CS497 Week 3 Notes (2/4, 2/6)

Scribe: Rishi Talreja

1 Multiple Stage Decision Making

We now move on to decision making in multiple stages. First we formally state the problem and basic definitions. The we discuss three basic representations of the problem and briefly discuss their pros and cons. After this, we show how to apply dynamic programming to make decisions in multiple stages.

One difference between the problem we are currently dealing with and what we have studied so far is that we do not worry about nature. Many of the other characteristics are similar. We define the following variables.

1. Let X be a state space.

   Example 1 *Some possibilities for X are the following:*

   (a) $X = \{1, 2, \ldots, K\}$.
   (b) $X = \mathbb{R}^d$.
   (c) $X = \mathbb{Z}^d$ *(the grid of integer points)*
   (d) *X can be the set of all boolean assignments to a vector of literals. In this case it X will be the set of all boolean strings. We will see this when we discuss the STRIPS representation.*

2. Let $x \in X$ be a specific state. Our actions will effect what state we are in.
3. Let $k = \{1, 2, \ldots, K\}$ be the stages we are making decisions over.
4. Let F denote the index of the final stage. If K is finite, then $F = K + 1$.
5. Let x_k be the state we are in at stage k, before a decision has been made.
6. Let x_I be the initial state and x_G be the goal state. Alternatively, we can have a set of possible goal states, X_G, or there may be no explicit goal.
7. Let U be a set of all possible actions.
8. Let $U(x) \subseteq U$ denote the set of all choices that can be made from state x.

Given a state x_k and an action $u_k \in U(x_k)$, we make a transition into a new state x_{k+1}. In other words, there is a partial function $f : X \times U \to X$ such that

$$x_{k+1} = f(x_k, u_k).$$

We call this equation the state transition equation. The reason f is a partial function is because not all choices can be applied to a given state. There are some state/action pairs for which f is undefined.

1.1 Representations

1. **STRIPS**

   The STRIPS representation is the classical approach to planning. It describes states in terms of conjunctions of possibly negated predicates applied to symbols. STRIPS operators consist of three components:

   (a) The **action** serves as a name for a possible action that can be taken.

   (b) The **precondition** is a conjunction of positive literals that says what must be true before the operator can be applied.

   (c) The **effect** of an operator is a conjunction of positive or negative literals that describes how the situation changes when the operator is applied.

   Consider the following example:

   Example 2 *Suppose that countries want to decide whether or not they want to attack the United States. We can define the state of the world to be a bit vector with the following entries:*

   $x = [HasWMD(Iraq), HasWMD(NKorea), ..., Friendly(Canand), \neg Friendly(Iraq)].$

   Then, operators such as the following could be applied to a state:

   $$\begin{aligned}&Op(ACTION: Attack(i),\\ &\quad PRECONDITION: HasWMD(i) \wedge \neg Friendly(i),\\ &\quad EFFECT: \neg HasWMD(i) \wedge HatesUS(i))\end{aligned}$$

   Applying an operator to a state changes the bits in the state, and puts the world in a different state.

   Some advantages of using the STRIPS representation are it is easy to directly express domain knowledge as well as output. On the other hand, representations can be very lengthy and are usually difficult to generalize.

2. **Decision Theoretic**

   Now we consider a decision theoretic approach. First we must define the cost, or loss, functional. Suppose there are K stages over which we make decisions. Let $L : U^K \times X^{K+1} \to \mathbb{R}$ be a partial function. In other words,

   $$L(u_1, u_2, \ldots, u_k, x_1, x_2, \ldots, x_K, x_{K+1}) \in \mathbb{R},$$

   where $u_i \in U(x_i)$ for all $i \in \{0, \ldots, K\}$. We think of this function as the cost of evaluating a plan. In can also be written in terms of costs of being in certain stages. Let $l(x_k, u_k)$ be the cost of being in stage x_k and applying u_k, for all $k \in \{1, \ldots, K\}$. Also, let $l_{K+1}(x_{K+1})$ be the cost of being in the final stage. Then,

   $$L = \sum_{i=1}^{k} l(x_i, u_i) + l_{K+1}(x_{K+1}).$$

   The reason L is a partial function is the same reason f, which was defined above, was a partial function. Some possibilities for the function l are shown below:

Example 3 *In classical planning, the following stage additive cost functions have often been used:*

$$l(x,u) = 0,$$
$$l_F(x_F) = \begin{cases} 0 & \text{if } x_F \in X_G, \\ 1 & \text{otherwise} \end{cases}$$

Alternatively, the following functions have often been used in optimal-step planning:

$$l(x,u) = 1,$$
$$l_F(x_F) = \begin{cases} 0 & \text{if } x_F \in X_G, \\ \infty & \text{otherwise} \end{cases}$$

We will see this formulation again soon.

