CS 161 Winter 2020 Section 6

February 20-21, 2020

1 Edsger's Apfelstrudel

You are eating at a cozy little restaurant which serves a *prix fixe* menu of $k + 1$ courses, with several available choices for each course. Each dish belongs to exactly one course (e.g., risotto can only be ordered as an appetizer, not a main), and you are effectively indifferent between most of the items on the menu (because they are all so tasty), but the main draw of this particular restaurant is that they serve a delicious 'bottomless' dessert: their world-famous Viennese-style apple strudel. They have an unlimited supply of this apple strudel, but each serving will still cost you $1. The restaurant also has a few interesting rules:

1. You must finish your current dish before ordering another.
2. Each dish after the first course depends on what you ordered in the previous course, e.g., you can only order salmon for your main if you ordered a Caesar salad or chicken noodle soup for the previous course. You are told on the menu exactly what these restrictions are before you order anything.
3. Most importantly, you are not allowed to have their unlimited dessert unless you finish one dish from each of the first k courses!

You are told the cost of each item in each course on the menu, and you plan your meal with a twofold goal: to be able to order the strudel, but also to save as much money as possible throughout the first k courses so that you have more money to spend on the unlimited dessert. Design an algorithm to find the smallest amount of money you can spend on the first k courses and still order the 'bottomless' strudel. If you would like, you may assume the very first course has exactly one choice (e.g., a single complimentary leaf of spinach that costs 0 dollars).

Notice that this problem can be modeled as a shortest-path graph problem with costs on vertices rather than edges. However, we can transform this graph fairly easily into a directed weighted graph.

First we construct a directed graph in which each node is a dish and edges exist from v_i to v_j if you must finish dish i in order to order dish j.

Next, we augment the edges with weights as follows: for all j, for all (v_i, v_j), the weight of (v_i, v_j) is the caloric value of v_j.

Finally, we run Dijkstra's algorithm as we saw in class to find the shortest (least cost) path from any dish in the first course to the strudel!

An equivalent formulation of this algorithm is to skip the edge-weighting step and instead use a modified version of Dijkstra's algorithm in which path lengths are given by the sum of the costs of the vertices along that path.

Aside: note that without the dependencies caveat, a naive greedy algorithm solves this problem.

2 Rod Cutting

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. For example, if $k = 8$, the maximum product is 18 from cutting the rod into three pieces of length 3, 3, and 2. Write an algorithm to determine the maximum product for a rod of length k.

To solve this problem we are going to exploit the following overlapping sub-problems. If we let $f(k)$ be the maximum product possible for a rod of length k, then we have:

$$f(k) = \max_{c \in \{2, k-1\}} (k, f(c) \cdot f(k-c)).$$

Another way to think of this is that we are going to try cutting the rod of length k into two rods of length c and $k-c$ and try all possible values of c, taking the one which produces the maximum product. Note that not cutting the rod at all 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, and also do not need to try cuts any larger than $\lfloor k/2 \rfloor$ since those will already have been explored due to the symmetry of the cutting. 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.

```
def max_rod_cut(k):

    # max_prods[i] := largest product for cutting rod of length i
    max_prods = [0 for _ in range(k + 1)]
    max_prods[1] = 1 # base case. length 1 cannot be cut more

    for i in range(2, k + 1):

        best_prod = i # compare against not cutting at all
        for cut in range(2, i // 2 + 1):
            remaining = i - cut # the length remaining
            p = max_prods[cut] * max_prods[remaining]
            best_prod = max(best_prod, p)

        max_prods[i] = best_prod

    return max_prods[k]
```

3 Cheeseville

The streets of Cheeseville form an $k \times k$ grid, and one of k^2 mice lives at the corner of each intersection. Note that, in this grid, the *bottom left* corner corresponds to (0,0) (i.e. we zero-index rows and count from bottom-up instead of from the top-down).

You're about to place a big slice of cheese at the South-West corner of Cheeseville, after which all k^2 mice will immediately smell it and run to the cheese as fast as they can.

