Notes for Feb 4 and Feb 6, 2003.
Scribed by Pedro DeRose

1 Multiple Stage (Sequential) Decision Making

Notation

K	number of the final stage
k	individual stages, or points where decisions are made. Range $\{1, \ldots, K\}$
x	a state
X	state space (e.g. $\{1, 2, \ldots, K\}$ or $\mathbf{R}^d$)
x_k	state at stage k. Therefore, x_1 is the initial state, and x_K is the final state
x_G	a goal state
F	stage $K+1$. Useful when considering x_F, i.e. the state where you run out of moves
I	the first stage. Useful when considering x_I, i.e. the initial state
u	actions, or choices, or inputs
U	action space
u_k	the action, during stage k, taking you from state x_k to state x_{k+1}
$U(x)$	actions allowed from state x
$l(x, u)$	the loss, or cost, of being in state x and choosing action u to leave
$l_F(x_F)$	the cost of reaching the final state
$l_I(x_I)$	the cost already accrued at the initial state

1.1 Loss (Cost) Functionals

Given the above notation, an **Operator** is simply a **state transition equation** of the form

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

with $f : X \times U \to X$, given that U is restricted to $U(x_k)$.

Let us define a loss, or cost, function $L(u_1, \ldots, u_K, x_1, \ldots, x_K, x_F)$ which returns the cost of evaluating a plan $\{u_1, \ldots, u_K\}$ from initial state x_1. One convenient representation of this function uses **stage additive loss**. Intuitively, this is calculated by adding the cost at the initial state, summing $l(x_k, u_k)$ for every k, then adding $l_F(x_F)$. Formally,

$$L = l_I(x_I) + \sum_{k=1}^{K} l(x_k, u_k) + l_F(x_F)$$

There remains the question of what to do when K is unknown or unspecified. One solution is to assume K is so big as to be essentially infinite and add a **termination action** u_T which

- Does not change the state
- Once applies, is never evoked again
- Prevents any more non-final losses from accumulating

Once the termination action is taken, we set F to the stage at which it was evoked and calculate the loss function normally. Another solution is to define a **termination state**, x_T, with no actions leaving it and no cost to remain in it. In either case, in order to avoid cheating by terminating before some large necessary cost (e.g. a driving program terminating while still at high speeds to avoid the cost of crashing), one would have to restrict from which states the agent is allowed to terminate.

1.2 Three Different World Representations

The problem described above is a generalized multi-stage planning problem. However, the representation we use is by no means set in stone. Any number of representations exist, each with its own advantages and disadvantages.

To make this point, we will look at a toy problem and demonstrate how it would be represented under three different world representations. Consider the stereotypical planning problem, BlockWorld, where an agent controlling a robotic arm is to place three blocks, A, B, and C, one on top of the other to form a tower with A at its base and C at its peak, and to do so in the fewest number of moves. To simplify the problem, we'll imagine that the robot arm is capable of responding to such requests as "move over B", "pick up the block under you", and "drop the block you're holding."

1.2.1 Decision (Control) Theoretic

This representation is the one we have used above to explain the problem in general terms. In fact, this is precisely the advantage it has over other representations—it is extremely easy to generalize. The disadvantage, however, is that its mathematical formalism will often obscure domain knowledge.

A decision theoretic agent would approach the BlockWorld problem by defining a real-world concept to each of its many mathematical concepts. X would comprise all the various states the blocks could be in relation to each other and the arm, namely which block is on top of which other block, which block the arm is over, and which block the arm is holding. U would comprise the actions to move over a given block, pick up the block under the arm, and drop the currently held block. Since we wish to build the tower in the fewest number of moves, $l(x, u)$ would be 1 for all x and u. In order to prevent early termination, we would make the cost of terminating be infinite from any other state except that in which our tower is built properly.

1.2.2 STRIPS (Chapter 11 in Russel & Norvig)

The concept behind this representation is to provide an easily human-writable definition of the world. This is done by defining a state representation made up of predicates, then defining a set of operators, each with a precondition and an effect represented by predicate formulas. For added simplicity in writing, the predicates used in these formulas can be parametric. One then defines a cost for each operator (optionally also dependent on the state). The advantage to this representation is that it is easy to read and understand. However, it tends to be lengthy and difficult to generalize.

