1}), w_{k+1}) + \tilde{J}_{k+2}(x_{k+2}) \right] \tag{7}$$

Eq(6) and (7) form the 1-step lookahead approximation method. The reason it's called a "lookahead" method, is because we use as the approximate value function $\tilde{J}_{k+1}(x_{k+1})$ the result of a 1-stage DP problem, starting from $x_k + 1$, where the terminal cost is "replaced" by yet another approximate function $\tilde{J}_{k+2}(x_{k+2})$. This terminal cost function, represents the approximate cost of the future, "from x_{k+2} onwards". Intuitively, this means that if we are the *current* state x_k, we look ahead a single time period to obtain an approximation of the cost-to-go from the next state x_{k+1}. Clearly this is simpler than executing the exact DP recursion (Eq(1)-(2)), since there we require to look ahead all time periods until the end, instead of just one.

2.2 Multistep Lookahead

Now, let's follow up the preceding argument by increasing the lookahead window from 1 time period, to l time periods. Then we can use the following as the

approximation for $\tilde{J}_{k+1}(x_{k+1})$:

$$\tilde{J}_{k+1}(x_{k+1}) = \min_{(\mu_{k+1},\ldots,\mu_{k+l-1})} \mathbb{E}_{w_{k+1},\ldots,w_{k+l-1}} \left[\sum_{i=k+1}^{k+l-1} g_i(x_i, \mu_i(x_i), w_i) + \tilde{J}_{k+l}(x_{k+l}) \right] \tag{8}$$

where $\tilde{J}_{k+l}(x_{k+l})$ is another (hopefully simpler) approximate value function. Note that this approximation is essentially a $(l-1)$-stage DP, with a terminal cost approximation. Hence the entire multistep lookahead approximation with lookahead window l, can be summarized as a l-stage DP problem (the very first stage plus the $(l-1)$ lookahead problem):

$$\min_{u_k,\mu_{k+1},\ldots,\mu_{k+l-1}} \mathbb{E}_{w_{k+1},\ldots,w_{k+l-1}} \left[g_k(x_k, u_k, w_k) + \sum_{i=k+1}^{k+l-1} g_i(x_i, \mu_i(x_i), w_i) + \tilde{J}_{k+l}(x_{k+l}) \right] \tag{9}$$

where a key observation is that once we are stage k (the *current* stage), after solving Eq(9), we obtain the control $\tilde{u}_k$, from stage k, and we obtain a truncated policy $(\tilde{\mu}^{k+1}, ..., \tilde{\mu}^{k++l-1})$ out of the $(l-1)$-stage DP problem from Eq(8). In the multistep lookahead minimization, this truncated policy is *discarded* and we only keep the control $\tilde{u}_k$, and we let the function $\tilde{\mu}_k(x_k) = \tilde{u}_k$.

Ideally, we would choose the lookahead window l, "large enough" so that we can use very simple terminal cost approximations. For example:

$$\tilde{J}_{k+l}(x_{k+l}) \equiv 0, \text{ or } \tilde{J}_{k+l}(x_{k+l}) = g_N(x_{k+l}) \tag{10}$$

And then, once the suboptimal control is computed via $\tilde{\mu}_k(x_k)$, we can apply it, and proceed to the next state x_{k+1}, via the dynamics:

$$x_{k+1} = f_k(x_k, \tilde{\mu}_k(x_k), w_k) \tag{11}$$

and re-solve the multistep lookahead minimization Eq(9) again, but now with the initial condition being x_{k+1} instead of x_k. This approach is called the **Rolling Horizon approach** and is often used when the minimization of Eq(9) can be done for sufficiently large window l.

One might be tempted to conclude that by increasing the lookahead window l, the performance of the suboptimal policy will improve. That is not true, however, and this highlights the pitfall of multistep lookahead: We must be very careful in selecting the lookahead window and whether or not it is suitable for a particular application. A simple example to highlight this fact is given by the simple shortest path problem in the figure below:

where the goal is to go form x_0 to x_4 via the path with the shortest length. Note that in this simple problem the DP boils down to a single decision on the stage 0: either we go up (which is the optimal decision), or we go down. Suppose that we use as a terminal cost approximation $\tilde{J}_{k+l})(x_{k+l}) \equiv 0$, that is simply assign a "zero cost" to the future after the lookahead window l. If we use a 2-step lookahead, that leads to selecting the optimal path. However, if we use a 3-step lookahead, we end up selecting the wrong option. This example highlights that a larger lookahead does not necessary imply a better performance of the suboptimal policy.

