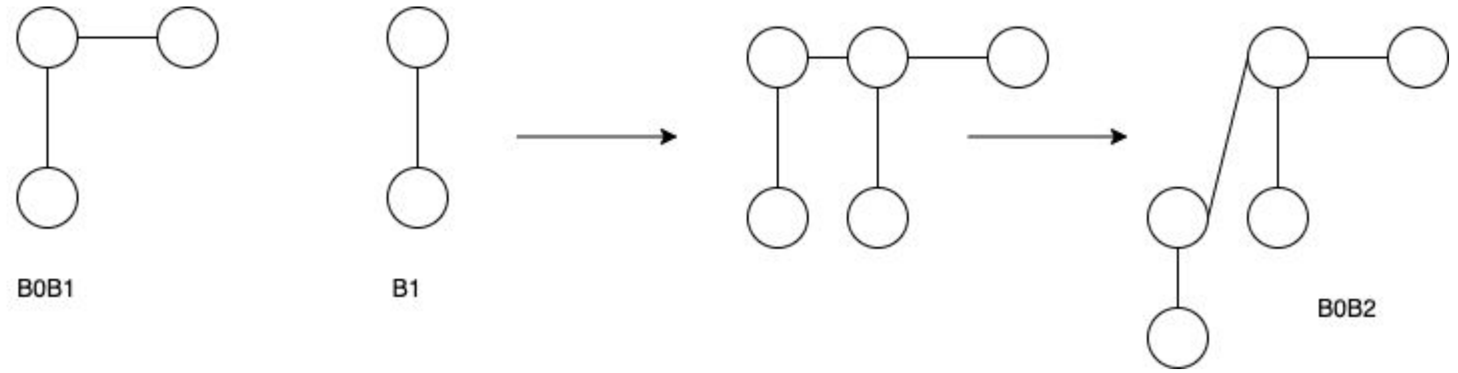
USC University of Southern California

CSCI 570 Exam 1 Review

Heaps and MST

Sanjeev

Merge in Binomial heaps



11 + 10 = 101 → B2B0

Question 1

Merge two binomial heaps

B0B1B2B4 and B1B4

What would be they sizes of individual component trees?

