CMPT 435 - Fall 2024 - Dr. Labouseur

Assignment Four – Dynamic & Greedy

Ryan Munger
Ryan.Munger1@marist.edu

November 17, 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 most efficient paths between nodes. 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.

Why wouldn't the greedy algorithm move?- *Local maximums are everything to him...*

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 assingment (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 << "\n\nGRAPH" << graphCount << "\n" << endl;
    Graph currentGraph(to_string(graphCount));
    // I am taking formal lang so regex it'll be!
    // this will take care of any small whitespace errors as well
    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.SSSP();
                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.SSSP(); // don't forget the last one!!
    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 a single object. I made a new graph display function to ensure the graphs were created correctly.

```
struct linkedVertex {
    string id;
    bool processed;
    vector<tuple<linkedVertex*, int>> neighbors; // no limit to neighbors!
};

// debug helper function
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;
        }
    }
};

class Graph {
    private:
        string graphID;
        map<string, linkedVertex> linkedObjs;

        // if we want to traverse again
        void resetProcessedFlags() {
            for (auto& pair : linkedObjs) {
                pair.second.processed = false;
            }
        };
```

```
    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() {
            for (const auto& pair : this->linkedObjs) {
                printLinkedVertex(pair.second);
            }
        };
```

2.3 SSSP

I don't know yet! $O(Somethingbad, probably)$.

1. Do something!

```
        void SSSP() {
            cout << "I don't know SSSP yet!" << endl;
        };
```

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

2.4 Graph in Action

Poop

```
GRAPH1

No SSSP yet!
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
```

3 Greedy Knapsack

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 knapsack in a vector. 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;
};
```

```
void spiceHeist(const string& filename) {
    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);
            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 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 up to $O(log(n))$.

```
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;
    ostringstream scoops;
    float capacityLeft = knapsack;
    for (Spice spice : spices) {
        if (capacityLeft == 0) {
            break; // no more spice!!
        } else if(capacityLeft >= spice.quantity) {
            capacityLeft -= spice.quantity;
            knapValue += spice.total_price;
            scoops << fixed << setprecision(2) << spice.quantity <<
 " scoops of " << spice.color << ", ";
        } else if (capacityLeft < spice.quantity) {
            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

I have provided a few examples of Spices loaded, knapsacks created, and heists completed!

```
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.... 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!*

Why did the dynamic algorithm score perfectly on its make-up test? *It had all the answers saved from last time...*