CS 577 # Assignment 3: Greedy Algorithms Fall 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: Ethan Yan Wisc id: 9084649137

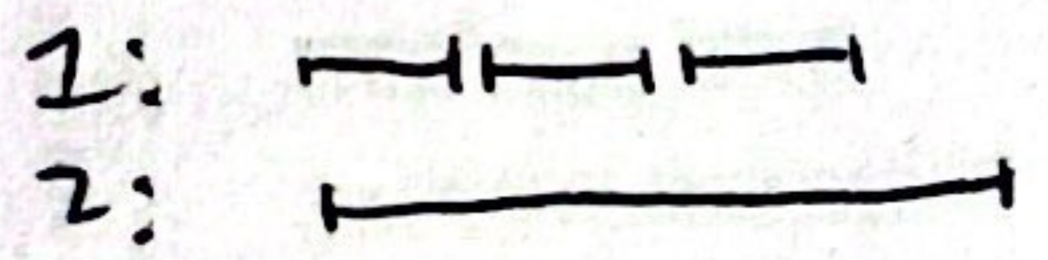
Greedy Algorithms

1. In one or two sentences, describe what a greedy algorithm is. Your definition should be informal, something you could share with a non computer scientist.

A greedy algorithm is an algorithm that repeatedly picks the best option at the given moment in time without regard for future moves, ie it picks the optimal pick ~~at~~ considering only that moment's circumstance.

2. There are many different problems all described as "scheduling" problems. In the following questions, pay attention to the details of the problem setup, as they will change each time!

   (a) Let each job have a start time, an end time, and a value. We want to schedule as much value of non-conflicting jobs as possible. Use a counterexample to show that Earliest Finish First (the greedy algorithm we used for jobs with all equal value) does NOT work in this case.



Algorithm will always pick from 1, despite 2 having greater value as it spans a longer period of time

   (b) *Kleinberg, Jon. Algorithm Design (p. 191, q. 7)* Now let each job consist of two durations. A job i must be preprocessed for p_i time on a supercomputer, and then finished for f_i time on a standard PC. There are enough PCs available to run all jobs at the same time, but there is only one supercomputer (which can only run a single job at a time). The completion time of a schedule is defined as the earliest time when all jobs are done running on both the supercomputer and the PCs. Give a polynomial time algorithm that finds a schedule with the earliest completion time possible.

In order to maximise parallel running ability for the f_i s, we want to process jobs with greatest to smallest f_is. In order to prioritize jobs with greater f_is, we need to sort all jobs, thus such greedy algorithms would use a nlog n sorting algorithm to sort all jobs by descending f_i. Suppose input is a list of n length of jobs (p_i, f_i). Algorithm:

~~for i in range (1,n):~~

```
1. Use sorting algo (eg quicksort), to sort jobs by $f_i$ in descending order.
2. for i in range (1,n):
3.     feed job to supercomputer, feed to PC once done.
4. Stop when all jobs comple.
```

Runtime: O(nlog n).

(c) Prove the correctness and efficiency of your algorithm from part (b).

Prove correctness:

consider ~~total~~ time of completion for each job, c_i.

$C_i = \left(\sum_{j=1}^{i} p_j \; [illegible]\right) + f_i$ since the preprocessing time of each job depends on prior job's preprocessing.

Consider the base case: ~~[illegible]~~ In the case of only 1 job, it is trivial.

If we have 2 jobs, (p_1, f_1) and (p_2, f_2). If $c_2 > c_1$, and $f_2 > f_1$, we can improve on it using greedy solution by putting job 2 first.

<u>Before</u>

$c_2 = p_1 + p_2 + f_2$ & $c_1 = p_1 + f_1$

<u>After</u>

$c_2 = p_2 + f_2$ $c_1 = p_1 + p_2 + f_1$

Before: completion time would be $p_1 + p_2 + f_2$

After: completion time would be $p_1 + p_2 + f_1$

} As $f_2 > f_1$ greedy soln does lower completion time proving base case.

<u>Inductive step</u>

In an inductive manner, we can expand this to a set of any size by ordering jobs by arranging them pairwise, performing swaps until none can be made. Note that changing order of a pair of jobs does not affect completion of latter job or earlier jobs that might cause overall increase in completion time. Thus, this is repeated till there are no more swaps.

Define example in base case as an inversion.

Note Greedy Algo produces schedule with no inversion.

From inductive step, we know there is solution with no inversions.

All schedules with no inversions have the same completion time.

$\Rightarrow$ only vary in jobs with same f_i but that does not change overall completion time

To order n jobs uses same algorithm used to order $n-1$ jobs. Since base case shown to hold, Induction holds.

Prove efficiency:

Uses effcient sort algorithm (eg quick sort) $\Rightarrow O(n \log n)$.

