GREEDY ALGORITHM

REVIEW

Review

1. (T/F) By using a binomial heap we can sort data of size n in $O(n)$ time.

Another proof is that in lecture 1 we proved that the lower bound for sorting is $\Omega(n \log n)$

① First we will need to turn the raw data into a heap
② Then run deleteMin to sort ∴ we cannot sort in $o(n)$ time.

2. Prove that it is impossible construct a min-heap (not necessarily binary) in a comparison-based model with *both* the following properties:

a. deleteMin() runs in $O(1)$,
b. buildHeap() runs in $O(n)$,
where n is the input size.

↳ In these algorithms the sorting order can be determined only based on comparisons between input elements

1. `deleteMin()`: In a comparison-based model, deleting the minimum element from a min-heap requires reorganizing the heap to maintain its properties. To achieve O(1) time complexity for `deleteMin()`, you would need to somehow know the minimum element in the heap without examining all the elements, which is not possible in a comparison-based model. In a comparison-based model, you must compare elements to determine their order, and finding the minimum element will always take at least O(n) time because you have to examine each element at least once.

2. `buildHeap()`: Building a heap with O(n) time complexity means that you can construct a heap from an arbitrary array of n elements without making more than O(n) comparisons. Again, this is not possible in a comparison-based model because each comparison involves examining two elements. To build a heap, you need to compare and potentially swap elements to satisfy the heap property, which requires Ω(n log n) comparisons in the worst case.

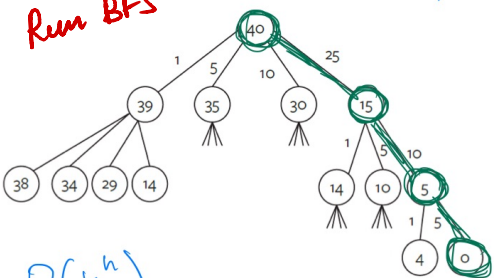
Therefore, it is impossible to have both `deleteMin()` in O(1) and `buildHeap()` in O(n) in a comparison-based model. These two properties are conflicting in such a model.

The Money Changing Problem

We are to make a change of $0.40 using US currency and assuming that there is an unlimited supply of coins. The goal is to compute the minimum number of coins.

penny, nickel, dime, quarter

Run BFS



h

greedy approach:
1. choose the largest coin

Runtime: O(h)

$O(4^h)$.

Since running a DFS to find the leaf nodes for the solution, the runtime will be exponential. Therefore, greedy approach

TRADITIONAL APPROACH

During the algorithm execution, we don't consider all available choices at any given point, but use a heuristic (greedy choice) to pick just one.

What is Greedy Algorithm?

There is no formal definition...

- It is used to solve optimization problems
- It makes a local optimal choice at each step
- Earlier decisions are never undone
- Does not always yield the optimal solution

Elements of the greedy strategy

There is no guarantee that such a greedy algorithm exists, however a problem to be solved must obey the following two common properties:

greedy-choice property

and

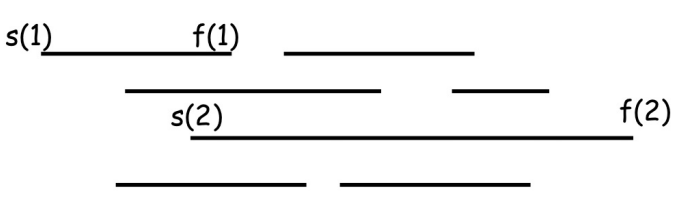
optimal substructure.

The proof of optimal substructure correctness is usually by induction.

The proof that a greedy choice for each subproblem yields a globally optimal solution is usually by contradiction.

Scheduling Problem

There is a set of n requests. Each request i has a starting time s(i) and finish time f(i). Assume that all requests are equally important and $s(i) \leq f(i)$. Our goal is to develop a greedy algorithm that finds the largest compatible (non-overlapping) subset of requests.



① Sort by $s(i)$, choose the earliest time

ALG: A
OPT: B, C

② Sort by $f(i) - s(i)$

ALG: C
OPT: A, B

③ Sort by $f(i)$

example 1 : ALG: B, C
example 2 : ALG: A, B.

