

# Assignment Four – Dynamic & Greedy

---

Ryan Munger  
Ryan.Munger1@marist.edu

November 27, 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.

```
165 void createGraphs(const string& filename) {
166     int graphCount = 1;
167     cout << "Graph #" << graphCount << ":\n" << endl;
168     Graph currentGraph(to_string(graphCount));
169     regex newGraphRe(R"(new graph)");
170     regex addVertexRe(R"(add\s*vertex\s*(\S+)");
171     regex addEdgeRe(R"(add\s*edge\s*(\S+)\s*-\s*(\S+)\s*(-?\d+)");
172
173     ifstream file(filename); // input file stream
174     if (!file) {
175         cerr << "File opening failed." << endl;
176     }
177     string instruction;
178
179     while (getline(file, instruction)) {
180         // ignore any commands we don't know, empty lines, comments
181         // etc.
182         // regex will allow some slack with white space, but
183         // assuming perfect syntax by user
184         // case 1: start a new graph
185         if (regex_match(instruction, newGraphRe)) {
186             // check to see if the current graph has anything in it
187             // if not, no need to start a new one
188             if (!currentGraph.isEmpty()) {
189                 currentGraph.displayGraph();
190                 graphCount++;
191                 cout << "\n\nGraph #" << graphCount << ":\n" <<
192                 endl;
193                 currentGraph = Graph(to_string(graphCount)); //
194                 start a new graph
195             }
196             } else {
197                 smatch match; // captures subexpressions/groups
198
199                 if (regex_match(instruction, match, addVertexRe)) { //
200                     case 2: new vertex
201                     string newVertex = match[1].str();
202                     currentGraph.addVertex(newVertex);
203                 } else if (regex_match(instruction, match, addEdgeRe))
204                 { // case 3: new edge
205                     string v1 = match[1].str();
206                     string v2 = match[2].str();
207                     int weight = stoi(match[3].str());
208                     currentGraph.addEdge(v1, v2, weight);
209                 }
210             };
211         }
212     }
```

```

206     file.close();
207     currentGraph.displayGraph();
208 };

```

## 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.

```

19 struct linkedVertex {
20     string id;
21     int distance; // for SSSP
22     linkedVertex* predecessor; // for SSSP
23     vector<tuple<linkedVertex*, int>> neighbors; // no limit to
        neighbors!
24 };
25
26 void printLinkedVertex(linkedVertex v) {
27     cout << "LinkedVertex " << v.id << "; Neighbors: " << endl;
28     if (v.neighbors.empty()) {
29         cout << "\tNo Neighbors" << endl;
30     } else {
31         for (const auto& tuple : v.neighbors) {
32             cout << "\tVertex: " << get<vertexTupleIdx>(tuple)->id
33             << " Weight: " << get<weightTupleIdx>(tuple) << endl;
34         }
35 };
36
63 public:
64     Graph(string id) {
65         this->graphID = id;
66     };
67
68     void addVertex(string vertex) {
69         linkedObjs[vertex] = linkedVertex{vertex}; // store by
        value
70     };
71
72     void addEdge(string vertex1, string vertex2, int weight) {
73         this->linkedObjs[vertex1].neighbors.push_back(
        make_tuple(&linkedObjs[vertex2], weight));
74     };
75
76     bool isEmpty() {
77         return this->linkedObjs.empty();
78     };
79
80     void displayGraph() {
81         // print graph objects to ensure validity
82         for (const auto& pair : this->linkedObjs) {

```

```

83         printLinkedVertex(pair.second);
84     }
85
86     this->SSSP();
87 };

```

## 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):  $O(|V|)$ .

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  $O(|V|)$ .

```

42     void initSingleSource(linkedVertex* s) {
43         // set all vertices to distance infinite (large but not
max)
44         // no predecessors yet
45         for (auto& pair : this->linkedObjs) {
46             pair.second.distance = functionalInfinity;
47             pair.second.predecessor = nullptr;
48         }
49         // set single source
50         s->distance = 0;
51     };