B0B1B2B4 → 10111

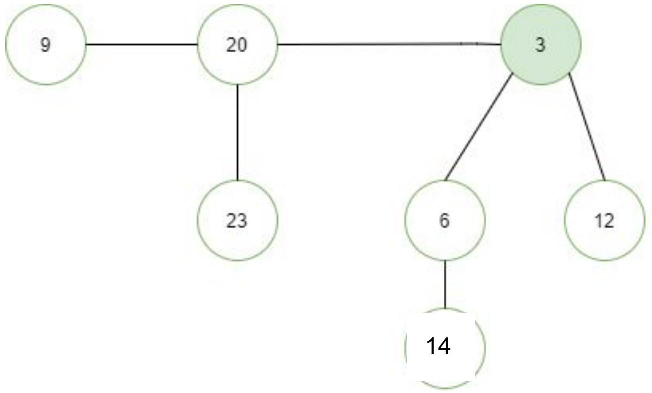
B1B4 → 10010

Perform Binary addition

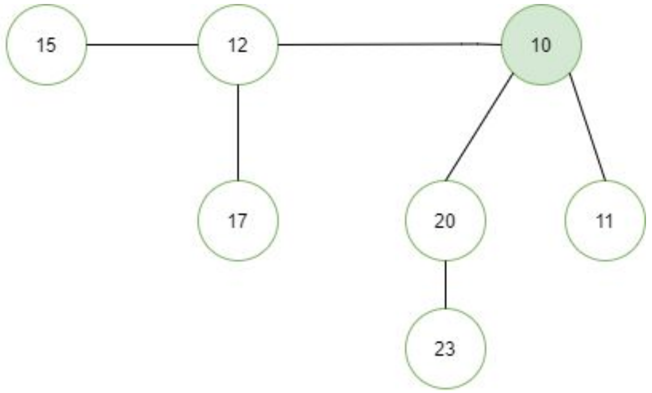
Result → 101001

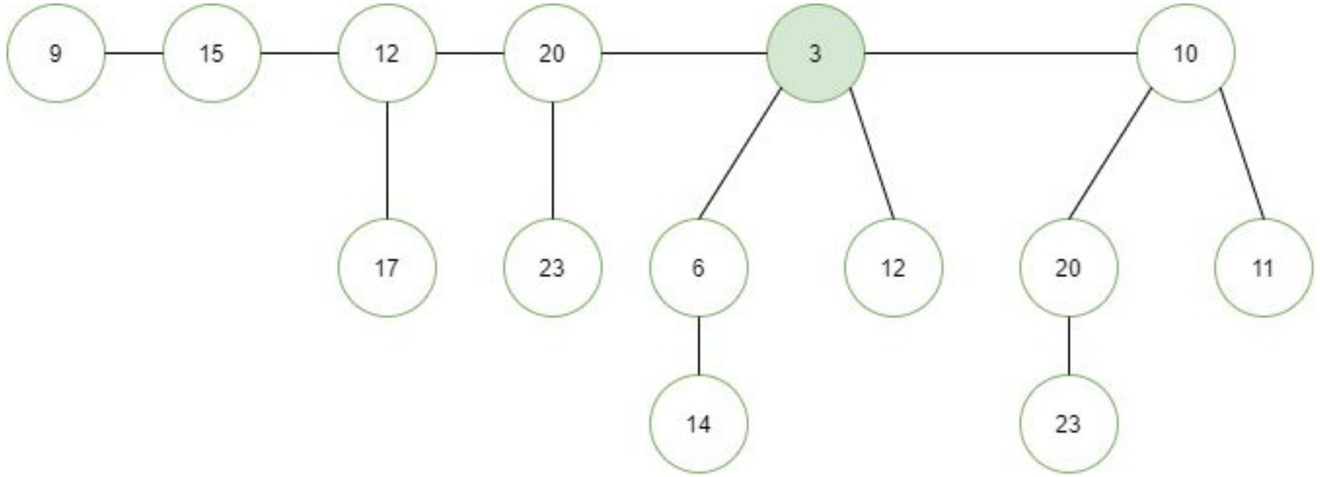
Heap after merging → B5B3B0

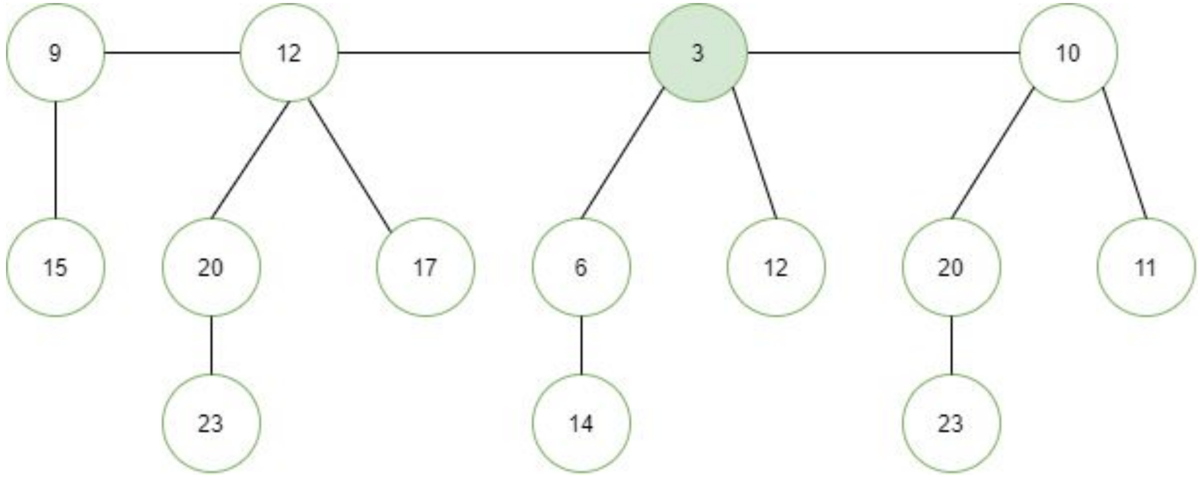
Question 2

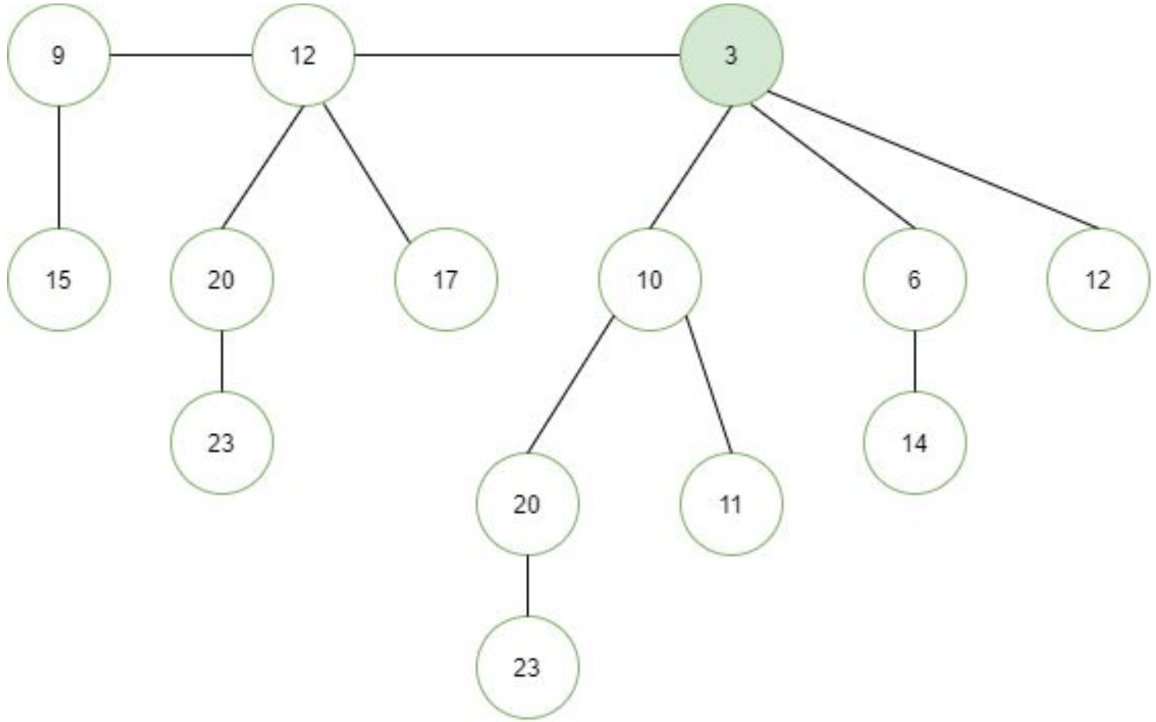


Merge with

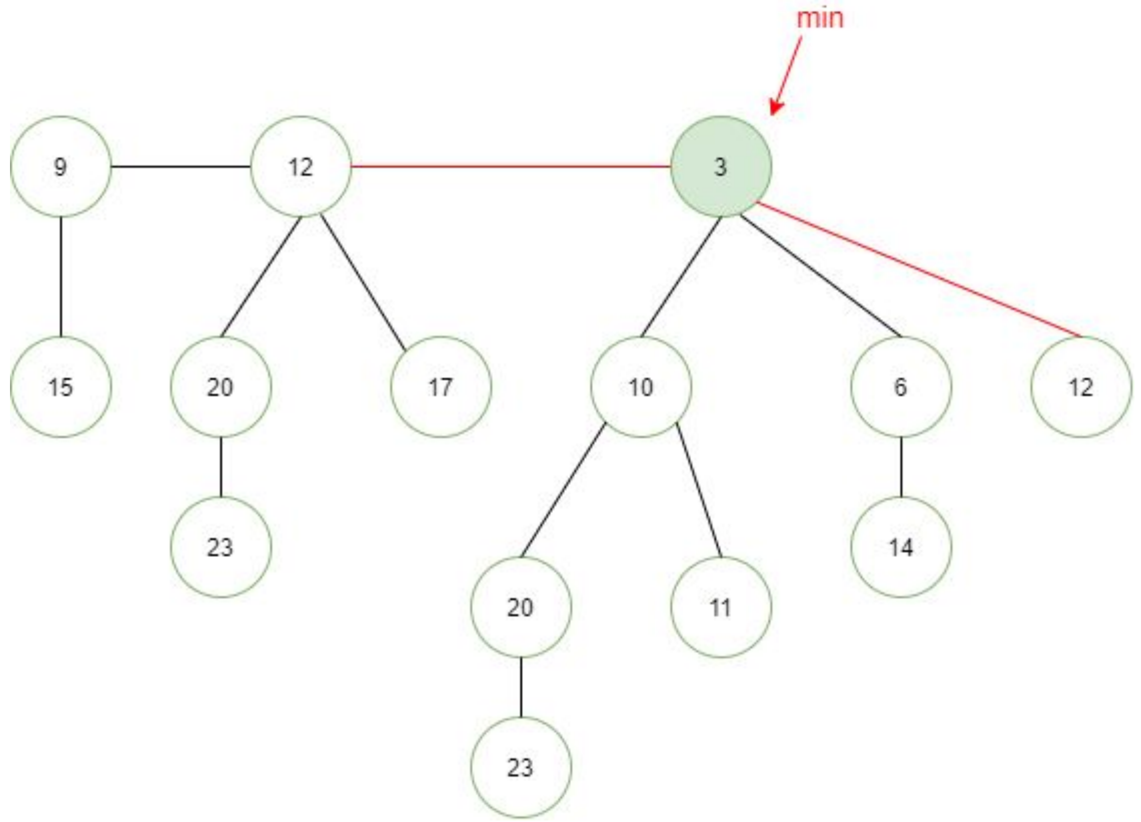


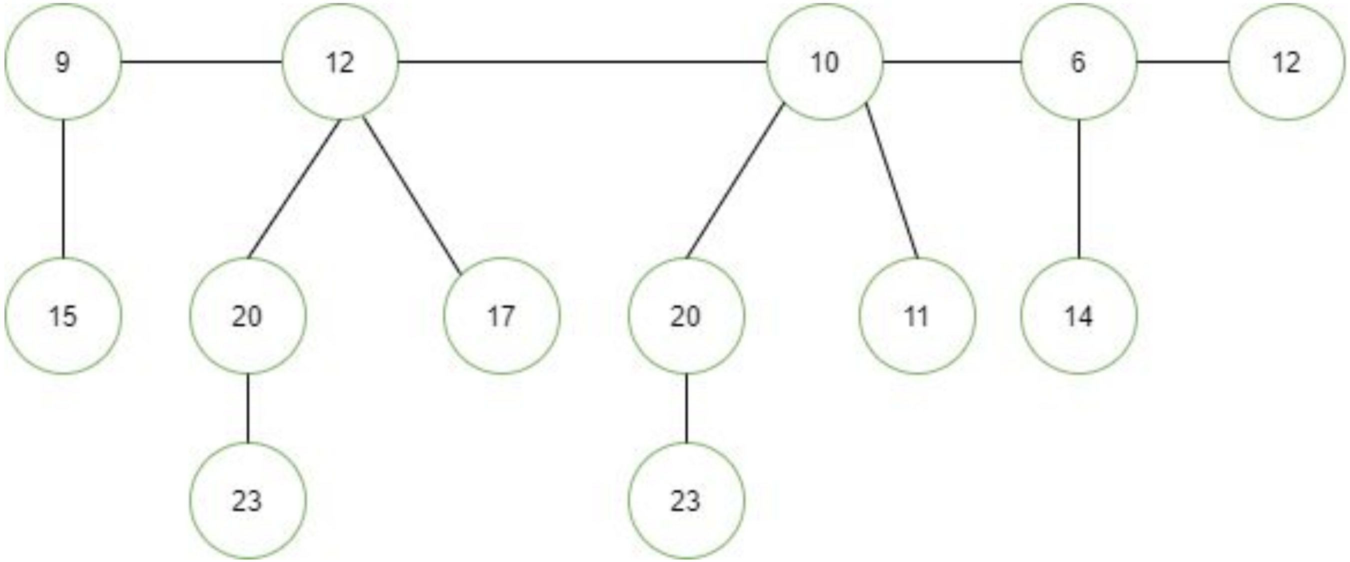

9
15
12
20
3
10
17
23
6
12
20
11
14
23

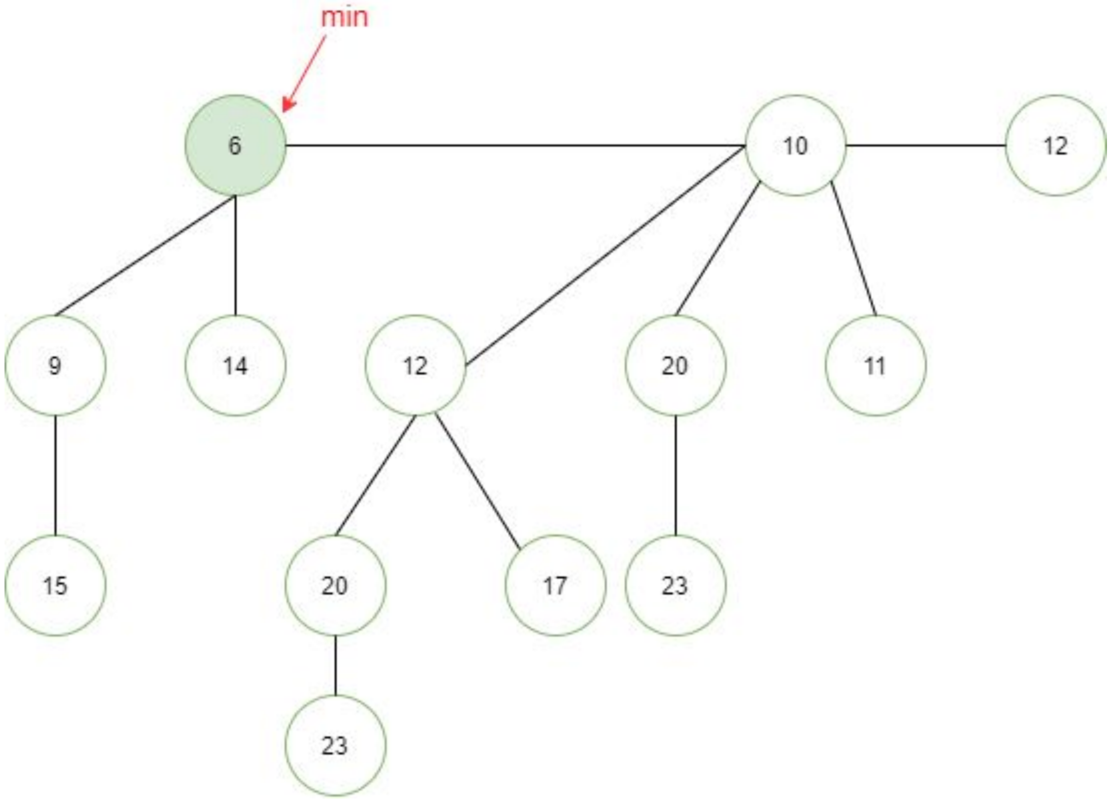

9
12
3
10
15
20
17
6
12
20
11
23
14
23


9
12
3
15
20
17
10
6
12
23
20
11
14
23

deleteMin()




9
12
10
6
12
15
20
17
20
11
14
23
23


min
6
10
12
9
14
12
20
11
15
20
17
23
23

Meeting Scheduler

There are N meetings to be scheduled. A company wants to make sure the first K meetings get scheduled on-time. Rest of the meetings can get delayed. Find the least number of rooms required to schedule the first K meetings exactly at their starting time. You are given start time S_i and end time E_i of each meeting.