Proof

In this approach we sort requests with respect to $f(i)$ in ascending order. Pick a request that has the earliest finish time. We will need to prove that we can do the same or better but not worse w/o knowing what the optimal is.

ALG: $i_1, i_2, \ldots i_k$ OPT: $j_1, j_2, \ldots j_m$

First let us prove how i is related to j

Prove $f(i_r) \leq f(j_r)$ by induction on r

Baselase : r = 1, first request

$f(i_1) \leq f(j_1)$

IH : assume that $f(i_{r-1}) \leq f(j_{r-1})$

for (r-1) requests

IS : prove it for the next request

$f(i_{r-1}) \leq f(j_{r-1}) \leq s(j_r)$

IH

what does it mean?

∃ another request, ALG will take it.

Next step : prove $k \geq m$

proof by contradiction

Assume $k < m$.

① $f(j_k) \leq s(j_{k+1})$ compatible requests

② $f(i_k) \leq f(j_k)$ by IH

P.T.O

Combine them together:

$$f(i_k) \leq f(j_k) \leq s(j_{k+1})$$

$\exists$ another request, our ALG will pick it.

$\therefore \quad k \geq m$

GENERAL STEPS:

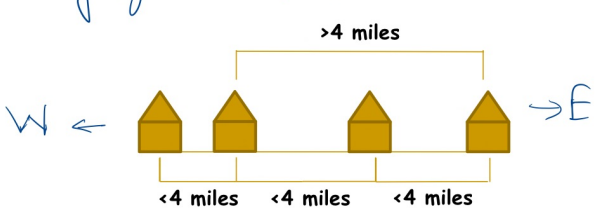
→ 1st prove by induction for substructure

→ Then prove that our ALG is optimal by contradiction. Assume ALG not optimal.

Discussion Problem 1

Let's consider a long, quiet country road with n houses scattered very sparsely along it. We can picture the road as a long line segment, with an eastern endpoint and a western endpoint. You want to place cell phone base stations at certain points along the road so that every house is within four miles of one of the base stations. Give an efficient algorithm that achieves this goal and uses as few base stations as possible.

input: array of houses, h_k



Algorithm:

① Sort h_K, from W to E

② Start with h_1, walk 4 miles & put a station

③ Mark all houses within 4 miles

Runtime = $O(n \log n + n)$

Proof of correctness

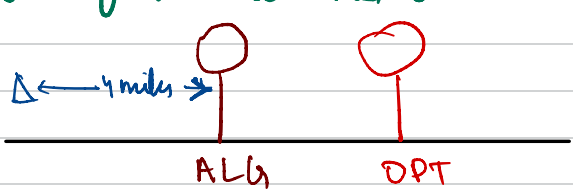
ALG: $s_1, s_2 \ldots s_K$ OPT: $t_1, t_2 \ldots t_m$

Base case: first station.

IH: assume (c-1) stations

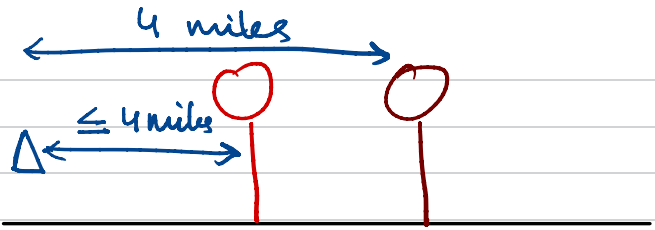
IS: prove for the next station.