There is enough cheese for all the mice, but not all cheese is the same: the mice that arrive first get to eat the best, stinkiest parts of the cheese. The problem: if two mice arrive at exactly the same time, they will fight over who gets to eat next best part of the cheese.

All mice run at exactly the same speed. All mice always take a shortest (minimum number of blocks) route, i.e. they never run North or East. But there are some small delays on each block, so paths with the same number of blocks can take slightly different time - which you hope will help as tie-breakers between the mice racing for the cheese.

Your input is a pair of matrices: an $k \times (k-1)$ matrix W with delays going West, where W_{ij} is the delay from street-corner $(i, j+1)$ to street-corner (i, j), and a $(k-1) \times k$ matrix S, where S_{ij} is the delay going South from $(i+1, j)$ to (i, j). Design an algorithm that predicts ties, i.e. it finds a pair of mice that is expected to arrive at $(0, 0)$ at exactly the same time, or notifies that such a pair doesn't exist.

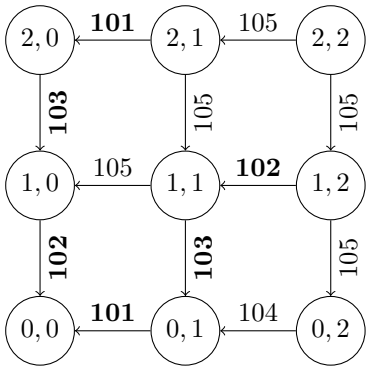
Input: A $k \times (k-1)$ matrix W with W_{ij} = the cost of going from $(i, j+1)$ to (i, j) and a $(k-1) \times k$ matrix S_{ij} = the cost of going from $(i+1, j)$ to (i, j).
Output: Either a pair $(i_1, j_1), (i_2, j_2)$ of mice that will arrive at exactly the same time, or $\emptyset$ if no ties will occur.

Example.

```
Input:

W = [
   [101, 104];  [105, 102];  [101, 105]
]
S = [
   [102, 103, 105];  [103, 105, 105]
]

Output:
[(1,2),(2,1)]
```



In the diagram above, North, East, South, and West are up, right, down, and left respectively. The mice living at $(1, 2)$ and $(2, 1)$ will arrive at $(0, 0)$ at the same time (the delays they encounter are bolded). The algorithm will output $[(1, 2), (2, 1)]$

We use **dynamic programming** to compute the length of the shortest path from each node to $(0, 0)$. Optimal sub-structure:

$$d(i, j) = \min\{d(i, j-1) + W(i, j-1), d(i-1, j) + S(i-1, j)\}.$$

We store all the distances in a **hash table** together with a pointer to the corresponding node. When we compute the distance for a new node, check to see if this distance is already in the hash table.

Pseudocode:

Algorithm 1: Tie-Detector(W, S)

Input: W: $k \times (k-1)$ matrix with cost of going from $(i, j+1)$ to (i, j); S: $(k-1) \times k$ matrix with cost of going from $(i+1, j)$ to (i, j).

Output: A pair $(i_1, j_1), (i_2, j_2)$ at exactly equal distance to $(0, 0)$ (or $\emptyset$ no such pair exists).

```
distances ← new hash table
/* Base cases                                                    */
d[0][0] ← 0
for j = 1 : k − 1 do
    d[0][j] ← d[0][j − 1] + W[0][j − 1]
    distances.add(d[0][j])
for i = 1 : k − 1 do
    d[i][0] ← d[i − 1][0] + S[i − 1][0]
    if distances.search(d[i][0]) then
        Return distances.search(d[1][0]).node, [i,0]
    else
        distances.add(d[i][0])

/* Main loop                                                     */
for i = 1 : k − 1 do
    for j = 1 : k − 1 do
        d[i][j] ← min{d[i][j − 1] + W[i][j − 1], d[i − 1][j] + S[i − 1][j]}
        if distances.search(d[i][j]) then
            Return distances.search(d[i][j]).node, [i,j]
        else
            distances.add(d[i][j])
Return ∅
```

Runtime analysis:

The algorithm iterates through the main loop $\underline{O(k^2)}$ times. Thanks to the hash table, each iteration only takes $O(1)$ time.