Dynamic Programming Notes

Steven M. LaValle

1 Defining the multiple-stage decision making scenario

1. Let the *state space*, X, be a nonempty, finite set of *states*.
2. Let $U(x)$ denote a finite set of *actions* that can be applied from a state $x \in X$. Assume $U(x)$ is specified for all $x \in X$.
3. Let $k \in \{1, \ldots, K+1\}$ denote the *stage*. It is assumed that K is larger than the number of stages required for the longest optimal path between any pair of states in X.
4. Let F denote the index of the *final stage*. If K is finite, then $F = K+1$.
5. Let f denote a *state transition equation*, which yields $x_{k+1} = f(x_k, u_k)$ for any $x_k \in X$ and $u_k \in U(x_k)$.
6. Let x_I denote the *initial state*, and let x_G denote the *goal state*. Alternatively, we may consider a set of goal states, X_G, or there may be no explicit goal.
7. Let L denote an additive loss functional, which is applied to the sequence, $u_1, \ldots, u_K$, of applied actions, and achieved states, $x_1, \ldots, x_F$:
$$L = \sum_{k=1}^{K} l(x_k, u_k) + l_F(x_F). \tag{1}$$
The final term, $l_F(x_F)$ is usually defined as $l_F(x_F) = 0$ if $x_F = x_G$, and $l_F(x_F) = \infty$ otherwise.
8. Each $U(x)$ contains a special *termination action*, u_T. If u_T is applied to x_k, then the action is repeatedly applied until stage K, and the state remains in x_k until the final stage. Furthermore, no loss accumulates: $l(x_k, u_T) = 0$ for any k and x_k.
9. The task is to find a sequence of actions, $u_1, \ldots, u_K$ that minimizes L.

2 Backwards dynamic programming

Since it involves less notation, we first consider the case of backwards dynamic programming. In this case, dynamic programming becomes a recursive formulation of the optimal *cost-to-go* to the goal.

Suppose that the final-stage loss is defined as $l_F(x_F) = 0$ if $x_F = x_G$, and $l_F(x_F) = \infty$ otherwise. The final-stage optimal cost-to-go is:
$$L^*_{F,F}(x_F) = l_F(x_F). \tag{2}$$
This makes intuitive sense: since there are no stages in which an action can be applied, we immediately receive the penalty for not being in the goal.

For an intermediate stage, $k \in \{2, \ldots, K\}$ the following represents the optimal cost-to-go:
$$L^*_{k,F}(x_k) = \min_{u_k, \ldots, u_K} \left\{ \sum_{i=k}^{K} l(x_i, u_i) + l_F(x_F) \right\} \tag{3}$$

In the minimization above, it is assumed that $u_i \in U(x_i)$ for every $i \in \{k, \ldots, K\}$. Equation (3) expresses the lowest possible loss that can be received by starting from x_k and applying actions over $K - k + 1$ stages. The termination action can be applied at any stage, but a loss, $l_F(x_F)$ must be paid at the final stage. Thus, if you are not already in the goal, you are forced to move; otherwise, infinite loss will be obtained.

The final step in backwards dynamic programming is the arrival at the initial stage. The cost-to-go in this case is

$$L^*_{1,F}(x_1) = \min_{u_1,\ldots,u_K} \left\{ l_I(x_1) + \sum_{i=1}^{K} l(x_i, u_i) + l_F(x_F) \right\}, \tag{4}$$

in which $l_I(x_1)$ denotes an initial loss. For most applications of backwards dynamic programming, we can safely make $l_I(x_1) = 0$ for all $x_1 \in X$. In forward dynamic programming, l_I becomes more important, and l_F becomes unimportant.

The dynamic programming equation expresses $L^*_{k,F}$ in terms of $L^*_{k+1,F}$:

$$L^*_{k,F}(x_k) = \min_{u_k \in U(x_k)} \left\{ L^*_{k+1,F}(x_{k+1}) + l(x_k, u_k) \right\}. \tag{5}$$

Note that x_{k+1} is given by the state transition equation: $x_{k+1} = f(x_k, u_k)$.