In order to extend our model to a model with infinite stages we must define a termination action. Let a termination action u_t be an action with the following properties:

(a) Once u_t is applied, it can never be applied again.

(b) After u_t is applied, no more losses can accumulate from l.

(c) After u_t is applied, state can not change.

Now, if we have an infinite number of stages, we can define L to be

$$L = \sum_{i=1}^{\infty} l(x_i, u_i) + l_F(x_F),$$

where F is the stage at which u_t is applied. Our task is to find a sequence of actions, $u_1, \ldots, u_k$ that minimize L.

One benefit of using the decision theoretic framework for multiple stage decision making is that it is the most straightforward to generalize out of the three we discuss here. However, because of this feature of the model we may lose some information about our original domain by generalizing.

3. **Graph Theoretic**

We now describe how to use a graph to represent a multiple stage decision making problem. Let $G = (V, E)$ be a directed, weighted graph. Define the vertex set to be the set of all possible states, $V = X$, and the edge set E to be the set of all operators that can be applied from every vertex. Let $E(v)$ denote the set of all ougoing edges form a vertex v. Then, if a vertex v represents a state x, then $E(v)$ represents $U(x)$, all possible choices that can be made at stage x. Now we can use standard shortest path algorithms such as Dijkstra's algorithm in order to make an optimal decision.

A variant of the graph representation is the tree representation. The initial state x_I is defined to be the root of the tree. Then, for each state x_k, there are branches down to every state, x_{k+1} that can be reached in a single stage. If we define all nodes in the tree that have the same state to be identical, we are back to our original graph representation.

Graph representations are natural to deal when algorithmic or complexity issues are at hand. Also, the representations can usually be generalized and can provide a nice visualization for the problem at hand.

1.2 Dynamic Programming

Given the definition of the multiple stage decision making problem above along with the definitions of the loss functional given in Section 1.1, we present a dynamic programming solution to our problem. Dynamic programming is a method that can be used for solving optimal path planning problems that was invented in 1957 by Bellman. Other methods such as breadth-first search, depth-first search, and A^* have been used to solve classical planning problems. First we present backwards dynamic programming, and then forward dynamic programming. Backwards dynamic programming is usually chosen over the forward version because less notation is needed.

1. **Backwards dynamic programming**

   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. In backwards dynamic programming, we assume that the goal state, x_G, is fixed. This is appropriate measure because it forces us to always at least find a path to the goal state.

   Now we define an optimal cost-to-go function, $L^*_{k,F} : X \to \mathbb{R}$, to be used with our dynamic programming recurrence. The final-stage optimal cost-to-go is:

   $$L^*_{F,F}(x_F) = l_F(x_F).$$

   This ensures that only paths which end up in the goal state, x_G, are considrered.

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

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

   In the minimization above, it is assumed that $u_i \in U(x_i)$ for every $i \in \{k, \ldots, K\}$. Equation (1) 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 or else an 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

   $$L^*_{1,F}(x_1) = \min_{u_1,\ldots,u_K} \left\{ l_I(x_1) + \sum_{i=1}^{K} l(x_i, u_i) + l_F(x_F) \right\}, \tag{2}$$

   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 $L^*_{k+1,F}$:

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

Note that x_{k+1} is given by the state transition equation, $x_{k+1} = f(x_k, u_k)$, so all entries in a table used for this dynamic programming solution can be filled.

2. **Forward dynamic programming**

Dynamic programming in this case involves the recursive formulation of the *cost-to-come* function from an initial state, x_I, which is fixed. Notice that the concepts here are the symmetric equivalent of backwards dynamic programming concepts, with a slight technicality.

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.

The initial optimal cost-to-come is:

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

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

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

The sum refers to a sequence of states, $x_1, \dots, x_k$, which are the result of applying the action sequence $u_1, \dots, u_{k-1}$. It is again assumed that $u_i \in U(x_i)$ for every $i \in \{1, \dots, k-1\}$. Notice that the state x_1 can be completely determined once $u_1, \dots, u_{k-1}$ 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

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

This equation looks the same as (2), 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.

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. We need to know which actions are possible. In backwards dynamic programming, we knew what actions we could take simply by applying each $u_k \in U(x_k)$ in $f(x_k, u_k)$ to obtain each possible x_{k+1}.

To try all possible actions in the case of forward dynamic programming, 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-1}. 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 a choice that can take us to x_k. Thus, we now have a backwards version of the state transition equation, $x_{k+1} = f(x_k, u_k)$. We can use ϕ to get all possible actions that take us to a given state x_k.

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

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

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