CMPT 435 - Fall 2023 - Dr. Labouseur

Assignment Four – LaTeX DP and Greedy

Patrick Tyler
Patrick.Tyler1@marist.edu

December 8, 2023

1 Dynamic Programming

Dynamic programming (DP) is the computer science technique of using results from a previous calculation is the rest of the program (breaking problems into smaller pieces). This is especially useful when you have expensive pure functions (a function which returns the same value for the same input and has no side effects). A popular problem which benefits heavily from a DP approach is calculating the fibonacci sequence recursively. There is a lot of identical calls when calculating the fibonacci sequence. So, if the results are cached a lot of computation can be saved as demonstrated in figure 1.

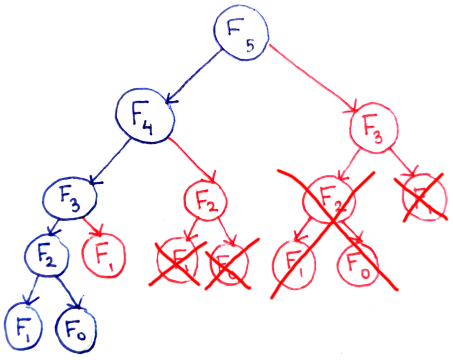

Figure 1: Fibonacci Sequence from Avik Das

1.1 Bellman Ford Algorithm

Another algorithm which takes advantage of DP is the Bellman Ford algorithm. This algorithm finds the shortest distances from a source vertex to the other reachable vertices in a weighted graph. The steps of this algorithm are as follows:

1. Initialize storage of current distance values and predecessors. The starting distance from the starting vertex should always compare as greater than a finite distance so infinity is used. The distance from the starting vertex to itself is 0.

```
244     unordered_map<Vertex*, int> distanceFromOrigin;
245     unordered_map<Vertex*, Vertex*> predecessors;
246     for(auto vertex : *this->vertices) {
247         distanceFromOrigin.insert({vertex, (vertex == origin)
    ? 0 : INT_MAX});
248         predecessors.insert({vertex, nullptr});
249     }
```

2. For each edge check where u is the **from** vertex, v is the **to**, and its distance is defined as the stored shortest path from the starting point. Check if u's distance plus the edge weight is less than v's current weight. If that check is true, then update the distance with the shortest value and v's predecessor to u. The checking and updating process is generally referred to as "relaxing" the edge.

```
254         for(auto& sourceVertex : *this->vertices ){
255             for(auto& toEdge : sourceVertex->neighbors) {
256                 auto& destinationVertex = toEdge.first;
257                 auto weight = toEdge.second;
258                 // relax
259                 if(distanceFromOrigin[sourceVertex] == INT_MAX
    ) {
260                     continue;
261                 }
262                 int weightFromSource = distanceFromOrigin[
    sourceVertex] + weight;
263                 if( weightFromSource < distanceFromOrigin[
    destinationVertex]){
264                     distanceFromOrigin[destinationVertex] =
    weightFromSource;
265                     predecessors[destinationVertex] =
    sourceVertex;
266                 }
267             }
268         }
```

3. Repeat step 2 until it has been completed one less than the count of vertices. This ensures that the algorithm has enough iterations to propagate the distance values at most the count of vertices minus 1 out. This should be intuitive because any given vertex can be at most vertices count minus 1 edges away from the starting vertex if it is reachable at all.

4. Check for any negative edge cycles. If one of these exists, then the minimum path is indefinitely small/negative infinity. This step is essentially the same as 2, but if there is any spot where the condition is true then there are negative cycle(s).

```
272     for(auto& sourceVertex : *this->vertices ){
273         for(auto& toEdge : sourceVertex->neighbors) {
274             auto& destinationVertex = toEdge.first;
275             auto weight = toEdge.second;
276             // relax
277             if(distanceFromOrigin[sourceVertex] == INT_MAX) {
278                 continue;
279             }
280             int weightFromSource = distanceFromOrigin[
    sourceVertex] + weight;
281             if( weightFromSource < distanceFromOrigin[
    destinationVertex]){
282                 // there is a negative weight cycle
283                 cout << "This graph has a negative weight
    cycle" << endl;
284                 return;
285             }
286         }
287     }
```

The following time complexities are based off these variables: $|V|$ is the number/ cardinality of vertices and $|E|$ is the number/ cardinality of edges. It is worth noting that $0 <= |E| <= |V|^2$ so substitutions can be made.

1. Time complexity $O(|V|)$ because it does some work for each vertex.
2. Time complexity $O(|E|)$ because it does some work for each edge.
3. Time complexity $O(|V| * |E|)$ because it does step 2 for each vertex.
4. Time complexity $O(|E|)$ because it does some work for each edge.

Overall this algorithm takes on the time complexity $O(|V|*|E|)$. It is important to note that due to this implementation of edge storage step 2 must at least loop through all vertices even if there is no edge on those vertices.