3 Forward dynamic programming

Dynamic programming in this case involves the recursive formulation of the *cost-to-come* from an initial state. Notice that the concepts here are the symmetric equivalent of backwards dynamic programming concepts.

Suppose that the initial loss is defined as $l_I(x_1) = 0$ if $x_1 = x_I$, and $l_I(x_1) = \infty$ otherwise. This will ensure that only paths that start at x_I are considered (in the backwards case, l_F ensures that only paths which end at x_G are considered).

The initial optimal cost-to-come is:

$$L^*_{1,1}(x_1) = l_I(x_1). \tag{6}$$

For an intermediate stage, $k \in \{2, \ldots, K\}$ the following represents the optimal cost-to-come:

$$L^*_{1,k}(x_k) = \min_{u_1,\ldots,u_{k-1}} \left\{ l_I(x_1) + \sum_{i=1}^{k-1} l(x_i, u_i) \right\}. \tag{7}$$

Note that the sum refers to a sequence of states, $x_1, \ldots, x_k$, which are the result of applying the action sequence $u_1, \ldots, u_{k-1}$. As in (3) it is assumed that $u_i \in U(x_i)$ for every $i \in \{1, \ldots, k-1\}$. The resulting x_k, obtained after applying u_{k-1} must be the same x_k that is named in the argument on the right side of (7). It might appear odd that x_1 appears inside the min above; however, this is not a problem. The state x_1 can be completely determined once $u_1, \ldots, u_{k-1}$ and x_k are given.

The final step in forward dynamic programming is the arrival at the final stage. The cost-to-come in this case is

$$L^*_{1,F}(x_F) = \min_{u_1,\ldots,u_K} \left\{ l_I(x_1) + \sum_{i=1}^{K} l(x_i, u_i) + l_F(x_F) \right\}. \tag{8}$$

This equation looks the same as (4), but x_1 is replaced with x_F on the right. This means that all paths ending at x_F are considered here, whereas all paths starting at x_1 were considered in the backwards case.

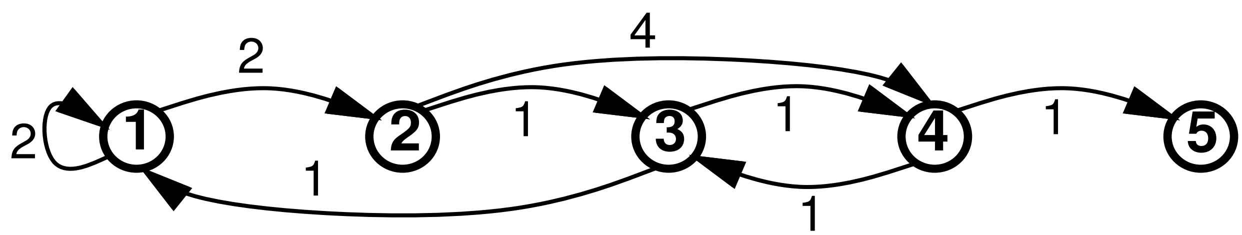

Figure 1: A five-state example is shown. Each vertex represents a state, and each edge represents an input that can be applied to the state transition equation to change the state. The weights on the edges represent $l(x_k, u_k)$ (x_k is the originating vertex of the edge).

To express the dynamic programming recurrence, one further issue remains. Suppose that $L^*_{F,k-1}$ is known by induction, and we want to compute $L^*_{1,k}(x_k)$ for a particular x_k. This means that we must start at some state x_{k-1} and arrive in state x_k by applying some action. Which actions are possible? In terms of the graph representation, this means that we need to consider all in-edges for the node x_k. For the case of backward dynamic programming, we used the out-edges, which are simply given by applying each $u_k \in U(x_k)$ in $f(x_k, u_k)$ to obtain each x_{k+1}. This effectively tries all out-edges of x_k.

To try all in-edges, we can formulate a *backwards state transition equation*, which can always be derived from the (forward) state transition equation. Let $x_{k-1} = \phi(x_k, \omega_k)$ denote the backwards state transition equation. The ω_k can be considered as a kind of *backwards action* that when applied to x_k, moves us backwards to state x_k. Let $\Omega(x_k)$ denote the set of all backwards actions available from state x_k. Each element, ω_k, of $\Omega(x_k)$ corresponds exactly to an in-edge to x_k in the graph representation. Furthermore, ω_k corresponds to an input, u_{k-1} that got applied to x_{k-1} to reach state x_k. Thus, we have simply derived a "backwards" version of the state transition equation, $x_{k+1} = f(x_k, u_k)$.

