Financial Optimization
ISE 347/447

Lecture 16

Dr. Ted Ralphs

Reading for This Lecture

- C&T Chapter 13

Dynamic Programming

- *Dynamic programming* is a methodology applied primarily to sequential decision processes, such as those occurring in stages over time.
- Because DP methods are well-suited for problems with a time dimension, they arise naturally in financial settings.
- Note that the term dynamic programming refers both to a modeling paradigm and to a specific set of methodologies.
- Dynamic programming methods are based on Bellman's *Principle of Optimality*.
- This states roughly that in a sequential decision process, every subsequence of an optimal decision sequence must also be optimal when viewed as a separate decision problem.
- This principle enables us to formulate recursive relationships that lead to algorithms for solving optimization problems.

Elements of Dynamic Programming

- The common elements of a DP include
  - A set of decision *stages*.
  - A set of possible *states* in each decision stage.
  - A set of *transitions* between states.
  - A *value function* for each state in each stage indicating the best objective value achievable from that state.
- Each transition has a cost and can be associated with either an action by the decision-maker or a random event.
- There are implicit constraints that determine what transitions are feasible.
- Initially, we will consider only *deterministic DPs*, in which the transitions are a result of actions.

An Example

- Recall the capital budgeting example from Lecture 13
- We have $4 million to invest in projects over the next three years.
- Each project has an associated cost and profit (in present value dollars) as follows:

	Year 1		Year 2		Year 3	
Project	Cost	Profit	Cost	Profit	Cost	Profit
1	0	0	0	0	0	0
2	1	2	1	3	1	2
3	2	4	3	9	2	5
4	4	10	-	-	-	-

- Here, we have a sequential decision process that is amenable to a dynamic programming approach.

The Principle of Optimality

- In this problem, the stages are the time periods and the states are represented simply as the amount of capital left to invest.
- We add a stage and a state $(0, 4)$ to represent the initial problem.
- Let's assume that we have already decided to invest in project 2 during the first period.
- This means that we have now reduced the problem to one with three stages in which we have a budget of \$3 million.
- The optimal sequence starting from state $(1, 3)$ can be determined independent of the stage 1 decision.
- The principle of optimality tells us that any transition made from state $(1, 3)$ in an overall optimal decision sequence must be consistent such a sequence starting initially from state $(1, 3)$.

Backward Recursion

- A DP can be initiated using either a *backward* or *forward recursion*.
- In the backward case, we compute the optimal decision starting from each state recursively, beginning at the last period.
- The value function for a state represents the cost of an optimal decision sequence beginning from the given state.
- In the last stage, there are no decisions left to be made, so the value function for all states is set to zero.
- We then consider transitions from the second to the third period.
- What are the states associated with the second period?

Solution Process

- Let us consider state $(2, 4)$.
- The only feasible transitions from state $(2, 4)$ are to states $(3, 2)$, $(3, 3)$, and $(3, 4)$.
- The associated costs are 5 , 2, and 0.
- Taking the maximum of these costs, we can determine that the value function at state $(2, 4)$ is 5.
- We can perform the same analysis for each of the remaining states associated with period 2.
- With full knowledge of the value functions in periods 2 and 3, the same basic analysis can now be applied to period 1.
- Finally, we move back to the initial state.

Forward Recursion

- In the case of forward recursion, the value function for state (i, j) represents the maximum profit that can be obtained in transitioning from the initial state to state (i, j).
- This time, we initialize by setting the value function of the initial state to zero.
- There is only one way to reach each of the states in period 1, so we simply set the value function to the cost of that transition.
- In period 2, let us consider state $(2, 3)$.
- There are two ways to transition to this state, from either $(1, 4)$ or $(1, 3)$.
- Since we know the values all states in period 1 already, the value at state $(2, 3)$ can easily be computed as 3.

Formalizing

- We first consider deterministic, discrete dynamic programs in which the set of states in each stage and the set of possible transitions from each state are finite.
- We consider a set of stages indexed by $1, \ldots, T$.
- The states in stage t are indexed $1, \ldots, K_t$ and are denoted by an ordered pair (t, k) consisting of the stage and the particular state in that stage.
- The set of feasible decisions in state (i, j) is denoted by $\mathcal{S}(i, j)$.
- Each decision results in a transition to a unique state denoted by $T((i, j), d)$.
- Note that the transition state is often in the next stage, but it does not have to be.
- The cost of the transition is denoted by $c((i, j), d)$.

