CS 577 # Assignment 4 – More Greedy Spring 2023

Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: Carson Miller Wisc id: 9081473028

## More Greedy Algorithms

1. *Kleinberg, Jon. Algorithm Design (p. 189, q. 3).*

   You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit $W$ on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package $i$ has a weight $w_i$. The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

   Prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed. Hint: Use the stay ahead method.

**Solution:**

Assume that there exists some optimal algorithm *X* that uses fewer trucks than our greedy algorithm *Y* to ship the same number of boxes *n*.

Consider the first *k* boxes that arrive at the New York station, such that *k* is the largest integer such that the sum of their weights is less than or equal to *W*. Both *X* and *Y* will load these *k* boxes onto the first truck and send it to Boston. In this scenario, neither algorithm leaves any boxes behind and both use 1 truck.

Next, consider some next box *i* > k that arrives at the New York station. Because *X* is assumed to use fewer trucks than *Y*, there must've been some way for *X* to load *i* onto truck 1 without exceeding *W* and without violating the company policy to ship boxes in the order that they arrive.

Algorithm *X* could've loaded box *i* onto truck 1 by removing the heaviest box from the truck and replacing it with box *i*. This would violate the policy of the order arrival of boxes, leading to a contradiction of the problem statement. Because of this policy, the most optimal algorithm for minimizing the number of trucks requires loading boxes as the arrive and sending the truck away once adding the weight $w_i$ of the next box *i* to the current weight is greater than *W*.

2. *Kleinberg, Jon. Algorithm Design (p. 192, q. 8).* Suppose you are given a connected graph $G$ with edge costs that are all distinct. Prove that $G$ has a unique minimum spanning tree.

**Solution:**

Suppose there are two distinct MSTs of G, T1 and T2. Let e be the smallest cost edge that is in T1 but not in T2. Because e is not in T2, adding it to T2 would create a cycle. If all of the other edges in this cycle were in T1, then T1 would contain a cycle, which it cannot because it is an MST. Thus, this cycle must contain an edge f that is not found in T1. Because all edge costs are distinct and e is the smallest cost edge of all edges in T1 but not T2, the cost of e must be less than the cost of f. If we were to replace f with e in T2, the result would be a MST with a smaller cost, which is a contradiction. Thus, there may not be two distinct MSTs of G and there can only be one unique MST of G.

3. *Kleinberg, Jon. Algorithm Design (p. 193, q. 10).* Let $G = (V, E)$ be an (undirected) graph with costs $c_e \geq 0$ on the edges $e \in E$. Assume you are given a minimum-cost spanning tree $T$ in $G$. Now assume that a new edge is added to $G$, connecting two nodes $v, w \in V$ with cost $c$.

   (a) Give an efficient ($O(|E|)$) algorithm to test if $T$ remains the minimum-cost spanning tree with the new edge added to $G$ (but not to the tree $T$). Please note any assumptions you make about what data structure is used to represent the tree $T$ and the graph $G$, and prove that its runtime is $O(|E|)$.

**Solution:**

Assuming T and G are represented by arrays of edges ({node A, node B, cost C}), I will design an algorithm that tests if T remains the minimum-cost spanning tree with the new edge added to G.

Perform DFS on T, starting from v, to find the path from v to w, while tracking the largest cost on that path.

If $c_e$ of the new edge is less than the largest cost on that path, T does not remain the minimum-cost spanning tree because the new edge can replace that largest cost edge to make the minimum cost smaller. Otherwise, T remains the minimum-cost spanning tree.

Because this algorithm only adds a slight, constant-time variation to DFS, the runtime is O(|V| + |E|), which is not as great as O(|E|) but necessary to find the path from v to w. I don't see how it can be done without finding that path.

   (b) Suppose $T$ is no longer the minimum-cost spanning tree. Give a linear-time algorithm (time $O(|E|)$) to update the tree $T$ to the new minimum-cost spanning tree. Prove that its runtime is $O(|E|)$.

**Solution:**

First, we would again perform DFS on T, starting from v, to find the path from v to w, while tracking the largest cost on that path. Once the path is complete, remove the edge on that path with the largest cost. The last step would be to add {v, w, $c_{vw}$} to T.

Because this algorithm only adds a slight, constant-time variation to DFS, the runtime is O(|V| + |E|), which is not as great as O(|E|) but necessary to find the path from v to w. I don't see how it can be done without finding that path.

4. In class, we saw that an optimal greedy strategy for the paging problem was to reject the page the furthest in the future (FF). The paging problem is a classic online problem, meaning that algorithms do not have access to future requests. Consider the following online eviction strategies for the paging problem, and provide counter-examples that show that they are not optimal offline strategies.[1]

   (a) FWF is a strategy that, on a page fault, if the cache is full, it evicts all the pages.

**Solution:**

I am trying to prove that FWF results in more page faults than optimal when used offline. My counterexample uses a cache of size 2. I will show the status of the cache after each request.

**Requests: {0, 1, 2, 1}**

| *FWF* | | *FF (more optimal)* | |
|---|---|---|---|
| 0 | | 0 | |
| Page faults = 1 | | Page faults = 1 | |
| 0 | 1 | 0 | 1 |
| Page faults = 2 | | Page faults = 2 | |
| 2 | | 2 | 1 |
| Page faults = 3 | | Page faults = 3 | |
| 2 | 1 | 2 | 1 |
| Page faults = 4 | | Page faults = 3 | |

   (b) LRU is a strategy that, if the cache is full, evicts the least recently used page when there is a page fault.

**Solution:**

I am trying to prove that FWF results in more page faults than optimal when used offline. My counterexample uses a cache of size 2. I will show the status of the cache after each request.

**Requests: {0, 1, 2, 0}**

| *LRU* | | *FF (more optimal)* | |
|---|---|---|---|
| 0 | | 0 | |
| Page faults = 1 | | Page faults = 1 | |
| 0 | 1 | 0 | 1 |
| Page faults = 2 | | Page faults = 2 | |
| 2 | 1 | 0 | 2 |
| Page faults = 3 | | Page faults = 3 | |
| 2 | 0 | 0 | 2 |
| Page faults = 4 | | Page faults = 3 | |

[1] An interesting note is that both of these strategies are $k$-competitive, meaning that they are equivalent under the standard theoretical measure of online algorithms. However, FWF really makes no sense in practice, whereas LRU is used in practice.

## 5. Coding problem

For this question you will implement Furthest in the future paging in either C, C++, C#, Java, or Python.

The input will start with an positive integer, giving the number of instances that follow. For each instance, the first line will be a positive integer, giving the number of pages in the cache. The second line of the instance will be a positive integer giving the number of page requests. The third and final line of each instance will be space delimited positive integers which will be the request sequence.

A sample input is the following:

```
3
2
7
1 2 3 2 3 1 2
4
12
12 3 33 14 12 20 12 3 14 33 12 20
3
20
1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

The sample input has three instances. The first has a cache which holds 2 pages. It then has a request sequence of 7 pages. The second has a cache which holds 4 pages and a request sequence of 12 pages. The third has a cache which holds 3 pages and a request sequence of 15 pages.

For each instance, your program should output the number of page faults achieved by furthest in the future paging assuming the cache is initially empty at the start of processing the page request sequence. One output should be given per line. The correct output for the sample input is

```
4
6
12
```