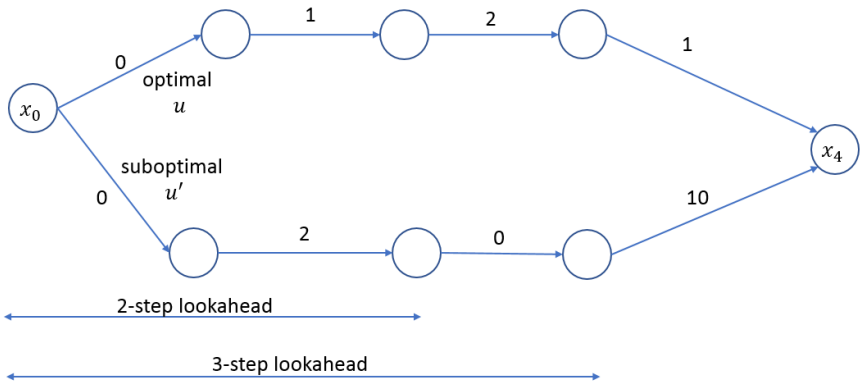


Figure 3: Schematic figure for a 4-stage deterministic DP, framed a Shortest Path Problem

2.3 Partially Deterministic Lookahead

The multistep lookahead is a very flexible method and allows for a myriad of extensions and incrementation Here, we will present one of such extensions which ties well with the deterministic shortest path formulations we covered in the first few lectures. Consider again the multistep lookahead problem with lookahead window l:

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left\{ \mathbb{E}_{w_k} \left[g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \right] \right\}, \forall k \in \{0, ..., N-1\} \tag{12}$$

$$\tilde{J}_{k+1}(x_{k+1}) = \min_{(\mu_{k+1}, ..., \mu_{k+l-1})} \mathbb{E}_{w_{k+1}, ..., w_{k+l-1}} \left[\sum_{i=k+1}^{k+l-1} g_i(x_i, \mu_i(x_i), w_i) + \tilde{J}_{k+l}(x_{k+l}) \right] \tag{13}$$

We will apply the *Certainty Equivalence* principle to replace all the disturbances $(w_{k+1}, ..., w_{k+l})$ in the expectation of Eq(13) by their average ("typical") values $(\bar{w}_{k+1}, ..., \bar{w}_{k+l})$, where:

$$\bar{w}_{k+1} = \mathbb{E}[w_{k+1} | x_{k+1}, u_{k+1}] \tag{14}$$

By doing so, Eq(13) becomes essentially a $l-1$ stage deterministic DP, which can be framed as a (deterministic) Shortest Path Problem. We can then leverage efficient forward-type DP algorithms to solve Eq(13) for each state x_{k+1}. A key observation is that the *current* disturbance w_k is still considered in full stochasticity, hence we still need to perform the expectation on Eq(9). But the trade-off is clear: We need to compute only one expectation instead of a (potentially) very large number of expectations.

The partially Deterministic Lookahead can be nicely represented in figure 4, where the transition from x_k to x_{k+1} is considered over the expectation w.r.t. w_k, but then from each possible x_{k+1} the problem becomes a shortest path problem, which can be represented by a Lookahead Tree (which is very similar to the "Trellis Diagram"). Lastly at the last stage of lookahead window, stage $k+l$, we add a terminal cost approximation $\tilde{J}_{k+l}(x_{k+l})$.