The DP Recursion

- The value function $v(i, j)$ at state (i, j) denotes either
  - The optimal cost/profit accumulated from the initial state to state (i, j) (forward method), or
  - The optimal cost/profit accumulated from state (i, j) to some state in the final stage (backward method).
- The principle of optimality gives us a recurrence for determining the value function in a given state.
- Let us consider the backward method for a minimization problem.
- In this case, we can write

$$v(i, j) = \min_{d \in \mathcal{S}(i,j)} \{v(T((i, j), d)) + c((i, j), d)\}.$$

- For the forward method, there is a similar recurrence.

The Knapsack Problem Revisited

- Recall the integer knapsack problem from Lecture 13.
  - We are given a set of items with associated values and weights.
  - We wish to select a subset of maximum value such that the total weight is less than a constant K.
  - We associate a 0-1 variable with each item indicating whether it is selected or not.

$$
\begin{aligned}
max \quad & \sum_{j=1}^{m} p_j x_j \\
s.t. \quad & \sum_{j=1}^{m} w_j x_j \leq K \\
& x \geq 0 \\
& x \quad integer
\end{aligned}
$$

- Knapsack problems arise as subproblems in many financial applications.

Formulating The Knapsack Problem as a DP

- Often, the most difficult part of using dynamic programming is formulating the problem as a DP.
- There are usually multiple ways of doing this.
- In our first formulation, there will be K stages representing the remaining capacity of the knapsack.
- Each stage has just one state in this formulation, so we use just one index to represent both.
- The transitions involve putting one of the items into the knapsack.
- Then we have that $\mathcal{S}(i) = \{d \mid w_d \leq i\}$, $T(i,d) = i - w_d$, and $c(i,d) = p_d$.
- The (backward) recurrence is then

$$v(i) = \max_{d \in \mathcal{S}(i)} \{v(i - w_d) + p_d\}.$$

Another Formulation

- Another approach is to associate stage i with item (variable) i.
- The state is the capacity remaining after adding items $1, \ldots, i-1$.
- The decision associated with stage i is then the number of items of type i to be included in the knapsack.
- We then have

$$\mathcal{S}(i,j) = \{d \in \mathbb{Z}_+ \mid d \leq j/w_i\}$$

- The transition function is

$$T((i,j),d) = (i+1, j - dw_i)$$

- Finally, the (backward) recurrence is

$$v(i,j) = \max_{d \in \mathcal{S}(i,j)} \{v(i+1, j-dw_i) + dp_i\}$$

Simple Knapsack Solver in Python

```
def knapsack01(obj, weights, capacity):
    """ 0/1 knapsack solver, maximizes profit. weights and capacity integer """
    n = len(obj)
    c = [[0]*(capacity+1) for i in range(n)]
    added = [[False]*(capacity+1) for i in range(n)]
    # c [items, remaining capacity]
    # important: this code assumes strictly positive objective values
    for i in range(n):
        for j in range(capacity+1):
            if (weights[i] > j):
                c[i][j] = c[i-1][j]
            else:
                c_add = obj[i] + c[i-1][j-weights[i]]
                if c_add > c[i-1][j]:
                    c[i][j] = c_add
                    added[i][j] = True
                else:
                    c[i][j] = c[i-1][j]
```

Simple Knapsack Solver in Python (cont'd)

```
# backtrack to find solution
i = n-1
j = capacity

solution = []
while i >= 0 and j >= 0:
    if added[i][j]:
        solution.append(i)
        j -= weights[i]
    i -= 1

return c[n-1][capacity], solution
```

Stochastic Dynamic Programming

- The framework discussed here can be enhanced to include stochasticity.
- In other words, the transition occurring after a decision is to one of several states, each with a certain given probability.
- For each state (i,j) and decision $d \in \mathcal{S}(i,j)$, we have a set of transition states denoted by $\mathbb{R}((i,j),d)$.
- For each $r \in \mathbb{R}((i,j),d)$, we have a probability $p((i,j),d,r)$ of transition to state $T((i,j),d,r)$ with cost $c((i,j),d,r)$.
- The objective function is stated in terms of expected values, so that the (backward) recurrence becomes

$$v(i,j) = \min_{d \in \mathcal{S}(i,j)} \left\{ \sum_{r \in \mathbb{R}((i,j),d)} p((i,j),d,r)[v(T((i,j),d,r)) + c((i,j),d,r)] \right\}$$