CS 161 W19: Recitation 7 Solutions

February 2019

Exercise 0

Suppose the various economies of the world use a set of currencies $C_1, \ldots, C_n$ — think of these as dollars, pounds, bitcoins, etc. Your bank allows you to trade each currency C_i for any other currency C_j at an exchange rate r_{ij}, that is, you can exchange each unit of C_i for $r_{ij} > 0$ units of C_j. Due to fluctuations in the markets, it is occasionally possible to find a sequence of exchanges that lets you start with currency A, change into currencies, B, C, D, etc., and then end up changing back to currency A in such a way that you end up with more money than you started with. That is, there are currencies $C_{i_1}, \ldots, C_{i_k}$ such that

$$r_{i_1 i_2} \times r_{i_2 i_3} \times \cdots \times r_{i_{k-1} i_k} \times r_{i_k i_1} > 1.$$

This is called an arbitrage opportunity, but to take advantage of it you need to be able to identify it quickly (before other investors leverage it and the exchange rates balance out again)! Devise an efficient algorithm to determine whether an arbitrage opportunity exists. Justify the correctness of your algorithm and its runtime.

Solution 0

Build the complete directed graph with the currencies C_i as the vertices, and assign the edge (C_i, C_j) weight $-\log r_{ij}$. We run Bellman-Ford on this graph, and conclude there is an arbitrage opportunity if and only if Bellman-Ford detects a negative cycle (the distance estimates change on the n^{th} iteration).

Correctness: An arbitrage opportunity is equivalent to a negative cycle in this graph:

$$\begin{aligned} r_{i_1 i_2} \times r_{i_2 i_3} \times \cdots \times r_{i_{k-1} i_k} \times r_{i_k i_1} &> 1 \\ \log\left(r_{i_1 i_2} \times r_{i_2 i_3} \times \cdots \times r_{i_{k-1} i_k} \times r_{i_k i_1}\right) &> \log 1 \\ \log r_{i_1 i_2} + \log r_{i_2 i_3} + \cdots + \log r_{i_{k-1} i_k} + \log r_{i_k i_1} &> 0 \\ -\log r_{i_1 i_2} - \log r_{i_2 i_3} - \cdots - \log r_{i_{k-1} i_k} - \log r_{i_k i_1} &< 0. \end{aligned}$$

Thus Bellman-Ford identifies a negative cycle if and only if there exists an arbitrage opportunity.

Runtime: $O(n^3)$ total. Bellman-Ford runs in time $O(nm)$ where m is the number of edges, and the complete directed graph has $m = n(n-1)$ edges.

Exercise 1

Suppose we have a rod of length k, where k is a positive integer. We would like to cut the rod into integer-length segments such that we maximize the *product* of the resulting segments' lengths. Multiple cuts may be made. Write an algorithm to determine the maximum product possible.

Solution 1

Identify optimal substructure. To solve this problem we are going to exploit the following overlapping sub-problems. Let $f(\ell)$ be the maximum product possible for a rod of length ℓ, and let $f(0) = 1$.

We are going to try cutting the rod of length ℓ into two rods of length c and $\ell - c$ (recursing on the rod of length $\ell - c$) and try all possible values of c, taking the one which produces the maximum product. Note that not cutting the rod at all (cutting of a length of ℓ) is another option which we can take. Also notice that we do not need to consider cutting off a length of 1 since that will never yield the optimal product.

Find a recursive formulation. We have

$$f(\ell) = \max_{c \in \{2, \ell\}} c \cdot f(\ell - c).$$

We also let $f(0) = f(1) = 1$; the empty product canoncially evaluates to 1, the multiplicative identity.

Use dynamic programming. The following algorithm implements this recursive formulation using bottom-up dynamic programming.

Algorithm 1: MaxRodCut(k)

```
// f[ℓ] is the largest product for a rod of length ℓ
f = array of k+1 zero's
f[0] = f[1] = 1
for ℓ from 2 to k do
    bestProduct = 0
    for c from 2 to ℓ do
        product = c · f[ℓ − c]
        bestProduct = max(bestProduct, product)
    f[ℓ] = bestProduct
return f[k]
```

The running time for this algorithm is $O(k^2)$ since for each value of k we loop through $O(k)$ values to get the answer for that k.

Exercise 2

Given an 8×8 chessboard and a knight that starts at position $a1$, devise an algorithm that returns how many ways the knight can end up at position xy after k moves. Knights move ± 1 squares in one direction and ± 2 squares in the other direction.

Solution 2

Identify optimal substructure. We can store an array of the number of paths to each position after i moves, for each i. The base case is simple — after 0 moves, the knight has one way to end up in his starting position — not move at all!

If we know how many ways to get to each of the positions after round $i-1$, to get the number of ways he could move to some xy, we would add up the total number of ways he could have gotten to any of his last steps — 1 away in some direction and 2 away in the other. For example, there are 3 ways to get to $a2$ — either from $c1$, $c3$, or $b4$ - so we would simply add up the number of ways to get to each of these positions after $i-1$ steps to count how many ways to end up in position $a2$ after i steps.

Finally, once we compute the array for k, we can return position xy. (Notice that even though we only cared about position xy, we still computed the number of ways to get to any point on the chessboard in the previous steps — this is because these points could be on the path, despite not being the end point.)

Find a recursive formulation. More formally, we can define the function $f(i, x, y)$ to be the number of ways the knight can get to position xy in i moves. This gives us the following recurrence (written out for clarity):

$$\begin{aligned} f(i,x,y) = & f(i-1,x-1,y-2,i) + f(i-1,x-1,y+2) + f(i-1,x+1,y-2) + f(i-1,x+1,y+2) \\ & f(i-1,x-2,y-1) + f(i-1,x-2,y+1) + f(i-1,x+2,y-1) + f(i-1,x+2,y+1) \end{aligned}$$

where the function evaluates to 0 if the position is off the board. Also, as noted above, $f(0,0,0) = 1$, i.e., there is 1 way to get to square $a1$ in 0 moves, while for all other squares $f(x,y,0) = 0$.

Use dynamic programming. The following algorithm implements this recursive formulation using bottom-up dynamic programming.

Algorithm 2: KnightMoves(final_x, final_y, k)

```
// f[i][x][y] is the number of ways to reach square xy in i moves
f = array of k+1 matrices representing the 8x8 chessboard
initialize all of f to 0's, except f[0][0][0] = 1
moveDirections = [(1, 2), (1, -2), (-1, 2), (-1, -2), (2, 1), (2, -1), (-2, 1), (-2, -1)]
for i from 1 to k do
    for square xy in f[i] do
        for delta_x, delta_y in moveDirections do
            f[i] += f[i-1][x - delta_x][y - delta_y] if those indices are within the 8x8 board
return f[k][final_x][final_y]
```

This algorithm runs in time O(k) — with a large constant, because the inner loops sum over 8 positions for each of 64 squares.