Using the backwards state transition equation, the dynamic programming equation is:

$$L^*_{1,k}(x_k) = \min_{\omega_k \in \Omega(x_k)} \left\{ L^*_{1,k-1}(x_{k-1}) + l(x_{k-1}, u_{k-1}) \right\}, \tag{9}$$

in which $x_{k-1} = \phi(x_k, \omega_k)$ and u_{k-1} is the input to which ω_k corresponds.

4 A simple example

Consider the example shown in Figure 1, in which $X = \{1, 2, 3, 4, 5\}$. Suppose that $x_I = 2$ and $x_G = 5$.

Using forward dynamic programming for $K = 4$, the following cost-to-come functions are obtained:

State	1	2	3	4	5
$L^*_{1,1}$	∞	0	∞	∞	∞
$L^*_{1,2}$	∞	0	1	4	∞
$L^*_{1,3}$	2	0	1	2	5
$L^*_{1,4}$	2	0	1	2	3
$L^*_{1,5}$	∞	∞	∞	∞	3

For any finite value that remains content from one iteration to the next, the termination action was applied. Note that the last step of the dynamic programming is useless in this example. Once $L^*_{1,4}$ is computed, the optimal cost-to-come to every possible state from x_I is determined. The addition of $l_F(x_F)$ in the last stage only preserves the cost-to-come for the goal state.

As an example of the backwards state transition equation, consider the construction of $L^*_{1,3}(4)$. There are two in-edges for state 4; thus, the size of Ω_3 is 2. One backwards action yields $x_2 = 3$, and the other yields $x_2 = 2$. The losses, $l(x_2, u_2)$, for each of these are 1 and 4, respectively.

Next consider applying backwards dynamic programming for $K = 4$:

State	1	2	3	4	5
$L^*_{5,5}$	∞	∞	∞	∞	0
$L^*_{4,5}$	∞	∞	∞	1	0
$L^*_{3,5}$	∞	5	2	1	0
$L^*_{2,5}$	7	3	2	1	0
$L^*_{1,5}$	∞	3	∞	∞	∞

Once again, the final step serves no useful purpose in this example. Note that $L^*_{2,5}$ contains the optimal cost to go to x_G from any initial state (by symmetry, forward dynamic programming yielded the optimal cost to come from x_I to any state), under the assumption that 3 steps are taken. If we would like to know the optimal cost to go from state 1, then more iterations are required.

It turns out that we do not need to precompute the value of K in practice. Consider the backwards dynamic programming iterations below:

State	1	2	3	4	5
$L^*_{9,9}$	∞	∞	∞	∞	0
$L^*_{8,9}$	∞	∞	∞	1	0
$L^*_{7,9}$	∞	5	2	1	0
$L^*_{6,9}$	7	3	2	1	0
$L^*_{5,9}$	5	3	2	1	0
$L^*_{4,9}$	5	3	2	1	0
$L^*_{3,9}$	5	3	2	1	0
$L^*_{2,9}$	5	3	2	1	0
$L^*_{1,9}$	∞	3	∞	∞	∞

After some number of iterations, the cost-to-go values stabilize. After this point, the termination action is being applied from all reachable states and no further loss accumulates. Once the values remain constant, the iterations can be stopped. If some vertices are not reachable, their values will also remain constant, at ∞.

How is an optimal action sequence returned? Assume that backwards dynamic programming is used. One method that always works is to store the action, u_k, that minimized (5) at vertex x_k. Alternatively, if the final step is skipped, then the optimal action sequence can be obtained using any starting state by repreatedly selecting u_k to minimize

$$L^*(x_{k+1}) + l(x_k, u_k), \tag{10}$$

in which L^* denotes the stabilized cost-to-go that is computed by the backwards dynamic programming computations (skipping the final step, which converts most values to ∞).