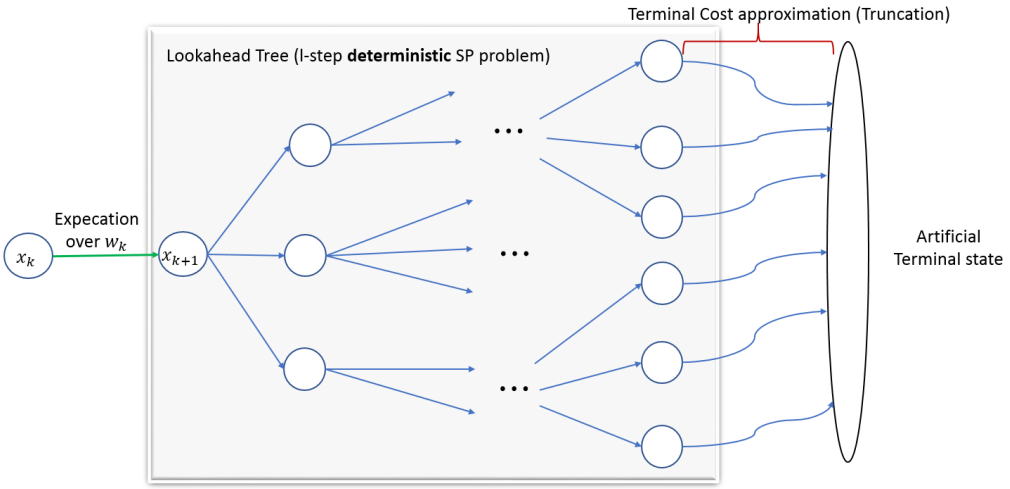


Figure 4: Schematic figure for lookahead DP, framed a Shortest Path Problem

2.4 Scenario-based Lookahead

We can further extended the Partially Deterministic Lookahead, by observing that the "typical" sequence of values $(\bar{w}_{k+1}, ..., \bar{w}_{k+l})$ is just a single sequence. Suppose we have in our disposal a simulator that is able to generate samples of the disturbance starting from each state x_{k+1}:

$$w^s(x_{k+1}) = (w^s_{k+1}, ..., w^s_{k+l-1}), s \in \{1, .., S\}$$

where S is the total number of sequences and each sequence is called a *Scenario*. Then, we can define the total scenario cost $C_s(x_{k+1})$ as the solution of the deterministic lookahead problem under each scenario s:

$$C_s(x_{k+1}) = \min_{(u_{k+1},...,u_{k+l-1})} \Big\{ \sum_{i=k+1}^{k+l-1} g_i(x_i, u_i, w^s_i) + \tilde{J}_{k+l}(x_{k+l}) \Big\} \tag{15}$$

where the dynamics become now deterministic for each scenario:

$$x_{i+1} = f_i(x_i, u_i, w^s_i), \forall i \in \{k+1, ..., k+l-1\} \tag{16}$$

Then we can compute a sample average approximation (SAA):

$$\tilde{J}_{k+1}(x_{k+1}) = \sum_{s=1}^{S} r_s C_s(x_{k+1}) \tag{17}$$

where $r = (r_1, ..., r_S)$ is some probability distribution vector, representing the relative "frequencies" of each scenario in the simulation. In some contexts, r is also called the scenario weight vector. For example, if we obtain i.i.d. unique scenario samples, then we can let $r_s = 1/S$ for every scenario $s \in \{1, ..., S\}$. Typically, these probabilities are closely tied to the simulation scheme that was used to generate such scenarios. We will postpone for now the discussion on *how* such sequences can be obtained.

The creation of scenarios can be nicely represented by a Scenario Tree (such representation is often found in the field of Stochastic Programming):

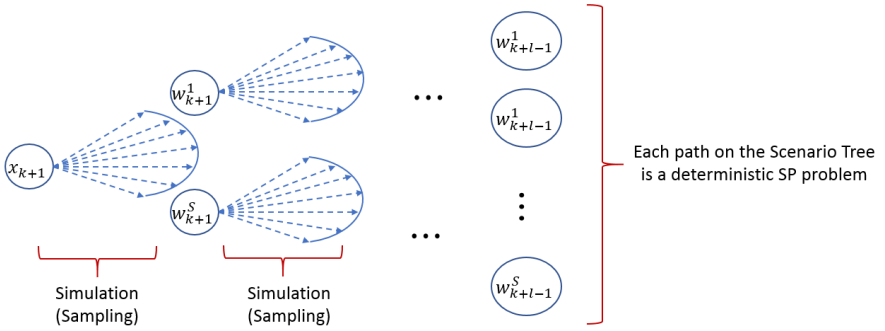


Figure 5: Scenario Tree representation: starting from x_{k+1}, we simulate, via sampling, diferent values for w_{k+1}, and then subsequently for $w_{k+2}, ...w_{k+l-1}$. Each path from the root note to the leaves form a scenario.

3 Rollout Algorithm

3.1 Deterministic Case

Now we will introduce another method to generate the approximate value function $\tilde{J}_{k+1}(x_{k+1})$. Let's first focus on the deterministic case: there are no disturbances and the entire DP problem is deterministic. We write here again the the suboptimal policy $\tilde{\pi} = (\tilde{\mu}_0, ..., \tilde{\mu}_{N-1})$ given by:

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \{g_k(x_k, u_k) + \tilde{J}_{k+1}(f_k(x_k, u_k))\}, \forall k \in \{0, ..., N-1\} \quad (18)$$