A)



impossible

B)



OPT # of stations > ALG # stations

MINIMUM SPANNING TREE

Kruskals algorithm

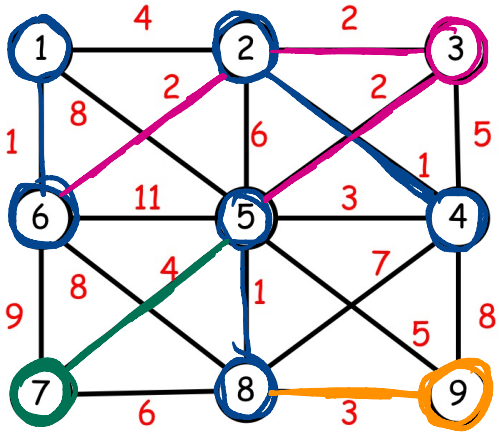
The algorithm builds a tree one EDGE at a time:

- Sort all edges by their weights. $O(E \log E)$
- Loop:
  - Choose the minimum weight edge and join correspondent vertices (subject to cycles). detecting a cycle = $O(V)$
  - Go to the next edge.
  - Continue to grow the forest until all vertices are connected.

Total runtime = $O(V*E + E \log E)$

$V*E$: E times cycle detection

$E \log E$: Sorting edges by weight



Excercise: suppose you do not sort edges

$O(\underbrace{V \cdot E}_{\text{cycle detection}} + \underbrace{E \cdot E}_{\text{find min}})$

Discussion Problem 2

You are given a graph G with all distinct edge costs. Let T be a minimum spanning tree for G. Now suppose that we replace each edge weight c_e by its square, c_e^2, thereby creating a new graph G_1 with the different distinct weights. Prove or disprove whether T is still an MST for this new graph G_1. $c \to c^2$

$MST(G) = T$

$MST(G_1) = T_1$

Case 1: $\forall\, c \geq 0$, then $T = T_1$

$\frac{1}{2}, 1, 2$

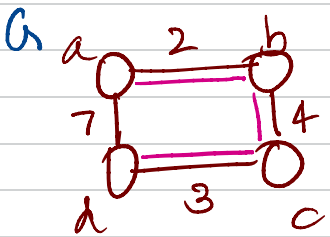
$\frac{1}{4}, 1, 4$ } same order

Case 2: $\exists\, c < 0$, then $T \neq T_1$ FALSE

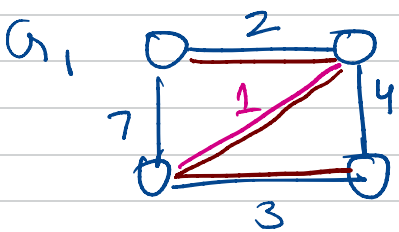
sorting order may change

Discussion Problem 3

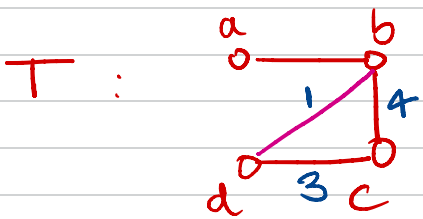
You are given a minimum spanning tree T in a graph G = (V, E). Suppose we add a new edge (without introducing any new vertices) to G creating a new graph G_1. Devise a linear time algorithm to find an MST in G_1.



MST(G) = T



$MST(G_1) = T_1$



remove b-c here

Algorithm:

1. Add new edge to T_1. This will create a cycle.
2. Traverse the cycle
3. Remove the largest edge

Prims Algorithm

The algorithm builds a tree one VERTEX at a time:

- Start with an arbitrary vertex as a sub-tree C.

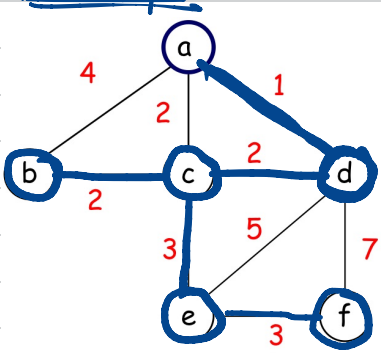
- Expand C by adding a vertex having the minimum weight edge of the graph having exactly one end point in C.

- Update distances from C to adjacent vertices.

- Continue to grow the tree until C gets all vertices.