Feed each of n jobs to supercomputer then to PC $\Rightarrow O(n) + O(n) = O(n)$

Overall algorithm would then be $O(n) + O(n \log n) = O(n \log n)$ which is polynomial

3. *Kleinberg, Jon. Algorithm Design (p. 190, q. 5)*
   (a) Consider a long straight road with houses scattered along it. We want to place cell phone towers along the road so that every house is within four miles of at least one tower. Give an efficient algorithm that achieves this goal using the minimum possible number of towers.

Have an array that sorts houses from west to east. Let this array be A.
While A is not empty:
Place ~~the first~~ a cell tower 4 miles to right (east) of first house in array.
Remove all houses covered by this cell tower

ie repeat placing cell tower, to 4 miles ~~a~~ east of the next house that is not covered until no houses left.

   (b) Prove the correctness of your algorithm.

Let $a_1, a_2, \ldots, a_i$ denote cell towers in my greedy algorithm from ~~left~~ west to east, indicating their distance from the first house.

So $a_1 = 4$ miles, $a_2 \geq 12$ miles, $a_3 \geq 20$ miles, ...

Let $b_1, b_2, \ldots, b_j$ denote cell towers in an optimal algorithm. from west to east, denoting their distance from the first house aswell.

we know that $b_1 \leq 4$ miles $\Rightarrow b_1 \leq a_1$

$b_2 \leq 12$ miles $\Rightarrow b_2 \leq a_2$ (Note b_2 cannot be larger than 12 or there would be a "dead zone right after the 8 mile mark where a house would not be covered by b_1 or b_2).

$b_3 \leq a_3$

$\vdots$

$b_j \leq a_i$

Since $b_1 \leq a_1$, $b_2 \leq a_2$, ... we know that $i \leq j$

ith tower in greedy algorithm is a further distance east or the same as the ith tower in the optimal algorithm. Therefore, number of towers in the optimal algorithm $\leq i$. Thus showing that the greedy algorithm in (a) is efficient, thus proving correctness.

4. *Kleinberg, Jon. Algorithm Design (p. 197, q. 18)* Your friends are planning to drive north from Madison to the town of Superior, Wisconsin over winter break. They have drawn a directed graph with nodes representing potential stops and edges representing the roads between them.

   They have also found a weather forecasting site that can accurately predict how long it will take to traverse one of the edges on their graph, given the starting time t. This is important because some of the roads on their graph are affected strongly by the seasons and by extreme weather. It's guaranteed that it never takes negative time to traverse an edge, and that you can never arrive earlier by starting later.

   (a) Design an algorithm your friends can use to plot the quickest route. You may assume that they start at time $t = 0$, and that the predictions made by the weather forecasting site are accurate.

Can use Dijkstras Algorithm. With graph $G(V, E)$ with ~~each~~ V being the set of vertizes & E being the edge list.