Can you do it in O(NlogN) complexity?

Can you do better? O(N + KlogN)?

Which data structure do you think would be the best?

Solution:

Create pairs of <timestamp, startTimeBool> true if start time, else false

Create a MaxHeap of all the pairs

Create pairs of <timestamp, startTimeBool> true if start time, else false

Create a MaxHeap of all the pairs

Realize its not MaxHeap but the one which is better here is MinHeap!!

Keep a variable to track the maxRoomCount, numMeetings

Pop the pairs: Increment/Decrement the curRoomCount based on the bool

When numMeetings reach K, we stop and return maxRoomCount

Complexity: $O(N + K \log N)$

[T/F] The smallest element in a binary max-heap of size n can be found with at most n/2 comparisons.

[T/F] The smallest element in a binary max-heap of size n can be found with at most n/2 comparisons.

True. In a max heap, the smallest element is always present at a leaf node. So we need to check for all leaf nodes for the minimum value → how many leaf nodes are there?
ceil(n/2) leaf nodes. → Therefore, we will only have to do at most n/2 comparisons.

MST 1

[T/F] If all edge weights of a given graph is the same, then every spanning tree of that graph is minimum.

[T/F] If all edge weights of a given graph is the same, then every spanning tree of that graph is minimum.

True.

Spanning tree: Any tree that covers all nodes of a graph is called a spanning tree.

MST 2

[T/F] If the weight of each edge in a connected graph is distinct, then the graph contains exactly one unique minimum spanning tree.

[T/F] If the weight of each edge in a connected graph is distinct, then the graph contains exactly one unique minimum spanning tree.

True.

Proof: Suppose two MSTs T1 and T2 of G,

E = set of edges that is in either T1 or T2 but not both.

e is the min edge in E.

Suppose e is not in T2. Adding e to T2 creates a cycle. Then in this cycle, at least one edge, say f, is not in T1. Since e is te min edge in E, then w(e) <= w(f). And all edges have distinct weights, w(e) < w(f). Replacing f with e results in a new spanning tree with less weight than T1 and T2. Therefore, contradiction.

MST 3

There are n cities and existing roads R_u. The government plans to build more roads such that any pair of cities are connected by roads. The candidate roads are R_p and each road has a cost c_i (c_i > 0). Design an algorithm to decide the roads to be built such that all cities are connected by road(s) and the cost is minimized.

MST 3

1: Construct a graph with n nodes.

2: Add R_u to the graph and set edge weight to be 0.

3: Add R_p to the graph and set edge weights to be the cost.

4: Run MST algorithm, return roads that are in both R_p and MST.

1: Get connected components of the graph with R_u.

2: Merge nodes and edges.

3: Add R_p, run MST.

Stable Matching

Xubin Zhang

Stable Matching

- Given two sets M and W
- Each member of one set has a strict preference order for each member of the other set
- Our goal is a stable matching
    - Perfect Matching: each member of M and each member of W appear in exactly one pair (m, w)
    - No instabilities: where in pairs (m, w) and (m’, w’)
        - m prefers w’ to w’
        - w’ prefers m to m’

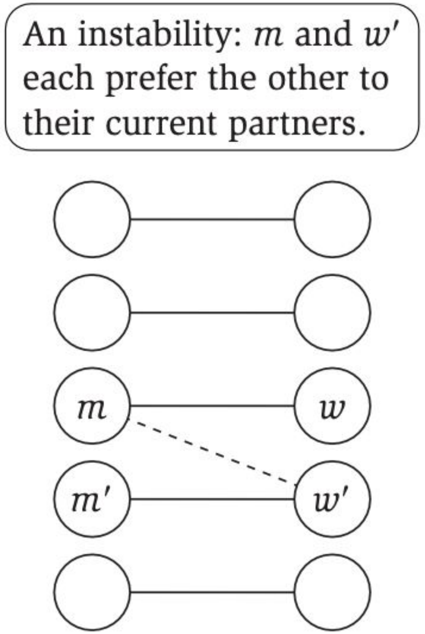

An instability: m and w' each prefer the other to their current partners.
m
w
m'
w'

Figure 1.1 Perfect matching S with instability (m, w').

Gale-Shapley Algorithm

```
Initialize all m in M and w in W to be free
    while there is a man m that is free and has not proposed to every woman:
        Let w be m’s highest ranking women he has not proposed to
            If w is free or prefers m to her current partner m’:
                engage (m, w)
                m’ (if exists) becomes free
```

Runtime: $O(n^2)$

Gale-Shapley Properties

- GS will always correctly return a stable matching
- The proposing side receives their best valid stable matches
- The accept/reject side receives their worst valid stable matches
- Gale-Shapley algorithm is a greedy algorithm

True or False

- There are at least 2 distinct solutions to the stable matching problem: one that is preferred by men and one that is preferred by women.
- False:
  - Consider the example: two men X and Y; two women A and B.
  - Preference list (decreasing order):
  - X’s list: A, B;   A’s list: X, Y;
  - Y’s list: A, B;   B’s list: X, Y.
  - The matching is unique: X-A and Y-B.

Prove that, if all men have the same list of preferences, there can be only one stable matching.

- 4 steps:
- 1. Prove that the first woman in the ranking has to be paired with her first choice in any stable pairing
    - the first woman in the ranking has to be paired with her first choice in any stable pairing, otherwise she and her first choice would form an instability since her first choice prefers her over any other woman

- 2. Prove that the second woman has to be paired with her first choice if that choice is not the same as the first woman's first choice. Otherwise she has to be paired with her second choice.
  - If the first and second women have different first choices, then the second woman must be matched to her first choice. Otherwise she and her first choice would form an instability (since her first choice is not matched to the first woman, he would prefer the second woman over his current match).
  - If the first choices are the same, then the second woman must be paired with her second choice, otherwise she and her second choice would form an instability (neither of them are matched to their first choices, and they are each other's second choice).