Now suppose we have another policy $\hat{\pi} = (\hat{\mu}_0, ..., \hat{\mu}_{N-1})$ and that we execute it starting at stage $k+1$. This policy is called the *Base Policy* (or the *Base Heuristic*). Now we will use the base policy $\hat{\pi}$ to bring the system from state x_{k+1} to state x_N and we compute the accumulated cost and let *that* be the approximate value function:

$$\tilde{J}_{k+1}(x_{k+1}) = \sum_{i=k+1}^{N-1} g_i(x_i, \hat{\mu}_i(x_i)) + g_N(x_N) \quad (19)$$

Now suppose the horizon N is very large, then it may not be possible to execute the base policy from stage $k+1$ to the end stage N. Then we stop the execution at some truncated stage $m < N$ and use some terminal cost approximation $\tilde{J}_m(x_m)$ to represent the "future" from m to N. Therefore, the approximate value function become:

$$\tilde{J}_{k+1}(x_{k+1}) = \sum_{i=k+1}^{m-1} g_i(x_i, \hat{\mu}_i(x_i)) + \tilde{J}_m(x_m) \quad (20)$$

This sequence of states and controls $(x_{k+1}, \hat{\mu}_{k+1}(x_{k+1}), ..., \hat{\mu}_{m-1}(x_{m-1}), x_m)$ is called a *rollout* (or a *rollout trajectory*). The name "rollout" is used pervasively in the approximate DP and RL literature, and it often means different, but related concepts. Here we use the name as the sequence of states and controls obtained by using a base policy (heuristic) beginning at some state x_{k+1}

and terminating either at the last stage N or at some truncated state $m < N$ (in the latter case we also use a terminal cost approximation).

The Rollout algorithm can be represented by a figure:

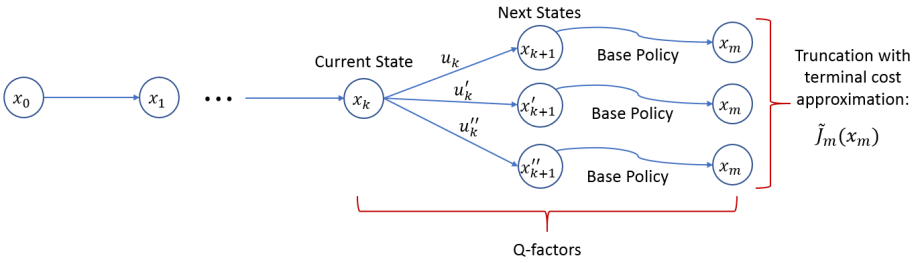


Figure 6: Schematic figure for the Rollout Algorithm

And we can define the **Rollout Policy** as:

$$\tilde{\mu}_k(x_k) = \arg \min_{u_k \in U_k(x_k)} \left\{ \tilde{Q}_k(x_k, u_k) \right\}, \forall k \in \{0, ..., N-1\} \tag{21}$$

$$\tilde{Q}_k(x_k, u_k) = g_k(x_k, u_k) + \tilde{J}_{k+1}(f_k(x_k, u_k)), \forall k \in \{0, ..., N-1\}$$

where we opted to use the Q-factor representation to highlight the fact that in the Rollout algorithm, we execute the base policy after each state x_{k+1} that can be reach from x_k using any of the controls $u_k \in U_k(x_k)$.

3.1.1 Example: Traveling Salesman Problem

We provide a simple example to provide a concrete application of the Rollout Algorithm. Recall the Traveling Salesman Problem, where of the salesman is to traverse every single city exactly one, with minimal tour length. The simple *Nearest-Neighbor Heuristic* can be seen as a special case of the Rollout Algorithm, where the Base Policy adds a new city to a partial tour that greedily minimize the aggregated cost and does not form a cycle.

In the DP formulation, $x_k = (c_0, ..., c_k)$ is a partial tour with k cities. The heuristic adds a new city c_{k+1} to the partial tour if the arc cost $g(c_k, c_{k+1})$ is minimized over all cities $c_{k+1} \neq c_0, ..., c_k$. Thus, at stage k, the heuristic expands the partial tour by forming $(c_0, ..., c_k, c_{k+1})$.

If we were to apply the Rollout Algorithm to the Traveling Salesman Problem, we can use the Nearest-Neighbor heuristic as our base policy. Then, at each state x_k, we apply the heuristic to every possible state x_{k+1} than can be reached from x_k (in this case for every possible city that can be added to the partial tour x_k). This is illustrated in figure 7.

Every rollout trajectory in the figure is a suboptimal but valid tour of cities. Then the Rollout Algorithm picks the best option found from x_k and adds the associated city to the partial tour. After obtaining the increment partial x_{k+1} we re-do the process again, essentially "rolling the horizon forward".

