Assignment Four – Dynamic & Greedy

Ryan Munger
Ryan.Munger1@marist.edu

November 22, 2024

1 Introduction

1.1 Goals

Assignment 4 focuses on dynamic programming and greedy algorithms. First, I have to load in several weighted, directed graphs by parsing a file of instructions. Then, I must implement the Bellman-Ford dynamic programming algorithm for Single Source Shortest Path (SSSP) in order to find the optimal path from a source vertex to each other vertex in the graph. Graphs aside, the greedy algorithm I need to implement is a version of the fractional knapsack problem. I need to read in a file that contains information about spices (price, quantity, etc.) and I must conduct a spice heist on Arrakis for each knapsack provided, maximizing my take.

1.2 Write-up Format

In this report I will describe the logic being presented and the asymptotic running time of the algorithms implemented. Below the text explanation, relevant code will follow in C++.

1.3 Limerick of Luck

To maximally fill your knapsack
And remain unburdened with a fallback,
    One must employ an algo of greed,
    Your capacity will not exceed,
A heist for the ages, engraved on a plaque.

2 Weighted Directed Graphs

2.1 Reading the Instructions

I was provided a file of instructions to create weighted directed graphs such as *add vertex* and *add edge*. I was able to utilize most of the code from my previous assignment (with different regex) to create my graph. This time, I only had to create a linked object representation.

```
void createGraphs(const string& filename) {
    int graphCount = 1;
    cout << "Graph #" << graphCount << ":\n" << endl;
    Graph currentGraph(to_string(graphCount));
    regex newGraphRe(R"(new graph)");
    regex addVertexRe(R"(add\s*vertex\s*(\S+))");
    regex addEdgeRe(R"(add\s*edge\s*(\S+)\s*-\s*(\S+)\s+(-?\d+))");

    ifstream file(filename); // input file stream
    if (!file) {
        cerr << "File opening failed." << endl;
    }
    string instruction;

    while (getline(file, instruction)) {
        // ignore any commands we don't know, empty lines, comments etc.
        // regex will allow some slack with white space, but assuming perfect syntax by user
        // case 1: start a new graph
        if (regex_match(instruction, newGraphRe)) {
            // check to see if the current graph has anything in it
            // if not, no need to start a new one
            if (!currentGraph.isEmpty()) {
                currentGraph.displayGraph();
                graphCount++;
                cout << "\n\nGraph #" << graphCount << ": \n" << endl;
                currentGraph = Graph(to_string(graphCount)); // start a new graph
            }
        } else {
            smatch match; // captures subexpressions/groups

            if (regex_match(instruction, match, addVertexRe)) { // case 2: new vertex
                string newVertex = match[1].str();
                currentGraph.addVertex(newVertex);
            } else if (regex_match(instruction, match, addEdgeRe)) { // case 3: new edge
                string v1 = match[1].str();
                string v2 = match[2].str();
                int weight = stoi(match[3].str());
                currentGraph.addEdge(v1, v2, weight);
            };
        };
    };
```

```
    file.close();
    currentGraph.displayGraph();
};
```

2.2 Graph Object

The graph object is even simpler than last time. Each graph has an ID and a map of linked vertex objects. This time, since my graph is weighted, I stored each edge in the neighbor list as a tuple. Adding vertices and edges is much simpler when we only have one representation to update! Since this is a directed graph, we only need to update one neighbor list. I made a new graph display function to ensure the graphs were created correctly.

```
struct linkedVertex {
    string id;
    int distance; // for SSSP
    linkedVertex* predecessor; // for SSSP
    vector<tuple<linkedVertex*, int>> neighbors; // no limit to neighbors!
};

void printLinkedVertex(linkedVertex v) {
    cout << "LinkedVertex " << v.id << "; Neighbors: " << endl;
    if (v.neighbors.empty()) {
        cout << "\tNo Neighbors" << endl;
    } else {
        for (const auto& tuple : v.neighbors) {
            cout << "\tVertex: " << get<vertexTupleIdx>(tuple)->id << " Weight: " << get<weightTupleIdx>(tuple) << endl;
        }
    }
};
```

```
    public:
        Graph(string id) {
            this->graphID = id;
        };

        void addVertex(string vertex) {
            linkedObjs[vertex] = linkedVertex{vertex}; // store by value
        };

        void addEdge(string vertex1, string vertex2, int weight) {
            this->linkedObjs[vertex1].neighbors.push_back(make_tuple(&linkedObjs[vertex2], weight));
        };

        bool isEmpty() {
            return this->linkedObjs.empty();
        };

        void displayGraph() {
            // print graph objects to ensure validity
            for (const auto& pair : this->linkedObjs) {
```

```
                printLinkedVertex(pair.second);
            }

            this->SSSP();
        };
```

2.3 SSSP

The Single Source Shortest Path (SSSP) algorithm aims to find the shortest paths from a single source vertex to all other vertices in a graph. This is a powerful algorithm in scenarios like routing and navigation systems. Two important algorithms for SSSP are Dijkstra's algorithm and the Bellman-Ford algorithm, which I have implemented in this lab.