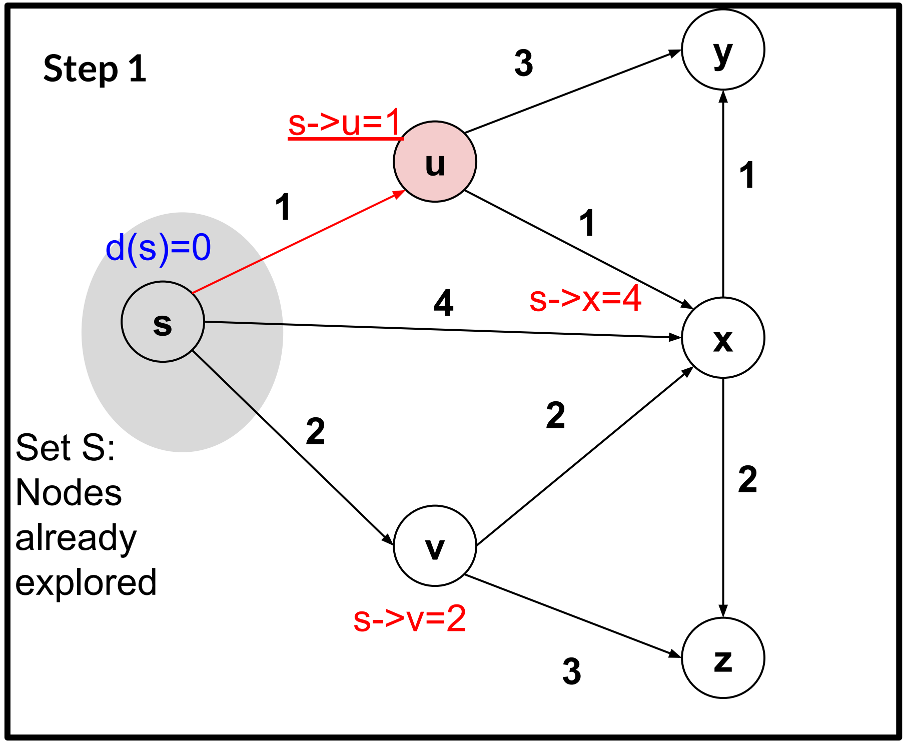
- 3. Continuing this way, assume that we have determined the pairs for the first $k-1$ women in the ranking. Who should the k-th woman be paired with?
    - The k-th woman should be paired with the first man on her list who has not been matched yet (with the first $k-1$ women). If she's not matched to him, they would form an instability. This is because the man would have to be matched to a woman ranked worse than k, so she would prefer the k-th woman over his current partner, and the k-th woman obviously prefers him to whoever she's matched with.

- 4. Prove that there is a unique stable pairing.
    - In the previous parts, we saw that for each woman, given the pairs for the lower-ranked women, her pair would be determined uniquely. So there is only one stable pairing.
    - This can be stated and proved more rigorously using induction. **Namely that there is a unique pairing for the first *k* women, assuming stability.** An induction on k would prove this.

Shortest Path

Nan Xu

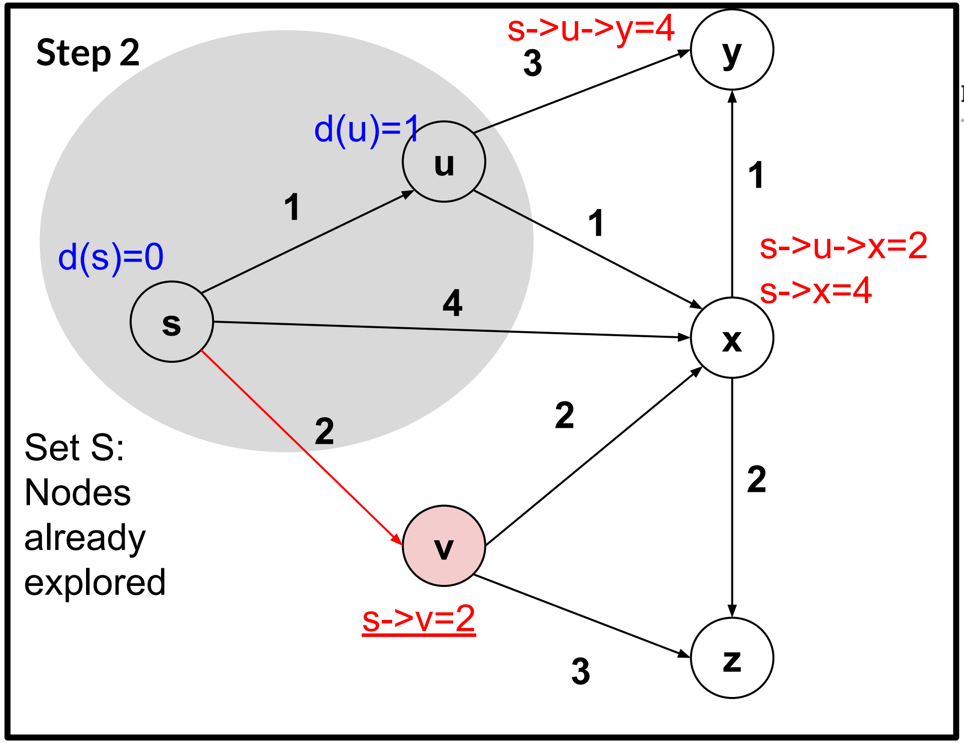
Recap: Dijkstra's Algorithm



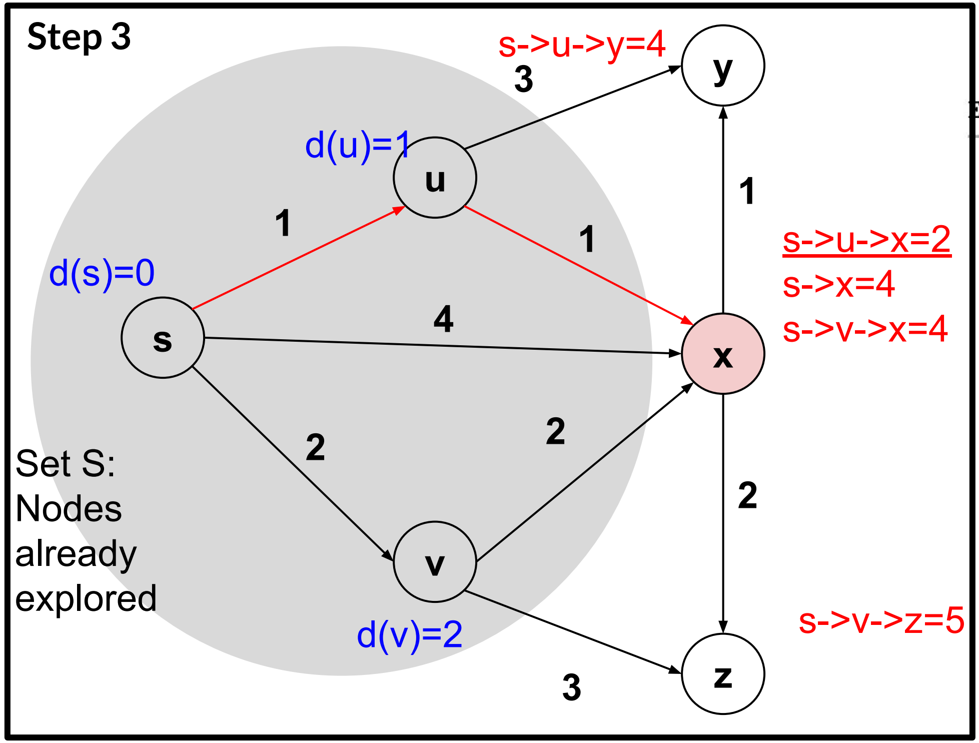
```
Dijkstra's Algorithm $(G, \ell)$
Let $S$ be the set of explored nodes
    For each $u \in S$, we store a distance $d(u)$
Initially $S = \{s\}$ and $d(s) = 0$
While $S \neq V$
    Select a node $v \notin S$ with at least one edge from $S$ for which
        $d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$ is as small as possible
    Add $v$ to $S$ and define $d(v) = d'(v)$
EndWhile
```