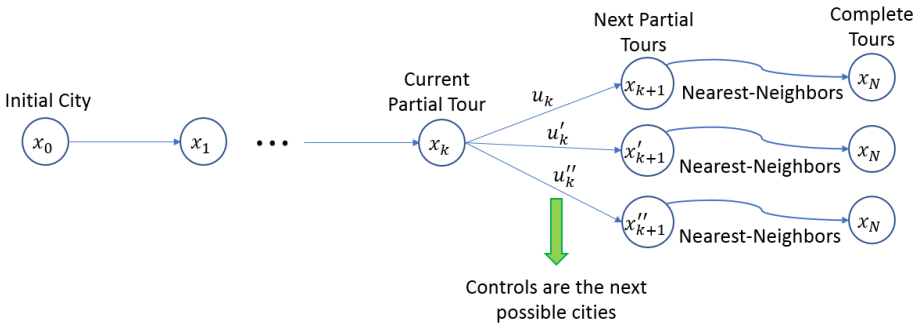


Figure 7: Schematic figure for the Rollout Algorithm applied to the TSP problem with the Nearest Neighbor Heuristic used as a base policy.

3.1.2 Policy Improvement

A key question in the Rollout Algorithm is whether the new rollout policy $\tilde{\pi}$ is better than the base policy $\hat{\pi}$. This can be achieved if the base policy is *Sequentially Improving*, that is:

$$\tilde{Q}_k(x_k, u_k) = g_k(x_k, u_k) + \tilde{J}_{k+1}(f_k(x_k, u_k)), \forall k \in \{0, ..., N-1\} \tag{22}$$

$$\tilde{J}_{k+1}(x_{k+1}) = g_N(x_N) + \sum_{i=k+1}^{N-1} g_i(x_i, \hat{\mu}_i(x_i)) \tag{23}$$

and

$$\min_{u_k \in U_k(x_k)} \{\tilde{Q}_k(x_k, u_k)\} \leq \tilde{J}_k(x_k) \tag{24}$$

which can be stated in words: " the minimal approximate Q-factor at x_k is smaller than the cost-to-go of the base policy executed at x_k".

Suppose that the base policy is sequentially improving. We want to show that:

$$J_{k,\tilde{\pi}}(x_k) \leq \tilde{J}_k(x_k) \tag{25}$$

In words: "the cost-to-go of the rollout policy at x_k is smaller than the cost-to-go of the base policy at x_k". The expression on the right-hand side of Eq(25) is the approximate value function generated by the base policy starting at state x_k. The expression on the left-hand side is the value function that is the *outcome* of the Rollout Algorithm, which can be expressed as follows:

$$\tilde{J}_{k,\tilde{\pi}}(x_k) = g_k(x_k, \tilde{\mu}(x_k)) + \tilde{J}_{k+1}(f_k(x_k, \tilde{\mu}(x_k))), \forall k \in \{0, ..., N-1\} \tag{26}$$

so it is the sum of the *current* cost $g_k(x_k, \tilde{\mu}(x_k))$ plus the approximate cost-to-go $\tilde{J}_{k+1}(x_{k+1})$ by using the base policy starting from the state $x_{k+1} = f_k(x_k, \tilde{\mu}(x_k))$.

If the inequality Eq(25) holds, we say that the rollout policy achieves **policy improvement**. We proceed to show that if the base policy is sequentially improving then the rollout policy achieves policy improvement. We do it by induction on the stages. Let $k = N$, then it follows $J_{N,\tilde{\pi}} = \tilde{J}_N = g_N$. Now,

assume it holds for $k+1$. For any state x_k, let $\hat{u}_k$ be the control value computed by the base policy for state x_k. Then we have:

$$\tilde{J}_{k,\tilde{\pi}}(x_k) = g_k(x_k, \tilde{\mu}(x_k)) + \tilde{J}_{k+1}(f_k(x_k, \tilde{\mu}(x_k))) \leq \tag{27}$$

$$g_k(x_k, \tilde{\mu}(x_k)) + \tilde{J}_{k+1}(f_k(x_k, \tilde{\mu}(x_k))) = \tag{28}$$

$$\min_{u_k \in U_k(x_k)} \left\{ g_k(x_k, u_k) + \tilde{J}_{k+1}(f_k(x_k, u_k)) \right\} = \tag{29}$$

$$\min_{u_k \in U_k(x_k)} \{\tilde{Q}_k(x_k, u_k)\} \leq \tag{30}$$

$$\tilde{J}_k(x_k) \tag{31}$$

where the first equality is applying the DP equation to the rollout policy $\tilde{\pi}$; the first inequality comes from the induction hypothesis; the second equality holds by definition of the rollout algorithm (Eq(18)); The third equality follows from the definition of the Q-factors; Lastly, the second inequality follows from the assumption that the base policy is sequentially improving.