1. Initialize single source: $O(|V|)$.
2. Relax all edges $|V| - 1$ times: $O(|V| * |E|)$.
3. Check for negative weight cycles: $O(|E|)$.
4. Report the optimized paths (Not part of the algorithm directly).

I will explain the time complexities for each portion later. Combining the complexities of the algorithm's subroutines, the time complexity of Bellman-Ford is $O(|V| * |E|)$ where V is the set of vertices and E is the set of edges.

Initialize Single Source

1. Assign an initial distance of infinity to all vertices.
2. Set the source vertex distance to 0.

Why? This ensures that the shortest distance to the source vertex itself is 0, and all other vertices start with an "infinite" distance so that any path we compute is "cheaper." Since we traverse all the vertices once, this operation is $O(|V|)$.

```
        void initSingleSource(linkedVertex* s) {
            // set all vertices to distance infinite (large but not max)
            // no predecessors yet
            for (auto& pair : this->linkedObjs) {
                pair.second.distance = functionalInfinity;
                pair.second.predecessor = nullptr;
            }
            // set single source
            s->distance = 0;
        };
```

Relaxing Edges

For each edge (*source*, *destination*) in the graph, check if the path from *source* to *destination* through that edge is better than the current distance to *destination*. If so, update *destination.distance* to the shorter value and set *source* as the predecessor of *destination*.

This process is called "relaxing" an edge. This is the main driver of Bellman-Ford because it allows us to find shorter paths and record them. We relax every edge $|V|-1$ times because in the worst case, the optimal path can have $|V|-1$ edges in it. Since we relax each edge essentially $|V|$ times, this is a costly operation at $O(|V|*|E|)$.

```
        // find shortest path by recording the optimal choice
        void relax(linkedVertex* source, linkedVertex* destination,
     int weight) {
            if (destination->distance > (source->distance + weight)
    ) {
                destination->distance = (source->distance + weight)
    ;
                destination->predecessor = source;
            }
        };
```

Why did the dynamic algorithm need scrap paper for its exam? *To write down all of its intermediate answers...*

Detecting Negative Weight Cycles

After we relax all of the edges $|V|-1$ times, we must iterate over them once more. If we can still relax an edge, this indicates the presence of a negative weight cycle. We return false if this is the case, as it makes it impossible to reliably report shortest paths with this algorithm. A negative weight cycle occurs when there is a closed path (a loop or cycle) between two edges that has a negative cost. In this case, traveling around this loop over and over would reduce your cost indefinitely. The shortest path would be to follow this cycle infinite times before continuing on to your destination. Since this is a single traversal of the edges, this costs $O(|E|)$.

```
            // detect negative weight cycles
            for (auto& pair : this->linkedObjs) {
                linkedVertex* current = &pair.second;
                for (auto& edge : current->neighbors) {
                    linkedVertex* destination = get<vertexTupleIdx
    >(edge);
                    int weight = get<weightTupleIdx>(edge);

                    if (destination->distance > (current->distance
    + weight)) {
                        return false; // negative weight cycle
    found
                    }
                }
            }
```

```
            // relax all edges |V| - 1 times
            for (size_t i = 1; i < this->linkedObjs.size(); ++i) {
                for (auto& pair : this->linkedObjs) {
                    linkedVertex* current = &pair.second;
                    for (auto& edge : current->neighbors) {
                        linkedVertex* destination = get<
    vertexTupleIdx>(edge);
                        int weight = get<weightTupleIdx>(edge);
                        relax(current, destination, weight);
                    }
                }
            }
```

Report the Paths

Now that we have computed the shortest paths, we have to extract this data from our linked objects. The way to do this is simple: we can just check the predecessor of the destination, its predecessor, and so on until we are back at the source. Since this will be revealed in reverse order, pushing them onto a stack and then popping it will make them human readable. I did not count this as part of the algorithm's time complexity, as the necessary operations do no include this. However, this operation will take $O(|V|)$ since in the worst case we will have to pass through every vertex as a predecessor.

```
        string getShortestPath(linkedVertex* destination) {
            // follow predecessors (reverse order)
            stack<string> pathStack;
            linkedVertex* predecessor = destination;
            while (predecessor != nullptr) {
                pathStack.push(predecessor->id);
                predecessor = predecessor->predecessor;
            }

            // put them in forward order for display
            ostringstream pathstr;
            while (!pathStack.empty()) {
                // Check if something is already in the stream for
    ->
                if (pathstr.tellp() > 0) {
                    pathstr << "->";
                }
                pathstr << pathStack.top();
                pathStack.pop();
            }
            return pathstr.str();
        };

        void SSSP() {
            linkedVertex* startVertex = &this->linkedObjs.begin()->
    second;
            // set distances, predecessors, etc
            bellmanFord(startVertex);

            cout << "SSSP: " << endl;
            for (auto& pair : this->linkedObjs) {
                linkedVertex* current = &pair.second;
```

```
                cout << startVertex->id << "->" << current->id << " cost is " << setw(2) <<
                    current->distance << "; shortest path is " << getShortestPath(current) << endl;
            }
        };
```