```

## 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  $O(|V| * |E|)$ . As discussed in class, if we relax every edge and we do not make any changes to the predecessor or distance of any vertex, there is no reason to iterate further.

```
53 // find shortest path by recording the optimal choice
54 bool relax(linkedVertex* source, linkedVertex* destination,
55 int weight) {
56     if (destination->distance > (source->distance + weight)
57 ) {
58         destination->distance = (source->distance + weight)
59 ;
60         destination->predecessor = source;
61         return true;
62     }
63     return false;
64 };

92 // relax all edges |V| - 1 times
93 bool changesMade;
94 for (size_t i = 1; i < this->linkedObjs.size(); ++i) {
95     changesMade = false;
96     for (auto& pair : this->linkedObjs) {
97         linkedVertex* current = &pair.second;
98         for (auto& edge : current->neighbors) {
99             linkedVertex* destination = get<
100 vertexTupleIdx>(edge);
101             int weight = get<weightTupleIdx>(edge);
102             if (relax(current, destination, weight)) {
103                 changesMade = true;
104             }
105         }
106     }
107     if (!changesMade) { break; } // no need to continue
108     if no changes made
109 }
```

---

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  $O(|E|)$ .

```

109         // detect negative weight cycles
110         for (auto& pair : this->linkedObjs) {
111             linkedVertex* current = &pair.second;
112             for (auto& edge : current->neighbors) {
113                 linkedVertex* destination = get<vertexTupleIdx>
114                 >(edge);
115                 int weight = get<weightTupleIdx>(edge);
116
117                 // Check if we can further reduce the distance
118                 if (destination->distance > current->distance +
119                     weight) {
120                     return false; // negative weight cycle
121                     found
122                 }
123             }
124         }
125     }

```

## 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 not 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.

```

126     string getShortestPath(linkedVertex* destination) {
127         // follow predecessors (reverse order)
128         stack<string> pathStack;
129         linkedVertex* predecessor = destination;
130         while (predecessor != nullptr) {
131             pathStack.push(predecessor->id);
132             predecessor = predecessor->predecessor;
133         }
134
135         // put them in forward order for display
136         ostream pathstr;
137         while (!pathStack.empty()) {
138             // Check if something is already in the stream for
139             ->
140             if (pathstr.tellp() > 0) {
141                 pathstr << "->";
142             }
143             pathstr << pathStack.top();
144             pathStack.pop();
145         }

```

```

145         return pathstr.str();
146     };
147
148     void SSSP() {
149         linkedVertex* startVertex = &this->linkedObjs.begin()->
second;
150         // set distances, predecessors, etc
151         bool success = bellmanFord(startVertex);
152         if (!success) {
153             cout << "Negative weight cycle detected. Results
may be unreliable!" << endl;
154         }
155
156         cout << "SSSP: " << endl;
157         for (auto& pair : this->linkedObjs) {
158             linkedVertex* current = &pair.second;
159             cout << startVertex->id << "->" << current->id << "
cost is " << setw(2) <<
160                 current->distance << "; shortest path is " <<
getShortestPath(current) << endl;
161         }
162     };

```

---

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

---

## 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.

```

3 Graph #1:
4
5 LinkedVertex 1; Neighbors:
6   Vertex: 2 Weight: 6
7   Vertex: 4 Weight: 7
8 LinkedVertex 2; Neighbors:
9   Vertex: 3 Weight: 5
10  Vertex: 4 Weight: 8
11  Vertex: 5 Weight: -4
12 LinkedVertex 3; Neighbors:
13  Vertex: 2 Weight: -2
14 LinkedVertex 4; Neighbors:
15  Vertex: 3 Weight: -3
16  Vertex: 5 Weight: 9
17 LinkedVertex 5; Neighbors:
18  Vertex: 3 Weight: 7
19  Vertex: 1 Weight: 2
20
21 SSSP:
22 1->1 cost is 0; shortest path is 1
23 1->2 cost is 2; shortest path is 1->4->3->2
24 1->3 cost is 4; shortest path is 1->4->3
25 1->4 cost is 7; shortest path is 1->4
26 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.

```
210 struct Spice {
211     string color;
212     float total_price;
213     float quantity;
214     float unit_price;
215 };

