Assignment Five

Josh Seligman

joshua.seligman1@marist.edu

November 20, 2022

1 Bellman-Ford Single Source Shortest Path

1.1 The Algorithm

The Bellman-Ford single source shortest path (SSSP) algorithm computes the shortest path from a single vertex in a directed and weighted graph to every other vertex that is connected to the source through a series of edges. As displayed in Algorithm 1, the Bellman-Ford routine uses dynamic programming to compute the shortest path to each vertex in the graph from the source. In other words, rather than computing and then comparing each possible path in the graph, the Bellman-Ford algorithm relies on neighboring vertices to share their known distances with each other, which would cause the best paths to spread throughout the graph so each vertex can take advantage of these routes rather than trying to determine it completely on their own. The only problem with the Bellman-Ford SSSP algorithm, however, is the case of a negative weight cycle. This is checked for on lines 9-13 of Algorithm 1. This edge case causes issues because going to a vertex via a negative weight and then looping back to the first vertex with a smaller positive weight will cause the algorithm to believe the path to the vertex is $-\infty$ if the loop on lines 3-7 did not terminate. An example negative weight loop is shown in Figure 1.1. In this image, if the path from vertex 1 to the source node goes through vertex 2, which goes through vertex 3 and so on, Since the negative weight from vertex 2 to 1 has a greater magnitude than the positive weight from vertex 1 to 2, one could theoretically go through the loop an infinite number of times to decrease the distance of vertex 1 to the source. It is for this reason that the algorithm will return false upon finding a negative weight cycle as the edge case prevents the algorithm from having an accurate representation of the distance from a vertex affected by the negative weight cycle to the source.

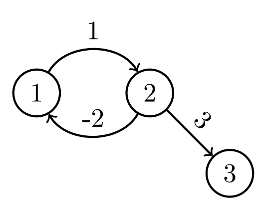

Figure 1.1: Sample negative weight-loop.

Algorithm 1 Bellman-Ford single source shortest path algorithm.

```
procedure BellmanFord(graph, weightFunction, sourceVertex)
    InitSingleSource(graph, sourceVertex)    // Initialize the vertices to do the algorithm
    for i ← 0, i < length(graph.vertices) − 1, i++ do
        for edge ϵ graph.edges do    // Iterate through all the edges
            Relax(edge.fromVertex, edge.toVertex, weightFunction)    // Make a better decision if needed
        end for
    end for
    out ← true    // Assume all went well
    for edge ϵ graph.edges do
        if edge.toVertex.distance > edge.fromVertex.distance+weightFunction(edge.fromVertex, edge.toVertex) then
            out ← false    // There is a negative weight loop, so the algorithm failed
        end if
    end for
    return out
end procedure
procedure InitSingleSource(graph, sourceVertex)
    for vertex ϵ graph.vertices do
        vertex.distance ← ∞    // Assume each vertex has no path to the source
        vertex.predecessor ← null
    end for
    sourceVertex.distance ← 0    // The source vertex has a distance of 0 to itself
end procedure
procedure Relax(fromVertex, toVertex weightFunction)
    if toVertex.distance > fromVertex.distance + weightFunction(fromVertex, toVertex) then
        toVertex.distance ← fromVertex.distance + weightFunction(fromVertex, toVertex)
        toVertex.predecessor ← fromVertex    // There is a better route to toVertex through fromVertex
    end if
end procedure
```

1.2 Asymptotic Analysis

2 Fractional Knapsack Algorithm

2.1 The Algorithm

The fractional knapsack algorithm solves the problem of maximizing the value of the objects one takes to fill up their knapsack. As displayed in Algorithm 2, the solution to the fractional knapsack problem requires a greedy approach by getting the most valuable spice available at each point the algorithm. To do this, the spices are sorted by their unit price to get the most value for the amount of spice taken. The loop on lines 7-18 will continue until the knapsack is full or until there are no more spices to consider and, as previously mentioned, will take as much of the most valuable spice that is available. Since the spices are sorted ahead of time, the greedy approach of taking the local maximum value and hope it leads to a global maximum value works because the spices are not changing and the unit prices of the spices taken will continue to decrease as the algorithm is run. This ensures that the global maximum value is always achieved for any set of spices and any knapsack capacity.