3.1.3 Using multiple base policies

We observe that the rollout algorithm is amenable to the use of many base policies. Suppose we have m different base policies and at a given state x_{k+1}, the m'th base policy produces the following rollout trajectory:

$$\hat{T}^m_{k+1} = \{x_{k+1}, \hat{u}^m_{k+1}, x_{k+2}, \hat{u}^m_{k+2}, ..., \hat{u}^m_{N-1}, \hat{x}^m_N\} \tag{32}$$

and corresponding cost $C(\hat{T}^m_{k+1})$, where

$$\hat{T}^m_{k+1} = \sum_{i=k+1}^{N-1} g_i(x_i, \hat{u}^m_i) + g_N(x_N) \tag{33}$$

Then we can define a *superheuristic* that assign the minimum cost to the approximate value function $\tilde{J}_{k+1}(x_{k+1})$:

$$\tilde{J}_{k+1}(x_{k+1}) = \min_m \{C(\hat{T}^m_{k+1})\}$$

and it follows that if every base policy is sequentially improving then the superheuristic is also sequentially improving (we leave this verification as an exercise).

3.1.4 Deterministic Tree Search

Now let's combine the following items together for the deterministic case:

1. Multistep lookahead
2. Rollout Policy
3. Rollout truncation with terminal cost approximation

The idea is to use a multistep lookahead DP formulation where we selectively use a base policy to provide approximation for the value functions. If the base policies are sequentially improving, then the policy that comes out of the

lookahead problem will achieve policy improvement. Then starting from x_k, we can write the lookahead problem by explicitly considering all possible states in the lookahead window that can be reached from x_k. This leads to a Tree Search framework, which is a powerful and flexible approximation framework for DP, specially when equipped with Deep Neural Networks, as we shall see.

The Deterministic Tree Search framework is best represented by a figure:

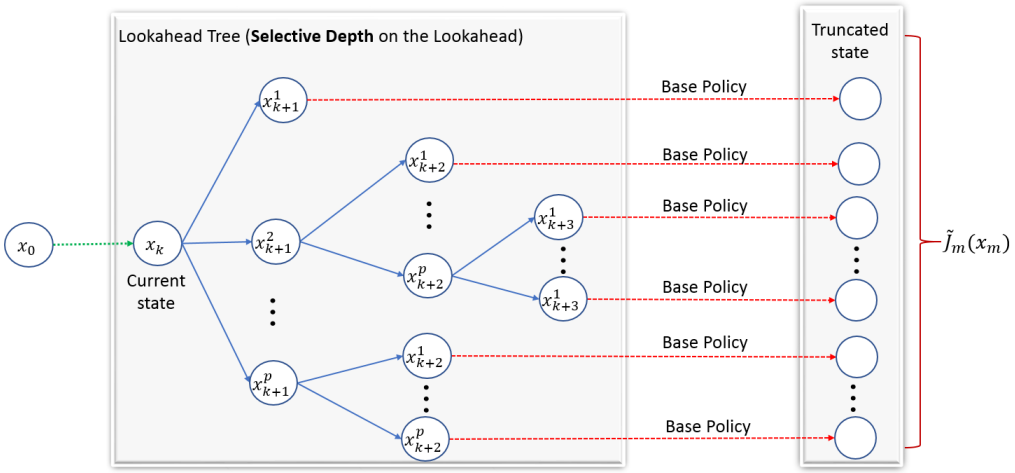


Figure 8: Schematic figure Deterministic Tree Search.

We observe the fact that the multistep lookahead tree is not symmetric and it has *selective depth*. This allows great flexibility in performing the lookahead. For each stage $i \geq k+1$ and each state x_i we can either perform the lookahead minimization (Eq(7)), which if $i < k+l$ would require creating the children nodes x_{i+1} out of x_i; or we can execute the base policy starting from x_i and going until the truncated state x_m (and using a terminal cost approximation $\tilde{J}_m(x_m)$).

The intuition behind this is that if we are able to find "smart" ways to select which nodes in the lookahead tree are worth computing the lookahead minimization and which nodes we just execute the base policy, we can avoid the exhaustive enumeration of all possible combinations of states and controls (essentially mitigating the curse of dimensionality).

