Assignment Four

Ethan Morton

Ethan.Morton1@Marist.edu

December 2, 2024

1 Graph

The graph has been modified since assignment 3. Now that linked objects are the only implementation for graphs, the Graph struct no longer uses template specializations, and it's just a graph with linked o objects. The Vertex struct has also been modified to make the Bellman-Ford algorithm more efficient and easier to write. Specifically, it now contains properties for distance and predecessor, which are both ints.

graph.hpp

```
16 //Vertex for Graph with Linked Objects
17 struct Vertex {
18     int id;
19     //pointer to adjacent vertex and weight
20     vector<pair<Vertex*, int>> neighbors;
21     int distance;
22     int predecessor;
23 
24     explicit Vertex(const int id) : id(id), neighbors(vector<pair<Vertex*,
       ↪  int>>()), distance(INT_MAX), predecessor(-1) {}
25 
26     void reset() {
27         distance = INT_MAX;
28         predecessor = -1;
29     }
30 };
```

1.1 Bellman-Ford Single-Source Shortest Path

This algorithm finds the shortest path from one vertex to another, by finding the minimum sum of the weights between the vertices in the path. All vertices have a distance of INT_MAX (which represents infinity) and

a predecessor of -1 (which represents no predecessor). The source vertex is initialized with a distance of 0 because it should cost nothing to get to itself.

The next thing to do is to go over each edge in the graph and relax them. This is done by iterating over each neighbor of each vertex, and we have to do this $V - 1$ times because the relaxation is different for each path. The algorithm for relaxation is for each edge, if the distance to the destination vertex via the current vertex is shorter than the known distance, the distance and predecessor of the destination vertex are updated.

After relaxation, we have to check for negative weight cycles. If the weight from A to B is 2 and the weight from B to A is -5, then we could just keep cycling between A and B, and it would subtract 3 from the total cost each time we do it, and we could just do it infinitely many times. If this occurs, we just assume that the shortest path cannot be found. We again have to check each edge for a negative weight cycle, but we only have to do it once.

Finally, we can print out all the shortest paths. We go through each vertex (except the source vertex), and loop through their predecessors until we get to the source vertex. During the loop, the vertex IDs are pushed to a vector. To print out the path, we iterate backwards through the vector to print each ID.

When the function returns, either due to the detection of a negative weight cycle or completion of the algorithm, we need to reset the vertices so the graph can do this algorithm again.

The time complexity of the algorithm is dominated by the relaxation step. It iterates over each neighbor of each vertex $V - 1$ times. Although it uses a triple-nested loop, one of them is used to iterate over each vertex, and the other two combines is what lets us iterate over each edge in the entire graph. This gives a time complexity of $O(V * E)$.

graph.hpp

```
49 void bellman_ford(const int srcID) {
50     // Initialize source vertex distance
51     vertices[id_lookup[srcID]].distance = 0;
52 
53     // Relax edges V - 1 times
54     for (int i = 0; i < vertices.size() - 1; i++) {
55         for (Vertex& vertex : vertices) {
56             for (const pair<Vertex*, int>& neighbor : vertex.neighbors) {
57                 // Relaxation step
58                 if (const int weight = neighbor.second; vertex.distance !=
                    ↪  INT_MAX && vertex.distance + weight <
                    ↪  neighbor.first->distance) {
59                     neighbor.first->distance = vertex.distance + weight;
60                     neighbor.first->predecessor = vertex.id;
61                 }
62             }
63         }
64     }
65 
66     // Check for negative-weight cycles
67     for (const Vertex& vertex : vertices) {
68         for (const pair<Vertex*, int>& neighbor : vertex.neighbors) {
```

```
            if (const int weight = neighbor.second; vertex.distance !=
             ↪  INT_MAX && vertex.distance + weight <
             ↪  neighbor.first->distance) {
                fprintf(stderr, "Graph contains a negative-weight cycle");
                for (Vertex& v : vertices) {
                    v.reset();
                }
                return;
            }
        }
    }

    // Print shortest paths
    for (const Vertex& vertex : vertices) {
        if (vertex.id == srcID) {
            // Skip the source vertex
            continue;
        }
        if (vertex.distance == INT_MAX) {
            printf("%d --> %d cost is ; path: none\n", srcID, vertex.id);
            continue;
        }
        // Reconstruct the path
        vector<int> path;
        for (int id = vertex.id; id != -1; id =
         ↪  vertices[id_lookup[id]].predecessor) {
            path.push_back(id);
        }

        // Print the result
        printf("%d --> %d cost is %d; path: ", srcID, vertex.id,
         ↪  vertex.distance);
        for (int i = static_cast<int>(path.size()) - 1; i >= 0; i--) {
            printf("%d", path[i]);
            if (i > 0) {
                printf(" --> ");
            }
        }
        printf("\n");
    }
    for (Vertex& v : vertices) {
        v.reset();
    }
}
```

2 Spice Heist

I created a spice struct to keep track of the information from spice.txt.

spice.hpp

```
8  struct Spice {
9      string name;
10     float total_price;
11     float quantity;
12 };
```

Each spice is parsed and pushed to a vector. Then each knapsack is parsed, and the capacity is pushed to a vector of ints.

To begin the algorithm, the spices vector is sorted by price per unit in descending order. This sort has a time complexity of $O(nlog_2(n))$. The reason for sorting is that in the fractional knapsack problem, you can take as much spice as you want until the pile runs out or the knapsack is at capacity. If the pile runs out and you still have capacity, then you can move on to the next pile. If the piles are sorted by price per unit, we can just iterate over the sorted spices and take from the most valuable piles first. When iterating, we need to keep track of how much space is left in the knapsack, and every time we take from a pile, we take the minimum value between how much spice is in the pile, and how much capacity is remaining. If we have taken the entire pile and there's capacity remaining, we move on to the next pile. If there is no capacity remaining, we break from the loop. This iteration has a time complexity of $O(n)$ since the worst case scenario requires that we iterate over each spice in the vector to fill a knapsack that can hold all the spice. We keep track of how much of each spice was taken, and it's printed out at the end. All this is done for each knapsack given in spice.txt.

main.cpp

```
190 int main(int argc, char* argv[]) {
191
192     //...
193
194     std::sort(spices.begin(), spices.end(), [](const Spice& a, const Spice&
        ↪  b) {
195         return a.total_price / a.quantity > b.total_price / b.quantity;
196     });
197     for (const float capacity : knapsacks) {
198         float remaining_cap = capacity;
199         float total_value = 0;
200         vector<Spice> spices_taken;
201         for (int i = 0; i < spices.size() && remaining_cap > 0; i++) {
202             Spice taken = spices[i];
203             float quantity = taken.quantity;
204             taken.quantity = std::min(quantity, remaining_cap);
205             remaining_cap -= taken.quantity;
206             total_value += taken.total_price * (taken.quantity / quantity);
207             spices_taken.emplace_back(taken);
208         }
```

```
209         printf("Knapsack of capacity %.1f is worth %.2f quatloos and
             ↪  contains ", capacity, total_value);
210         for (size_t i = 0; i < spices_taken.size(); i++) {
211             printf("%.1f scoops of %s", spices_taken[i].quantity,
                 ↪  spices_taken[i].name.c_str());
212             if (i != spices.size() - 1) {
213                 printf(", ");
214             }
215         }
216         printf("\n");
217     }
218 
219     return 0;
220 }
```