Why did they add a timer to chess? *Mr. Dy Namic Algorithm...*

2.4 Graphs in Action

I have included one such graph below. This graph contains no negative weight cycles. As we can see, the graph was loaded in correctly and the shortest paths from the source to each other vertex was computed effectively.

```
Graph #1:

LinkedVertex 1; Neighbors:
  Vertex: 2 Weight: 6
  Vertex: 4 Weight: 7
LinkedVertex 2; Neighbors:
  Vertex: 3 Weight: 5
  Vertex: 4 Weight: 8
  Vertex: 5 Weight: -4
LinkedVertex 3; Neighbors:
  Vertex: 2 Weight: -2
LinkedVertex 4; Neighbors:
  Vertex: 3 Weight: -3
  Vertex: 5 Weight: 9
LinkedVertex 5; Neighbors:
  Vertex: 3 Weight: 7
  Vertex: 1 Weight: 2

SSSP:
1->1 cost is  0; shortest path is 1
1->2 cost is  2; shortest path is 1->4->3->2
1->3 cost is  4; shortest path is 1->4->3
1->4 cost is  7; shortest path is 1->4
1->5 cost is -2; shortest path is 1->4->3->2->5
```

3 Greedy Knapsack

Why wouldn't the greedy algorithm move?- *Staying local is important to him...*

3.1 Gathering Information

I gathered information about spices and knapsacks with regex in a similar fashion to my graphs. I stored my spices in a vector of Spice objects and my knapsacks in a vector as well. I used float values for everything, as this is the *fractional* knapsack problem, and there is no reason quantities and capacities cannot be decimal.

```
struct Spice {
    string color;
    float total_price;
    float quantity;
    float unit_price;
};
```

```
    cout << "\n\nLoading in Spices and Knapsacks!" << endl;
    regex spiceRe(R"(\s*spice\s*name\s*=\s*(\S*)\s*;\s*total_price\s*=\s*(\d*.?\d*)\s*;\s*qty\s*=\s*(\d*.?\d*)\s*;)");
    regex knapsackRe(R"(knapsack\s*capacity\s*=\s*(\d*.?\d*)\s*;)");

    // store spices and knapsacks
    vector<Spice> spiceInventory;
    vector<float> knapsacks;

    ifstream file(filename); // input file stream
    if (!file) {
        cerr << "File opening failed." << endl;
    }
    string instruction;

    while (getline(file, instruction)) {
        smatch match; // captures subexpressions/groups
        // case 1: adding a spice
        if (regex_match(instruction, match, spiceRe)) {
            string color = match[1].str();
            float total_price = stof(match[2].str());
            float quantity = stof(match[3].str());
            float unit_price = total_price / quantity;

            Spice newSpice = Spice{color, total_price, quantity, unit_price};
            printSpice(newSpice);
            spiceInventory.push_back(newSpice);
        } else if (regex_match(instruction, match, knapsackRe)) { // case 2: knapsack
            float newKnapsackCapacity = stof(match[1].str());
            knapsacks.push_back(newKnapsackCapacity);
```

```cpp
            cout << "New Knapsack: " << newKnapsackCapacity << endl;
        };
    };
    file.close();
    // sort our spices based on unit price
    spiceSort(spiceInventory);
    // maximize take for each knapsack!
    cout << "\nMaximizing Take:" << endl;
    for (float knapsack : knapsacks) {
        maximizeTake(knapsack, spiceInventory);
    }
};
```

3.2 Organizing Spice

To maximize take, we will examine the unit price of each spice (how much it is worth per quantity). To do this, we first sort the Spice list. I made a custom version of insertion sort (for simplicity) to accomplish this. I put it in descending order. Since I used insertion sort, this action will take $O(n^2)$ time due to the nested loop. We can use a better sorting algorithm, such as merge or quick sort, to optimize this down to $O(log(n))$.

```cpp
// Insertion sort to get descending order based on unit price
void spiceSort(vector<Spice>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; i++) {
        int insertIdx = i;
        Spice currentCheck = arr[i];
        for (int j = i-1; j >= 0; j--) {
            if (arr[j].unit_price < currentCheck.unit_price) {
                arr[j+1] = arr[j];
                insertIdx = j;
            } else {
                break;
            }
        }
        arr[insertIdx] = currentCheck;
    }
};
```