To represent our BlockWorld problem, we could define the state as a bit-string, with each position representing a different predicate. The predicates themselves would represent the states

we used in the decision theoretic representation. The cost for all actions would be 1, and the cost for not reaching the goal would be infinite. The individual actions would look something like this:

```
Op(Action: MoveOver(block),
   Precondition: !amOver(block),
   Effect: amOver(block))

Op(Action: PickUp(block)
   Precondition: amOver(block) & !amHolding(A) & !amHolding(B) & !amHolding(C),
   Effect: amHolding(block))
```

$\vdots$

1.2.3 Graph Representation

The third representation uses a directed, weighted graph $G(V, E)$ where the vertexes V represent X, the set of states, and the edges E represent the set of all operator-state tuples (STRIPS-operators are templates, while these are all the instantiations of all templates). The weights along the edges represent costs of actions. The result is that this formulation allows us to consider a plan as a Markov chain. The advantage of this representation is that it allows for us to use dynamic programming approaches and graph searches to form plans. The disadvantage is that all operator instantiations must be represented.

To build the BlockWorld graph representation, we start with the STRIPS representation. For each state, we create a vertex. Then, for every operator whose precondition is fulfilled by a state, we draw an edge from that state to the resulting state and give it a weight equal to its cost.

1.3 Planning as Search

The task at hand is to find a plan, represented by $\{u_1, \ldots, u_K\}$, that minimizes our loss function L. One way to do this is to use a graph representation and search the graph for the shortest path to the goal state x_G. We'll begin by assuming the graph contains no negative cycles. In other words, there is no series of actions the agent can repeat to lessen their loss. Given this assumption, one method of planning is to use Dijkstra's algorithm. This takes one initial state and calculates the optimal **cost-to-come** for every other state, i.e. the lowest cost to come from the initial state to the given state.

An important observation is that Dijkstra's is just an implementation of a broader family of algorithms, namely **forward dynamic programming**. All these algorithms compute the cost-to-come from a single starting state to any state. This, in turn, is just a mirror image of **backward dynamic programming**, which computes the **cost-to-go** from any state to a single goal state. Both define a loss function $L^* : X \rightarrow \mathbf{R}$ that computes the loss for the optimal path to/from a given state. The subscripts on the function are notation devices to tell us the range of states being considered.

Forward Dynamic Programming	Backward Dynamic Programming
cost-to-come (fixed x_I, any goals)	cost-to-go (fixed x_G, any source)
$L^*_{1,1}(x_1) = l_I(x_I)$	$L^*_{F,F} = l_F(x_F)$
$L^*_{1,F}(x_F) = \min_{u_1 \dots u_K} l_I(x_I) + \sum_{i=k}^{K} l(x_i, u_i) + l_F(x_F)$	$L^*_{1,F}(x_F) = \min_{u_1 \dots u_K} l_I(x_I) + \sum_{i=k}^{K} l(x_i, u_i) + l_F(x_F)$
$1<k<F, L^*_{1,k}(x_k) = \min_{u_1 \dots u_{k-1}} l_I(x_I) + \sum_{i=1}^{k-1} l(x_i, u_i)$	$1<k<F, L^*_{k,F}(x_k) = \min_{u_1 \dots u_K} \sum_{i=k}^{K} l(x_i, u_i) + l_F(x_F)$

Given the above loss functions with their base cases, it's trivial to build a dynamic programming algorithm to compute $L^*(x)$. It's simply a matter of expressing the current step in terms of the last step. Notice that we have a function, namely the state transition function, that will return x_{k+1} given x_k and u_k. It's existence makes it more convenient to consider backward dynamic programming rather than forward because in backward, if our current state is x_k, our last state was x_{k+1}. Therefore, the dynamic programming equation

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

can be transformed through the state transition function f into

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

As we iterate over this equation, starting with $k = K$ and reducing it by one on each iteration, we are in essence assigning to each state the optimal cost of a path of length $F - k$ from it to the goal state. Once we reach $k = 1$, we will have assigned the cost-to-go value for each state, thereby allowing us to determine the optimal path. The forward dynamic programming algorithm can be created just as easily, provided we have a function that gives us x_k from u_k and x_{k+1}. Lastly, in situations where K is unknown but there exists a termination action or state, we can start at $k = 0$ and count down until the costs assigned to all the states stop changing, at which point we end the algorithm and determine the optimal path.