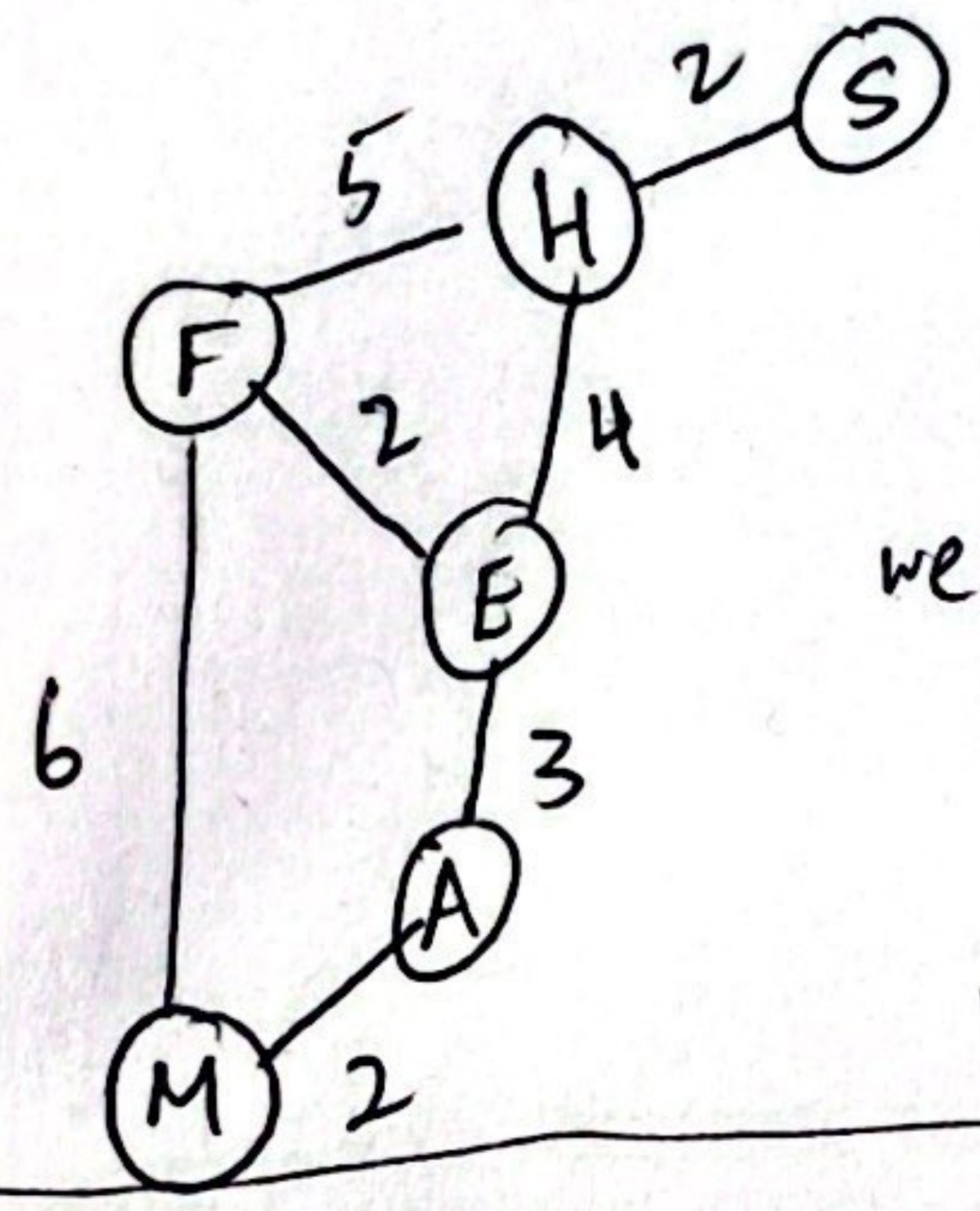
```
function Dijkstra(graph, sourceNode):
    for each vertex v in graph.vertizes:
        dist[v] ← infinity
        prev[v] ← null
        add v to notVisitedArray
    dist[source] ← 0.

    while notVisitedArray is not empty:
        u ← vertex in notVisitedArray with min(dist[u])
        remove u from notVisitedArray.

        for each neighbour w of u in notVisitedArray:
            tempDist ← dist[u] + graph.edges(w, u)
            if tempDist < dist[v]:
node.name       dist[v] ← tempDist
~~destNode.name~~   prev[v] ← u
~~destNode~~ = Superior    "Madison"
while node.name is not ~~destName~~: path.append(node)
                                   node = node.prev
                    return path.reverse.
```

(b) Demonstrate how your algorithm works using a small example with 6 nodes. Your demonstration should include any data structures you maintain during the execution of your algorithm and any queries you make to the weather forecasting site. For example, if your algorithm maintains a "current path" that grows from (M)adison to (S)uperior, you might show something like the following table:

Path	Total time
M	0
M,A	2
M,A,E	5
M,A,E,F	6
M,A,E	5
M,A,E,H	10
M,A,E,H,S	13



```
for each vertex v in graph.vertices:
    dist [v] ← infinity
    prev [v] ← null
    add v to not VisitedArray.
dist[source] ← 0
```

we now have dist = { M : 0, A : inf, E : inf, F : inf, H : inf, S : inf } prev = { M : null, A : null, E : ', F : ', H : ', S: null }

notVisitedArray = [M, A, E, F, H, S]

```
while notVisitedArray is not empty:
    u ← vertex in notVisitedArray with min(dist[v])
    remove u from notVisitedArray
    for each neighbour w of u in notVisitedArray:
        tempDist ← dist[u] + graph.edges(w,u)
        if tempDist < dist[v]:
            dist[v] ← tempDist
            prev[v] ← u
```

Run 1:
u = M
dist = { M : 0, A : 2, E : inf, F : 6, H : inf, S : inf } prev = { M : null, A : M, E : null, F : M, H : null, S : null }
notVisitedArray = [A, E, F, H, S].

Run 2: u = A
dist = { M : 0, A : 2, E : 5, F : 6, H : inf, S : inf } prev = { M : null, A : M, E : A, F : M, H : null, S : null }
notVisitedArray = [E, F, H, S]

Run 3: u = E
dist = { M : 0, A : 2, E : 5, F : 6, H : 9, S : inf } prev = { M : null, A : M, E : A, F : M, H : E, S : null }
notVisitedArray = [F, H, S].

Run 4: u = F, dist, prev same as Run 3, notVisitedArray = [H, S]

Run 5: u = H
dist = { M : 0, A : 2, E : 5, F : 6, H : 9, S : 11 } prev = { M : null, A : M, E : A, F : M, H : E, S : H } notVisitedArray = [S]

Run 6: u = S, dist, prev same as run 5, notVisitedArray = []

```
~~while~~ node = S
    while node.prev is not null:
        path.append(node)
        node ← node.prev
```

path = [S, H, E, A, M]

return path.reverse

so [M, A, E, H, S] is returned.