In addition, this formulations highlights the central idea between **exploration and exploitation**: One needs to find a "good" balance between the depth of the lookahead (exploitation) and the breadth of the tree (checking as many different states and stages as possible, i.e.: exploration).

As an example, we can define simple pruning rule: at some state x_{i+1} inside the lookahead window, we compute a scoring function $S_{i+1}(x_{i+1})$. If this scoring function is above some threshold α_{i+1} then we say that the score is "bad" and we execute the base policy to obtain an approximate value function $\tilde{J}_{i+1}(x_{i+1})$ starting from x_{i+1}. If it falls below the threshold, then we say that the score is "good" and we expand the lookahead tree by considering all possible states x_{i+2} that can be reached from x_{i+1}. For instance, we can use the 1-step lookahead

DP problem as the scoring function. Namely:

$$S_{i+1}(x_{i+1}) = \min_{u_{i+1} \in U_{i+1}(x_i+1)} \left\{ g_{i+1}(x_{i+1}, u_{i+1}) + \tilde{J}_{i+2}(f_{k+1}(x_{k+1}, u_{i+1})) \right\}$$

$$\tilde{J}_{i+2}(x_{i+2}) = \tilde{J}_m(x_m) + \sum_{j=i+2}^{m-1} g_j(x_j, \hat{\mu}_i(x_j))$$

where we use the base policy to compute $\tilde{J}_{i+2}(x_{i+2})$.

Lastly, we note that we can use many different base policies: for example we can use one policy to obtain the rollout trajectories from the leaf nodes of the lookahead tree to the truncated stage m; and use another different base policy in the computation of the scoring functions. This allows great flexibility in handling different practical DP problems.

3.2 Stochastic Case

We can extend the analysis of the previous subsection to the stochastic case. Here, instead of the sequentially improving property for the base policy, we assume another related property: We assume that base policy is sequentially consistent. A base policy is sequentially consistent if the sequence of states $(x_k, ..., x_m)$ generated by the policy starting from stage k also produces the sequence $(x_{k+1}, ..., x_m)$ if it starts from stage $k+1$. In words: the policy stays the course and obey the bellman principle of optimality (we refer to the first lecture notes). Then we can again prove that:

$$J_{k,\tilde{\pi}}(x_k) \leq \tilde{J}_k(x_k) \tag{34}$$

arguing by induction: The base case is the same. Now suppose it holds for $k+1$. Then we can write:

$$\tilde{J}_{k,\tilde{\pi}}(x_k) = \mathbb{E}_{w_k}\left[g_k(x_k, \tilde{\mu}(x_k), w_k) + \tilde{J}_{k+1}(f_k(x_k, \tilde{\mu}(x_k), w_k)) \right] \leq \tag{35}$$

$$\mathbb{E}_{w_k}\left[g_k(x_k, \tilde{\mu}(x_k), w_k) + \tilde{J}_{k+1}(f_k(x_k, \tilde{\mu}(x_k), w_k)) \right] = \tag{36}$$

$$\min_{u_k \in U_k(x_k)} \left\{ \mathbb{E}_{w_k}\left[g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \right] \right\} \leq \tag{37}$$

$$\mathbb{E}_{w_k}\left[g_k(x_k, \hat{\mu}_k(x_k), w_k) + \tilde{J}_{k+1}(f_k(x_k, \hat{\mu}_k(x_k), w_k)) \right] = \tilde{J}_k(x_k) \tag{38}$$

where for the last inequality follows from the sequentially consistency property.

Now the key challenge in the stochastic case is how to execute the base policy $\hat{\pi}$ in order to obtain the rollout trajectories. We do so by using simulation and sampling: Namely starting from x_k, we compute:

$$x_{i+1} = f_i(x_i, \hat{\mu}_i(x_i), w_i), \, \forall i = k+1, ..., N-1 \tag{39}$$

for sampled sequences $(w_k, ..., w_{N-1})$ of disturbances, like we did in the scenario-based lookahead. We can generate, say, S of such sequences of disturbances. Then we can use the sample average to approximate the expected Q-factors:

$$\tilde{Q}_k(x_k, u_k) = \mathbb{E}_{w_k}[g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, \hat{\mu}_k(x_k), w_k))] \tag{40}$$

$$\tilde{Q}_k(x_k, u_k) \approx \sum_{s=1}^{S} r_s \big(g_k(x_k, u_k, w_k^s) + \tilde{J}_{k+1}(f_k(x_k, \hat{\mu}_k(x_k), w_k^s)) \big) \tag{41}$$

where $r = (r_1, ..., r_S)$ again is the probability distribution of the scenarios ("weights" of each scenario in the simulation). Then the **Rollout Policy** $\tilde{\pi} = (\tilde{\mu}_0(x_0), ..., \tilde{\mu}_{N-1}(x_{N-1}))$ in the stochastic case becomes:

$$\tilde{\mu}_k(x_k) \in \min_{u_k \in U_k(x_k)} \Big\{ \sum_{s=1}^{S} r_s \big(g_k(x_k, u_k, w_k^s) + \tilde{J}_{k+1}(f_k(x_k, \hat{\mu}_k(x_k), w_k^s)) \big) \Big\}, \; \forall k \in \{0, ..., N-1\} \tag{42}$$