Student: *Hey Professor, why did I get a 0 on my DP homework?*
Professor: *It was copied. I literally found the exact code online.*
Student: *Yup, why do the work when we can already lookup the answer that's DP 101 right?*
Professor: *You're a moron.*

---

2 Greedy Algorithms

A greedy algorithm takes locally optimal solutions to approximate a globally optimal solution. Giving optimal change is a great example of how a problem can benefit from a greedy approach even if it needs a DP backtracking approach in certain cases. With US currencies you can always find the optimal change (least amount of bills and or coins) with the diagramed greedy method: taking the highest denomination that fits adding that to change and repeating the process with what is leftover as seen in figure 2. If the problem had any set of valid denominations this would not be possible with a greedy approach. For instance, if you had this set of currencies {20, 9, 1} and need to give 28 dollars of change, the solution from the greedy approach would be {20, 1 x 7}. However, the optimal solution is {9 x 3, 1}.

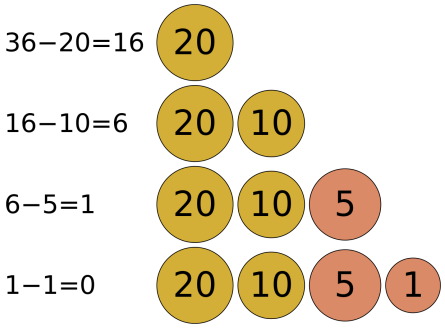

Figure 2: Change diagram and example from wikipedia

2.1 Spice Heist

The intergalactic Spice Heist is a flavor of the knapsack problem. Which much like the currency problem has a subproblem, all or none nothing knapsack, where a greedy solution does not produce a globally optimal answer. In this heist, however, it is permissable to take any fraction of spices. This version allows for a the greedy approach much like giving change with denomination that are multiples of each other: simply just take all of the most cost-effective spices first until the sack is at capacity or there are no more spices to take.

The time complexity depends on the sorting implemented. The implementation in the index uses a binary search tree (without balancing) to sort the spices. As each spice is inserted it takes $O(n)$ to correctly insert it in the BST (think of the stick BST from previous the assignment) for its given where n is the amount of spices currently in the BST. However, it will likely only take a little over $log_2 n$ operations if the spices are randomly arranged in terms of the unit cost. This operation happens n times so therefore adding all spices is a $O(n^2)$

operation, but more likely around $nlog_2n$ operations will be done. Getting the spices sorted from the tree is done through an in-order traversal and the elements are appended to a vector for convenience. Retrieving this vector in the doHeist function takes $O(n)$ time if the vector has not been retrieved since last insertion or is constant time if the sorted vector is already stored. Then, looping through the spices to and doing checks to add the most amount is $O(n)$. This makes the overall process of adding spices and generating the best knapsack for a given capacity $O(n^2)$.

```
563     void doHeist(int knapsackCapacity) {
564         vector<Spice*>* orderedSpices = this->spices->getInOrder();
565         int ogKnapsackCapacity = knapsackCapacity;
566         stringstream scoops;
567         scoops << " quatloos and contains";
568         int quatloosSum = 0;
569         for(auto spiceRef : (*orderedSpices)) {
570             auto spice = (*spiceRef);
571             if (knapsackCapacity == 0) { break; }
572 
573             int quantityAdded = min(spice.quantity,
    knapsackCapacity);
574             knapsackCapacity -= quantityAdded;
575             quatloosSum += spice.unitCost * quantityAdded;
576             if(quantityAdded == 1 ) {
577                 scoops << " " << quantityAdded << " scoop of " <<
    spice.name << ",";
578             } else {
579                 scoops << " " << quantityAdded << " scoops of " <<
    spice.name << ",";
580             }
581         }
582         // replace last comma with a period
583         auto scoopsMessage = scoops.str();
584         scoopsMessage.back() = '.';
585 
586         cout << "Knapsack of capacity " << ogKnapsackCapacity << "
    is worth " << quatloosSum << scoopsMessage << endl;
587 
588     }
```

Guy:
Why do we call politicians greedy? *Because they make locally optimal solutions to extend their power with the global consequences of screwing future generations.*
Interviewer:
Sir, the question was: *What is a greedy algorithm?*

---

3 Miscellaneous

As always the source code can be found in the index. The chosen way to do the file reading and parsing was incredibly stupid and probably not even more efficient than easier and more readable approaches. You can look at the commit messages to get a better understanding of my frustrations. However, the parsing should not take all the credit for my frustrations, because I thought the raw binary output (mounds of red text) was from incorrect parsing when it was really from bad memory management.

Here's a C++ exercise for the reader. Which copy has a likely bug in it?
Source code and latex has the answer but the explanation can be found below.

```
        // other function stuff
        Spice* spice = new Spice();
        spice->name = name;
        spice->price = totalPrice;
        spice->quantity = quantity;
        spice->unitCost = totalPrice / quantity;
        this->spices->insert(spice);
        // end of function
```

```
        // other function stuff
        Spice spice = (*new Spice());
        spice.name = name;
        spice.price = totalPrice;
        spice.quantity = quantity;
        spice.unitCost = totalPrice / quantity;
        this->spices.insert(&spice);
        // end of function
```

The problem is that C++ automatically garbage collects the Spice object that is dereferenced in the function and therefore my BST was storing pointers to collected objects. When I was debugging this I printed the attributes as they were being adding which obviously still worked and left me very confused, because the only problem was mounds of raw binary bad encoding being output when I was printing it later.