Recap of Dijkstra's Algorithm

```
Dijkstra's Algorithm ($G, \ell$)
Let $S$ be the set of explored nodes
    For each $u \in S$, we store a distance $d(u)$
Initially $S = \{s\}$ and $d(s) = 0$
While $S \neq V$
    Select a node $v \notin S$ with at least one edge from $S$ for which
        $d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$ is as small as possible
    Add $v$ to $S$ and define $d(v) = d'(v)$
EndWhile
```

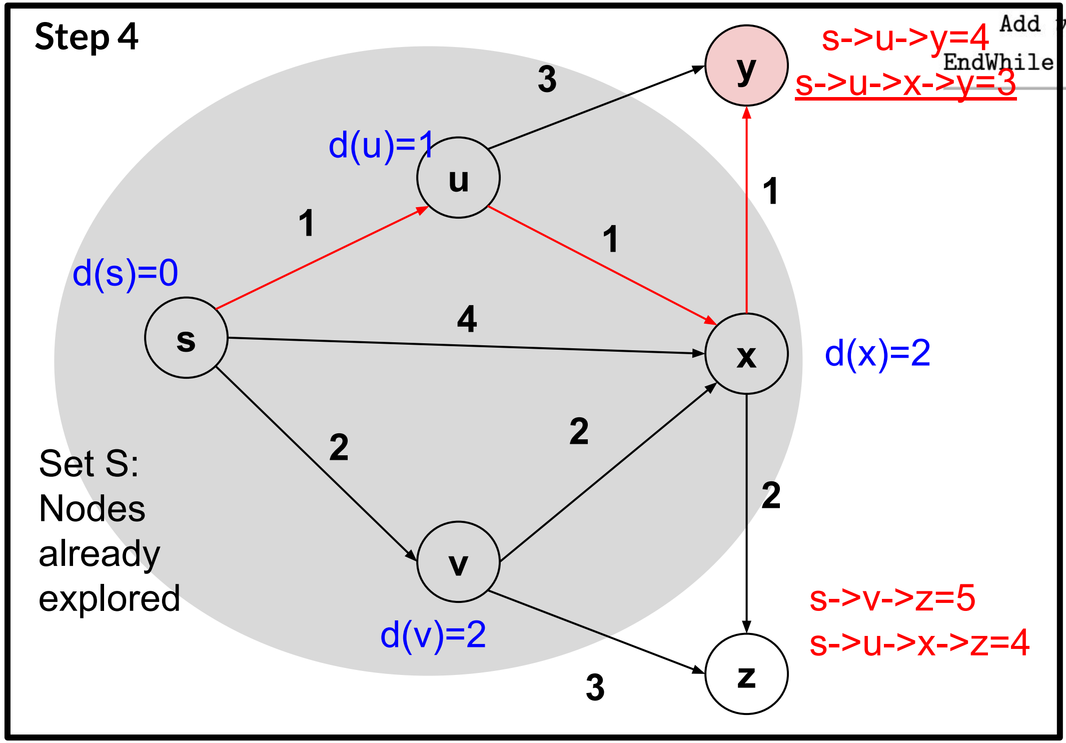


Recap of Dijkstra's Algorithm



```
Dijkstra's Algorithm $(G, \ell)$
Let $S$ be the set of explored nodes
    For each $u \in S$, we store a distance $d(u)$
Initially $S = \{s\}$ and $d(s) = 0$
While $S \neq V$
    Select a node $v \notin S$ with at least one edge from $S$ for which
        $d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$ is as small as possible
    Add $v$ to $S$ and define $d(v) = d'(v)$
EndWhile
```

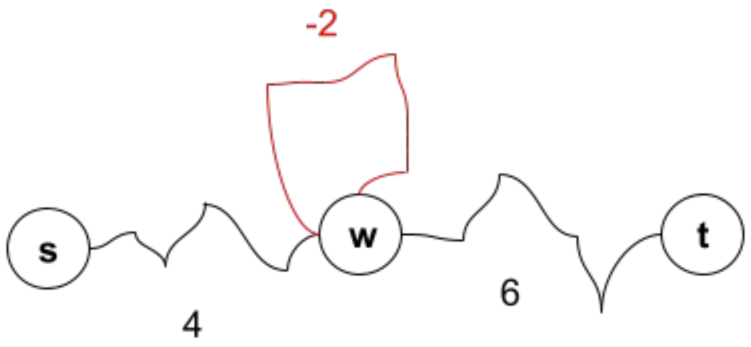
Recap of Dijkstra's Algorithm



```
Dijkstra's Algorithm $(G, \ell)$
Let $S$ be the set of explored nodes
    For each $u \in S$, we store a distance $d(u)$
Initially $S = \{s\}$ and $d(s) = 0$
While $S \neq V$
    Select a node $v \notin S$ with at least one edge from $S$ for which
        $d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$ is as small as possible
    Add $v$ to $S$ and define $d(v) = d'(v)$
EndWhile
```

NOTE: Negative Weights - Negative Cycles

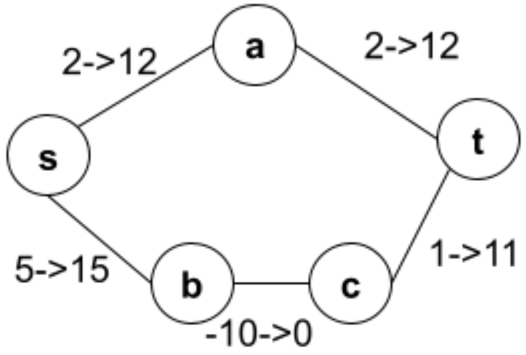
- Negative cycles: if some cycle has a negative total cost, we can make the s-t path as low cost as we want



s->w->t: 4+6=10
s->w (cycle once)->t: 4-2+6=8
s->w (cycle twice)->t: 4-2*2+6=6
…
s->w (cycle N times)->t: 4-2*N+6=-∞, when N->∞

NOTE: Negative Weights - Solve with a Big Number?