- We make use of a Heap and for updating & deleting values we use deleteMin & decreaseKey

Example:



T={a}

root ± deleteMin

b-4	c-2	d-1	e-∞	f-∞

T={a,d}

root deleteMin decreaseKey decreaseKey

b-4	c-2	e-5	f-7

T={a,d,c}

root deleteMin decreaseKey

b-2	e-3	f-7

T={a,d,c,b}

e-3	f-7

T={a,d,c,b,e}

root ± decreaseKey deleteMin

f-3

T={a,d,c,b,e,f}

Complexity of PRIMS

1. deleteMin, run V times
2. decreaseKey, E times

Runtime:

Binary Heap : $O(E \cdot \log V + V \cdot \log V)$

Fibonacci Heap: $O(V \cdot \log V + E \cdot 1)$ amortized cost

---

Discussion Problem 4

Assume that an unsorted array is used as a heap.
What would the running time of the Prim algorithm?

$O(V \cdot V + E \cdot 1)$

It takes **linear** time to traverse through the array to find the minimum value for **deleteMin**.

It takes **constant** time to **decreaseKey** in an array as there are indices used to access elements.

Assume that we need to find an MST in a dense graph using Prim's algorithm. Which implementation (heap or array) shall we use?

In a **dense** graph $E = O(V^2)$

∴ The runtime complexity
- Binary Heap : $O(V^2 \log V)$
- Array : $O(V^2)$

∴ use ARRAYs for DENSE GRAPHS

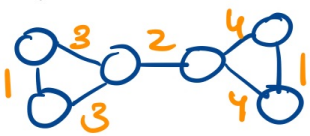
Discussion T/F Questions

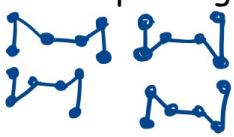
Proof By Example

(T/F) The first edge added by Kruskal's algorithm can be the last edge added by Prim's algorithm.

Proof By Example

(T/F) Suppose we have a graph where each edge weight value appears at most twice. Then, there are at most two minimum spanning trees in this graph.

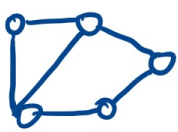




4

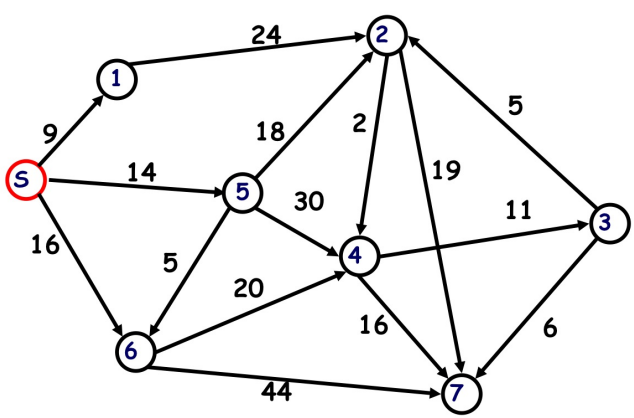
(T/F) If a connected undirected graph $G = (V, E)$ has $V + 1$ edges, we can find the minimum spanning tree of G in $O(V)$ runtime. Assignment 4

$V = 5$
$E = 6$



The Shortest Path Problem

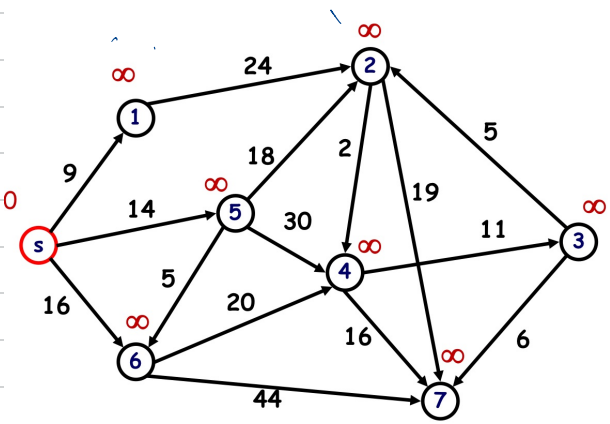
What is the shortest distance from s to 4?



Greedy approach

When algorithm proceeds all vertices are divided into two groups
- vertices whose shortest path from the source is known
- vertices whose shortest path from the source is NOT discovered yet.

Example :



solution tree = S

heap : {1, 2, 3, 4, 5, 6, 7}

Please refer lecture slide 29