3.2.1 Monte Carlo Tree Search (MCTS)

Now we can adapt the Deterministic Tree Search into the stochastic case, leading to the famous **Monte Carlo Tree Search** (MCTS) framework. The idea is still the same: We establish the lookahead tree with *selective depth*, but now for each state transition we perform sample averaging to compute the expected values necessary to obtain the Q-factors. At each leaf of the lookahead tree, we again simulate rollout trajectories (using sampling to obtain the disturbances) from the leaf nodes to the truncated state $m < N$; and lastly, we use a terminal cost approximation $\tilde{J}_m(x_m)$ at the truncated state.

We can represent the MCTS framework by the following figure:

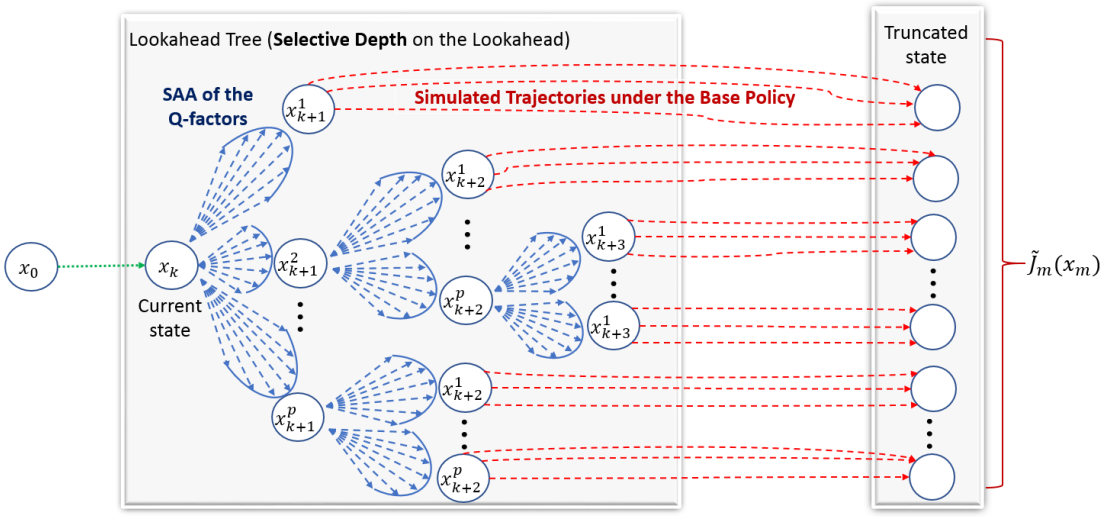


Figure 9: Schematic figure of the Monte Carlo Tree Search: Note that now, we perform sample averaging to compute the Q-factors in the lookahead tree, where for every pair (x_k, u_k) we simulate several future states x_{k+1}, obtain their approximate cost to go $\tilde{J}_{k+1}(x_{k+1})$ and then perform a sample average approximation to obtain the approximate Q-factors $\tilde{Q}(x_k, u_k)$. The approximate function $\tilde{J}_{k+1}(x_{k+1})$ itself is given either by the lookahead minimization or by execution, via simulation, of the base policy starting from x_{k+1}.

The same issues raised before appear here as well: We need to develop pruning rules that now account for the stochasticity of the disturbances; We

need to provide the base policies to obtain the simulated rollouts; We need to provide a suitable terminal cost approximation on the truncated state; We need to specify how the simulation is done, and how can we mitigate simulation error and variance.

As we shall see, the machinery of Deep Neural Networks will play a key role in answering those needs. Success cases of MCTS, such as AlphaGo[2] and AlphaZero [3], rely on the MCTS framework equipped with special deep neural networks that provide approximations for the scoring functions, the base policies and the simulation/sampling requirements.

References

[1] D. P. Bertsekas, *Reinforcement learning and optimal control.* Athena Scientific, 2019.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[3] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.