- Adding a large number M to each edge DOES NOT work
  - New cost of path P: M*len(P)+old_cost(P)
  - When M is big, the number of hops (path length) will dominate rather than old path cost

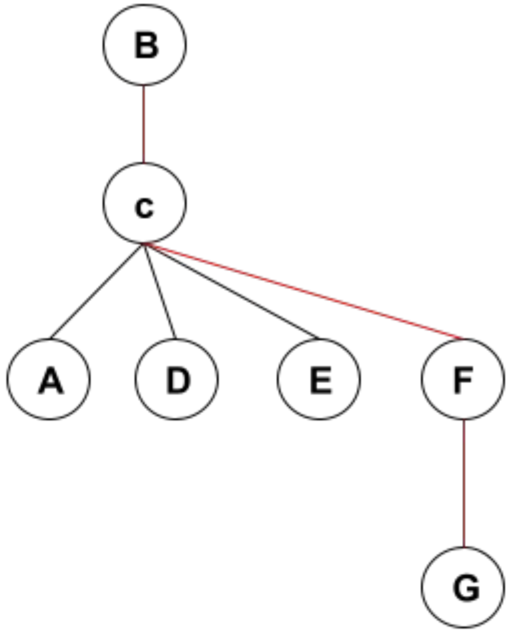


Before: s-b-c-t (-4) is shorter than s-a-t (4)
After adding 10 to all edges: s-a-t (24) is shorter than s-b-c-t (26)

Solution: Bellman-Ford in Dynamic Programming Section

Problem I

- The diameter of a graph is the maximum of the shortest paths' lengths between all pairs of nodes in graph G.
- Design an algorithm which computes the diameter of a connected, undirected, unweighted graph in O(mn) time, and explain why it has that runtime.



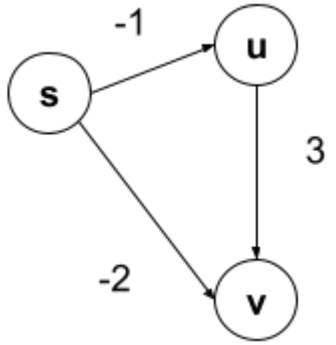
Diameter of this graph is: 3

Solution I

- Unweighted graph => BFS can be used for shortest path search for each source node in O(m+n)
  - For one source node s, report the maximum layer reached
- Repeat BFS for each node in O(n)
- Total time complexity: O(n(m+n))
  - Connected graph, the number of edges are at least n-1, at most n(n-1)/2, hence n=O(m)
  - => O(m+n) is O(m)
  - total time complexity: O(nm)

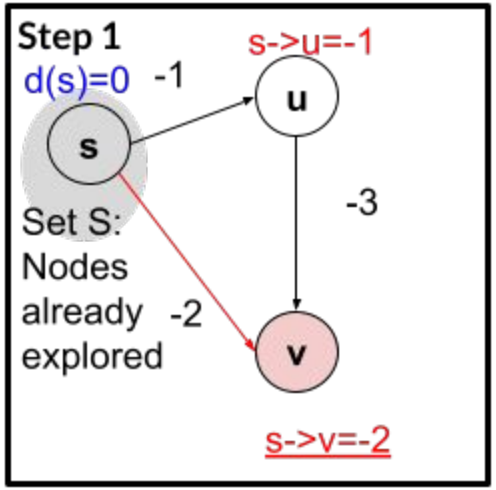
Problem II:

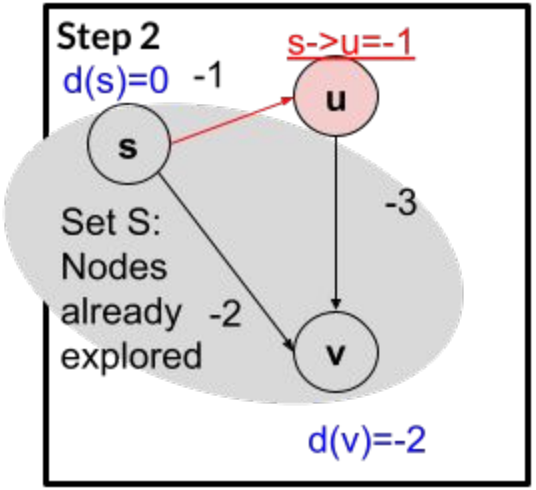
- Dijkstra’s algorithm works correctly on a directed acyclic graph even when there are negative-weight edges.
- TRUE or FALSE



Solution II

- False

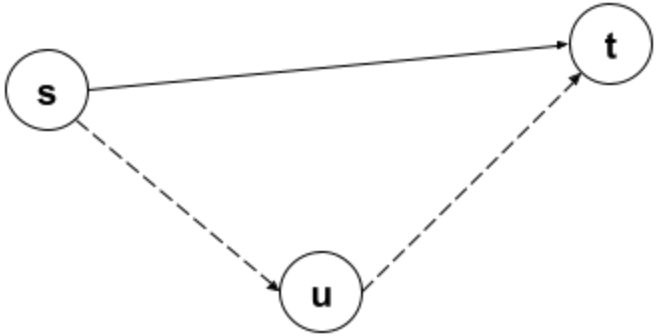




Dijkstra's algorithm: d(v)=-2
Actual shortest path: s-u-v, with cost -4

Problem III

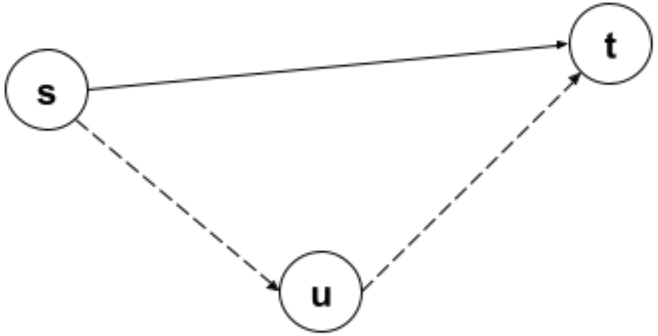
- Suppose that you want to get from vertex s to vertex t in a connected undirected graph G = (V; E) with positive edge costs, but you would like to stop by vertex u (imagine that there are free burgers at u) if it is possible to do so without increasing the length of your path by more than a factor of **α**



- Describe an efficient algorithm in O(|E| log |V|) time that would determine an optimal s-t path given your preference for stopping at u along the way if doing so is not prohibitively costly. (In other words, your algorithm should either return the shortest path from s to t, or the shortest path from s to t containing u, depending on the situation.)