272 void spiceHeist(const string& filename) {
273     cout << "\n\nLoading in Spices and Knapsacks!" << endl;
274     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*;)" );
275     regex knapsackRe(R"(knapsack\s*capacity\s*=\s*(\d*.\?\d*)\s*;)" );
276
277     // store spices and knapsacks
278     vector<Spice> spiceInventory;
279     vector<float> knapsacks;
280
281     ifstream file(filename); // input file stream
282     if (!file) {
283         cerr << "File opening failed." << endl;
284     }
285     string instruction;
286
287     while (getline(file, instruction)) {
288         smatch match; // captures subexpressions/groups
289         // case 1: adding a spice
290         if (regex_match(instruction, match, spiceRe)) {
291             string color = match[1].str();
292             float total_price = stof(match[2].str());
293             float quantity = stof(match[3].str());
294             float unit_price = total_price / quantity;
295
296             Spice newSpice = Spice{color, total_price, quantity,
unit_price};
297             printSpice(newSpice);
298             spiceInventory.push_back(newSpice);
299         } else if (regex_match(instruction, match, knapsackRe)) {
300             // case 2: knapsack
301             float newKnapsackCapacity = stof(match[1].str());
knapsacks.push_back(newKnapsackCapacity);
```



```

302         cout << "New Knapsack: " << newKnapsackCapacity << endl
303     };
304 };
305 file.close();
306 // sort our spices based on unit price
307 spiceSort(spiceInventory);
308 // maximize take for each knapsack!
309 cout << "\nMaximizing Take:" << endl;
310 for (float knapsack : knapsacks) {
311     maximizeTake(knapsack, spiceInventory);
312 }
313 };

```

## 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  $O(\log(n))$ .

```

225 // Insertion sort to get descending order based on unit price
226 void spiceSort(vector<Spice>& arr) {
227     int n = arr.size();
228     for (int i = 1; i < n; i++) {
229         int insertIdx = i;
230         Spice currentCheck = arr[i];
231         for (int j = i-1; j >= 0; j--) {
232             if (arr[j].unit_price < currentCheck.unit_price) {
233                 arr[j+1] = arr[j];
234                 insertIdx = j;
235             } else {
236                 break;
237             }
238         }
239         arr[insertIdx] = currentCheck;
240     }
241 };

```

### 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  $O(n \log(n))$  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.

```
243 void maximizeTake(float knapsack, vector<Spice> spices) {
244     float knapValue = 0;
245     if (knapsack == 0) {
246         cout << "Knapsack of Capacity " << fixed << setprecision(2)
247             << knapsack << " is worth " <<
248             fixed << setprecision(2) << knapValue << " quatloos and
249             contains no scoops." << endl;
250         return;
251     }
252     ostringstream scoops;
253     float capacityLeft = knapsack;
254     for (Spice spice : spices) {
255         if (capacityLeft == 0) {
256             break; // no more spice!!
257         } else if (capacityLeft >= spice.quantity) { // take all the
258             spice
259             capacityLeft -= spice.quantity;
260             knapValue += spice.total_price;
261             scoops << fixed << setprecision(2) << spice.quantity <<
262             " scoops of " << spice.color << ", ";
263         } else if (capacityLeft < spice.quantity) { // take what we
264             can fit
265             knapValue += capacityLeft * spice.unit_price;
266             scoops << fixed << setprecision(2) << capacityLeft << "
267             scoops of " << spice.color << ", ";
268             capacityLeft = 0;
269         }
270     }
271     string scoopString = scoops.str();
272     // replace last comma with period
```

```

267     scoopString.pop_back();
268     scoopString.back() = '.';
269
270     // report the value of our knapsack and scoops taken
271     cout << "Knapsack of Capacity " << fixed << setprecision(2) <<
knapsack << " is worth " <<
272         fixed << setprecision(2) << knapValue << " quatloos and
contains " << scoopString << endl;
273 };

```

### 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.

```

123 Loading in Spices and Knapsacks!
124 Spice:
125   Color: red
126   Total Price: 4
127   Quantity: 4
128   Unit Price: 1
129 Spice:
130   Color: green
131   Total Price: 12
132   Quantity: 6
133   Unit Price: 2
134 Spice:
135   Color: blue
136   Total Price: 40
137   Quantity: 8
138   Unit Price: 5
139 Spice:
140   Color: orange
141   Total Price: 18
142   Quantity: 2
143   Unit Price: 9
144 New Knapsack: 1
145 New Knapsack: 6
146 New Knapsack: 10
147 New Knapsack: 20
148 New Knapsack: 21
149
150 Maximizing Take:
151 Knapsack of Capacity 1.00 is worth 9.00 quatloos and contains 1.00
scoops of orange.
152 Knapsack of Capacity 6.00 is worth 38.00 quatloos and contains 2.00
scoops of orange, 4.00 scoops of blue.
153 Knapsack of Capacity 10.00 is worth 58.00 quatloos and contains
2.00 scoops of orange, 8.00 scoops of blue.
154 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.
155 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!*

---