12.4 Module Exercise: Dynamic Programming

Ashlie Ossege (ajo5fs)
Diana McSpadden (hdm5s)

Part A:

Dynamic programming (DP) is the algorithm that breaks down your problem into subproblems, solves and stores the subproblem results for comparison and reuse as needed, and ultimately combines subproblem results to create an optimal solution. Comparing DP to baking baklava, your first element is the **“sub-structure”** where you can decompose the problem - individual ingredients into multiple sub-problems which are: 1) Preparing phyllo 2) Honey Sauce 3) Nut and spice mixture. Then the **“table-structure”** comes into play, and stores the elements in a table to be used multiple times, like separate bowls of honey sauce and mixture being reached back into to apply to each layer. Instead of getting the honey out of the cabinet, mixing it with lemon and returning it for each phyllo layer, we save the honey and nut mix in bowls we continue to use as we assemble, thus, saving time. Finally, the **“bottom-up computation”** of assembling and baking. Layer the bottom of the pan and build up your baklava – or in other words, combining prepared sub-problem results and eventually arrive at the optimal solution to the complete problem – delicious baklava.

Part B:

Floyd's Algorithm for All-Pairs Shortest Paths

We are investigating Floyd's algorithm, an efficient algorithm for determining the shortest path in a graph that is both weighted and directed. A useful analogy is determining the shortest driving distance between locations traveling through other locations. Floyd's algorithm takes into account one-way roads (the "directed" aspect), and distances between intermediate points (the "weighted" aspect). In fact, this "traveling" problem is an example of a use case for Floyd's algorithm.

What problems does this algorithm solve?

While Floyd's algorithm can be used to find the shortest path in a traffic scenario, other algorithms are more efficient in scenarios where "distance" is assumed to be positive, such as real world driving problems. However, Floyd's algorithm, unlike other shortest-path algorithms, can be used when there are "distances" or costs with negative values in the graph. While negative distances do not exist in the simple traffic problem, imagine a toy problem of finding "least costly" driving distance where a "cost" is subtracted from distance if a route does **not** include a tollway. In this scenario one would require an algorithm that can evaluate negative costs. In addition to working with negative values Floyd's Algorithm can be retooled to find a "maximum flow" path by checking not for the min, but for the max. An example where this is useful is optimal routing (think electricity or bandwidth).

Dynamic Programming

The Initial problem is to find the length of the shortest path between every pair of vertices. **Dynamic programming** is a part of this algorithm in the following ways:

Sub-structures:

- Solving the sub problems of each vertex to each other vertex.

Table-structures:

- Developing an adjacency matrix, containing connectivity information, with n vertices, and assigning ∞ for nonexistent edges (where pair distances are not possible).

```
In [2]: adjMatrix
Out[2]:
[[0, 2, 5, inf, inf, inf],
 [inf, 0, 7, 1, inf, 8],
 [inf, inf, 0, 4, inf, inf],
 [inf, inf, inf, 0, 3, inf],
 [inf, inf, 2, inf, 0, 3],
 [inf, 5, inf, 2, 4, 0]]
```

- The direct path length (stored weights from each path) from vertex i to vertex j is stored as a[i, j]

```
In [2]: distanceMatrix
Out[2]:
[[0, 2, 5, 3, 6, 9],
 [inf, 0, 6, 1, 4, 7],
 [inf, 15, 0, 4, 7, 10],
 [inf, 11, 5, 0, 3, 6],
 [inf, 8, 2, 5, 0, 3],
 [inf, 5, 6, 2, 4, 0]]
```

Bottom-up computation:

- Compare direct path or “detour” path (k), and store the minimum value of the two path calculations.

```
def floyd(A):
    n = len(A)

    # update the distance matrix with the shortest distance from row vertex to column vertex
    for k in range(n):
        for i in range(n):
            for j in range(n):
                # choose the shortest (i.e. minimum) of the current item of interest,
                # or other items in row + other items in rows
                A[i][j] = min(A[i][j], (A[i][k] + A[k][j]))

    return A
```

Effectiveness of algorithm

Floyd's algorithm, which specifically breaks down the main problem into smaller ones, is crucial to efficiently solving the shortest path in a weighted and directed graph. Solving the problem would not be effective if dynamic programming was not used in cases where graphs contain negative weights, or required shortest distances between all vertices, versus a single vertex (as in Dijkstra’s algorithm). Without a dynamic program algorithm, the shortest path between pairs problems would have to be solved with a brute force implementation, which needs to check all possible paths, and there are n x n! possible paths in a graph with n vertices. In addition to determining the paths, the weight of each path needs to be calculated.

Sources:

1. https://www.techiedelight.com/pairs-shortest-paths-floyd-warshall-algorithm/
2. http://www.csl.mtu.edu/cs4321/www/Lectures/Lecture%2016%20-%20Warshall%20and%20Floyd%20Algorithms.htm

3. http://www.cs.umsl.edu/~sanjiv/classes/cs5740/lectures/floyd.pdf
4. https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
5. https://www.uio.no/studier/emner/matnat/ifi/INF3380/v12/undervisningsmateriale/inf3380-floyd.pdf
6. https://www.youtube.com/watch?v=3wmsXtSUm8Y