Algorithm 2 Fractional Knapsack algorithm.

```
1: procedure FractionalKnapsack(spices, capacity)
2:     sort(spices)    // Sort the spices by unit value, descending order
3:     quantityTaken ← new int[spices.length]    // Store an array to keep track of how much of each spice
       was taken
4:     capacityLeft ← capacity    // Start with empty knapsack
5:     totalValue ← 0    // Start off with no value
6:     curSpiceIndex ← 0    // Start with most valuable spice per unit
7:     while capacityLeft > 0 && curSpiceIndex < spices.length do
8:         if capacityLeft ≥ spices[curSpiceIndex].quantity then    // Enough space to take everything
9:             quantityTaken[curSpiceIndex] ← spices[curSpiceIndex].quantity
10:            capacityLeft ← capacityLeft − spices[curSpiceIndex].quantity
11:            totalValue ← totalValue + spices[curSpiceIndex].value
12:        else    // Take what we can
13:            quantityTaken[curSpiceIndex] ← capacityLeft
14:            totalValue ← totalValue + capacityLeft ∗ spices[curSpiceIndex].unitPrice
15:            capacityLeft ← 0
16:        end if
17:        curSpiceIndex ← curSpiceIndex + 1    // Move on to next spice
18:    end while
19:    return quantityTaken, totalValue
20: end procedure
```

2.2 Asymptotic Analysis

Listing 1 contains the C++ implementation of the fractional knapsack algorithm. First, line 3 makes a call to a quicksort algorithm for the spices to put them in descending order by their unit prices, which runs in $O(s*log_2 s)$ time, where s is the number of spices available to be taken. Next, the loop defined on line 5 iterates iterates through each of the knapsacks because the implementation takes in many knapsacks for testing. This loop will run k times, where k is the number of knapsacks. The sorting algorithm is executed before the loop because all of the knapsacks are being filled with the same data and, therefor, the sorting operation only has to be done once. Line 6 is also specific to the implementation as it dequeues the next knapsack from the queue, which is a constant time operation. Lines 9, 15, 16, and 19 all define variables to keep track of, which are all constant time operations. The small loop on lines 10-12 initializes the array to use all 0s, which is a

C++ specific problem as other programming languages do this automatically, which will cause these lines to be excluded from the asymptotic analysis. Next, the loop defined on line 22 continues until the knapsack is full or until there are no more spices. The worst case scenario is when there is not enough spice to fill the knapsack, which causes the loop to run a total of s times. The entire body of the loop matches what is done in Algorithm 2, which is only the constant time operations of conditions and assignments. Thus, the entire loop on lines 22-43 runs in $O(s)$ time. Lastly, lines 46-62 will be excluded as they are outputting the results. Overall, for each individual knapsack, the runtime complexity is $O(s * log_2 s + s)$, which is $O(s * log_2 s)$ because $s * log_2 s$ is the dominant term in the expression. However, the implementation runs the $O(s)$ loop k times, which will cause the overall runtime complexity of the C++ implementation to become $O(s * log_2 s + k * s)$.

3 Appendix

3.1 Fractional Knapsack Algorithm

```
1  void runAlgo(SpiceArr* spices, Queue<int>* knapsacks) {
2      // Start off by running a sort on the spices array to make them in descending order
3      quickSort(spices);
4
5      while (!knapsacks->isEmpty()) {
6          Node<int>* curKnapsack = knapsacks->dequeue();
7
8          // Create an array that corresponds with the spice array for keeping track of what
               was taken by the knapsack
9          int quantityTaken[spices->length];
10         for (int i = 0; i < spices->length; i++) {
11             quantityTaken[i] = 0;
12         }
13
14         // Start off with an empty knapsack and a value of 0
15         int capacityLeft = curKnapsack->data;
16         double spiceValue = 0;
17
18         // Start considering the first element in the array (most valuable per unit)
19         int spiceIndex = 0;
20
21         // Continue until the knapsack is full or until there is no more spice to take
22         while (capacityLeft > 0 && spiceIndex < spices->length) {
23             // If there is space for the entire pile of spice, take it all
24             if (capacityLeft >= spices->arr[spiceIndex]->getQuantity()) {
25                 // Update the array of spice taken
26                 quantityTaken[spiceIndex] = spices->arr[spiceIndex]->getQuantity();
27
28                 // Be greedy and take everything available if possible
29                 capacityLeft -= spices->arr[spiceIndex]->getQuantity();
30                 spiceValue += spices->arr[spiceIndex]->getPrice();
31             } else {
32                 // Update the table entry
33                 quantityTaken[spiceIndex] = capacityLeft;
34
35                 // Compute the value of the spice we can take
36                 spiceValue += capacityLeft * spices->arr[spiceIndex]->getUnitPrice();
37
38                 // Update the capacity to be 0
39                 capacityLeft = 0;
40             }
41             // Go on to the next spice
42             spiceIndex++;
43         }
44
45         // Start with this text
```

```cpp
        std::cout << "Knapsack of capacity " << curKnapsack->data << " is worth " << spiceValue << " quatloos and contains";

        // Iterate through all of the spices
        for (int j = 0; j < spices->length; j++) {
            // Only print out the spices we take
            if (quantityTaken[j] > 0) {
                // Little formatting logic
                if (j > 0) {
                    std::cout << ", ";
                } else {
                    std::cout << " ";
                }
                // The amount and name of the spice taken
                std::cout << quantityTaken[j] << " scoops of " << spices->arr[j]->getName();
            }
        }
        std::cout << "." << std::endl;

        // Memory management
        delete curKnapsack;
    }
}
```

Listing 1: Fractional Knapsack Algorithm (C++)