3.3 Maximizing Take

I implemented a greedy algorithm. This class of algorithm takes locally optimal choices and hopes for a globally optimal solution. In this case, we will achieve a globally optimal solution by pillaging as much of the highest value spice we can fit, then the next, and so on. This algorithm will only take $O(n)$ time as it is simply a single traversal of the spice list. It is even less than a single traversal, as we can expect most knapsacks to fill up before we reach the end of our spice inventory list! Thus, fractional knapsack is a $O(nlog(n)$ algorithm if you count sorting the spice list, or $O(n)$ on its own.

1. Examine the most valuable spice.
2. If we have no more knapsack capacity, we are done.
3. If we have more capacity than quantity of that spice, take everything! Record our scoops.
4. If we have less capacity than the quantity of that spice, take as much as we can fit. Record scoops.
5. Move to the next most valuable spice and repeat.
6. Finally, report on our knapsack value and scoops taken.

```
void maximizeTake(float knapsack, vector<Spice> spices) {
    float knapValue = 0;
    if (knapsack == 0) {
        cout << "Knapsack of Capacity " << fixed << setprecision(2) << knapsack << " is worth " <<
            fixed << setprecision(2) << knapValue << " quatloos and contains no scoops." << endl;
        return;
    }
    ostringstream scoops;
    float capacityLeft = knapsack;
    for (Spice spice : spices) {
        if (capacityLeft == 0) {
            break; // no more spice!!
        } else if(capacityLeft >= spice.quantity) { // take all the spice
            capacityLeft -= spice.quantity;
            knapValue += spice.total_price;
            scoops << fixed << setprecision(2) << spice.quantity << " scoops of " << spice.color << ", ";
        } else if (capacityLeft < spice.quantity) { // take what we can fit
            knapValue += capacityLeft * spice.unit_price;
            scoops << fixed << setprecision(2) << capacityLeft << " scoops of " << spice.color << ", ";
            capacityLeft = 0;
        }
    }
    string scoopString = scoops.str();
```

```
    // replace last comma with period
    scoopString.pop_back();
    scoopString.back() = '.';

    cout << "Knapsack of Capacity " << fixed << setprecision(2) <<
    knapsack << " is worth " <<
        fixed << setprecision(2) << knapValue << " quatloos and
    contains " << scoopString << endl;
};
```

3.4 Greed in Action

Spices were loaded, knapsacks were created, and heists were completed! For each knapsack, the most optimal scoops were scooped. I think "scamped" would be cooler to say though. Optimal scoops were scamped.

```
Loading in Spices and Knapsacks!
Spice:
  Color: red
  Total Price: 4
  Quantity: 4
  Unit Price: 1
Spice:
  Color: green
  Total Price: 12
  Quantity: 6
  Unit Price: 2
Spice:
  Color: blue
  Total Price: 40
  Quantity: 8
  Unit Price: 5
Spice:
  Color: orange
  Total Price: 18
  Quantity: 2
  Unit Price: 9
New Knapsack: 1
New Knapsack: 6
New Knapsack: 10
New Knapsack: 20
New Knapsack: 21

Maximizing Take:
Knapsack of Capacity 1.00 is worth 9.00 quatloos and contains 1.00
    scoops of orange.
Knapsack of Capacity 6.00 is worth 38.00 quatloos and contains 2.00
     scoops of orange, 4.00 scoops of blue.
Knapsack of Capacity 10.00 is worth 58.00 quatloos and contains
    2.00 scoops of orange, 8.00 scoops of blue.
Knapsack of Capacity 20.00 is worth 74.00 quatloos and contains
    2.00 scoops of orange, 8.00 scoops of blue, 6.00 scoops of
    green, 4.00 scoops of red.
Knapsack of Capacity 21.00 is worth 74.00 quatloos and contains
    2.00 scoops of orange, 8.00 scoops of blue, 6.00 scoops of
    green, 4.00 scoops of red.
```

4 Conclusion

Dynamic programming is extremely powerful, yet not very efficient. Dynamic algorithms such as Bellman-Ford for SSSP tackle variable & complex problems with ease! It is interesting to think about algorithms such as these that run our navigation systems, networking, and more!
Greedy algorithms are much simpler than they seem. All that needs to be done is to take the greediest, most locally & immediately optimal action. It is important to note though that while this did produce a globally optimal solution in this knapsack case, it does not always turn out this way, such as the cases of the 0-1 knapsack problem and traversing directed graphs!

Why did the greedy algorithm get full so quickly? *It ate all the appetizers and spared no room!*
