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: Kayley Seow Wisc id: kseow

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:

Givens:

- Truck with maximum weight limit W
- 2 truck at least available
- with a package i. we have weight W_i
- Company policy wants shipped in order of arrival, so first in last out

P(n): algorithm uses n trucks

Base case P(1): one truck, total weight Σw_i of packages not exceed W

Inductive Reasoning: Assume P(k), prove P(k+1). let P^* P be optimal solution, packages ordered differently.

Company policy requires packages in certain order, a package i that comes after j cannot be switched. Only difference between optimal is fitting packages on the truck or putting it on another truck. Our solution P fits all of the packages on the truck, so it is the most optimal.

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:

We assume there are two unique minimum spanning trees in G, m and m'. There are some edges that the MST's don't share. let e' be an edge belonging to the MST m'. Adding e' to m would create a cycle between e' and another edge e in m. Since all edges are distinct, one of the edges between e and e' will have lower cost. But, this would mean that one of the MST's is not using the lowest costing edge, by it being an MST. By contradiction, we would not have m and m'.

- maximum weight of ~~MST~~ cycle not in MST

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:

Let there be an MST from W to V nodes, and edge e added to graph between w and v

Conduct a BFS on G, from node W to V

```
    int max=edge e weight
    if current edge > max
        max = current edge
end BFS
```

if max is not the edge e we added, then the MST changes. if not, the MST stays the same.

$O(|E|+|V|) \in O(E) = O(2E+1) = O(E)$

$|E| = |V| - 1$

$|V| = |E| + 1$

   (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:

Conduct BFS on G, from node W to W

Iterate through T, and keep track of max and replace it with e.

→ because the new edge creates a cycle, we can remove max weight

$O(|E|+|V|) \Rightarrow O(E) = O(2E+1) = O(E)$

$|E| = |V| - 1$

$|V| = |E| + 1$

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:

$\mathcal{U} = \{a, b, c\}$

$k = 2$

$6 = \langle abcbcab \rangle$

$\boxed{a \mid b}$ $2+2+2+1$

c b

c a

b

With FWF, there are 7 page faults, as the cache is cleared every two elements, and if not, there are page faults, from adding in the elements. In comparison, FF only has 5 page faults so it is more efficient than FWF.

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

Solution:

$M = \{a, b, c\}$

$k = 2$

$6 = \langle a, b, c, b, c, a, b \rangle$

$\boxed{a \mid b}$ $2+1+1+1+1$ $\boxed{a \mid b}$

c a ↑ c

b orig

With LRU, there are 6 page faults, as the cache is cleared with recently. We have one more page fault than the FF, which makes it more optimal.

[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- instances
2- pages
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
```