Solution III

- Positive edges => Dijkstra's algorithm for shortest path from source node s, and from source node u
  - shortest path from s => we know $d_s(t)$ and $d_s(u)$
  - Shortest path from u => we know $d_u(t)$
- Compare $d_s(u) + d_u(t)$ and $\alpha^* d_s(t)$
  - If $d_s(u) + d_u(t) \leq \alpha^* d_s(t)$, stop by u for burger!
  - If $d_s(u) + d_u(t) > \alpha^* d_s(t)$, go directly from s to t
- Time complexity
  - Connected graph: |V|=O(|E|)
  - Running Dijkstra's algorithm twice:
    - Binary heap: O((|V| +|E|) log |V|) => O(|E| log |V|)
    - Fibonacci heap: O(|E|+|V|log|V|) => O(|E| log |V|)



USC University of Southern California

CSCI 570 Exam 1 Review

BFS & DFS

Breadth First Search (BFS)

- BFS is a *graph search algorithm*, an algorithm for exploring a graph $G=(V,E)$.
- BFS starts at a vertex s and explores outwards, visiting all nodes distance 1 from s, then all nodes distance 2 from s, and so on.
- BFS can find a path between two vertices s and t, if such a path exists.
- BFS finds the shortest path between s and t if the graph is unweighted.
- BFS is implemented using a *queue* (FIFO).
- The worst-case runtime of BFS is $O(|V| + |E|)$.

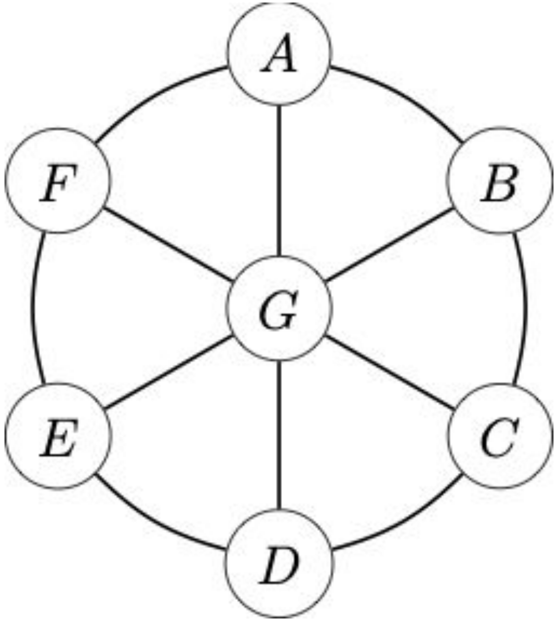
BFS Tree

- A graph search algorithm induces a *search tree* $T \subseteq G$, with root *s* and edges $u \rightarrow v$, where the first time the algorithm explored *v* was by traversing the edge *(u,v)* in *G*.

  (3.3) *For each $j \geq 1$, layer L_j produced by BFS consists of all nodes at distance exactly j from s. There is a path from s to t if and only if t appears in some layer.*

Problem 1

List the order in which vertices are visited when executing BFS starting from vertex E and breaking ties alphabetically.



Answer:
E, D, F, G, C, A, B

Depth First Search (DFS)

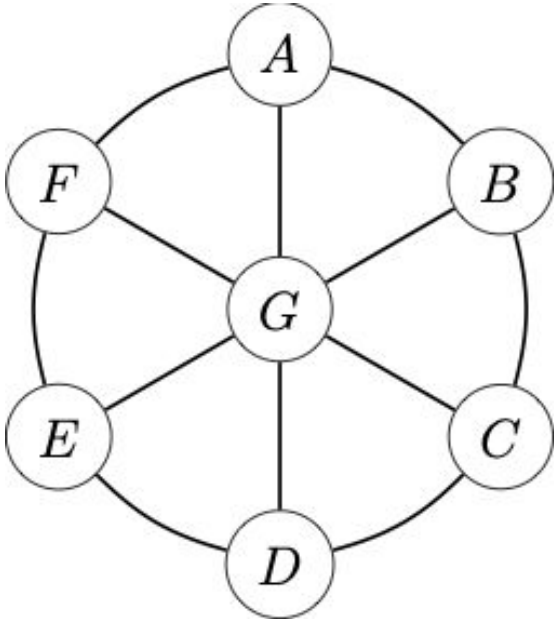
- DFS is also a graph search algorithm.
- DFS starts at a vertex *s* and explores by walking around the graph, backtracking when it hits a dead-end.
- DFS can find a path between two vertices *s* and *t*, if such a path exists.
- DFS is implemented using a *stack* (LIFO).
- The worst-case runtime of DFS is also $O(|V| + |E|)$.

DFS Tree

(3.7) *Let T be a depth-first search tree, let x and y be nodes in T, and let (x, y) be an edge of G that is not an edge of T. Then one of x or y is an ancestor of the other.*

Problem 2

List the order in which vertices are visited when executing DFS starting from vertex E and breaking ties alphabetically.



Answer:
E, D, C, B, A, F, G

BFS v. DFS

- Both BFS and DFS can find a path from *s* to *t*.
- Both BFS and DFS can find the number of connected components in *G*.
- Both run in time $O(|V| + |E|)$.
- BFS is implemented using a queue (FIFO). DFS is implemented using a stack (LIFO).
- BFS can find shortest paths in unweighted graphs.

Problem 3

Let $G=(V,E)$ be an unweighted undirected graph, and let s, t be vertices in G. Describe an algorithm for finding the shortest path from s to t that has a length that is a multiple of 10.

Answer: Construct a graph $G'=(V', E')$ with 10 vertices $v_0, \ldots, v_9$ for each vertex v in G. Draw a directed edge between vertices $(u_i, v_{i+1\%10})$ for each edge $\{u,v\}$ in E. Run BFS or DFS on G' to find a path p' from s_0 to t_0. Then, to construct a path p in G, for each vertex v_i in p', add v to p.

Note G' has $O(10n)$ vertices and $O(10m)$ *edges.* Therefore, running BFS or DFS on G' takes time $O(n + m)$.

USC University of Southern California

CSCI 570 Exam 1 Review

DIVIDE AND CONQUER

Problem 1

Given a sorted array of distinct integers A[1, …, n]. Decide whether there is an index ‘i’ for which A[i] = i.

Describe an efficient divide and conquer algorithm for this problem.

Explain the time complexity.

Solution

- The algorithm is similar to “Binary Search” considering the fact that we have a sorted array
- If the array has just one integer, then we check whether A[0] = 0 with one comparison(Base Case)
- Divide the list into two parts at the middle index: left part and the right part.
- Consider the largest element of the left part A[m] i.e., element at the end of the array. Check A[m] = m, if True, then we return True and we are done.
- If A[m] > m, we discard the right half and continue the algorithm recursively on the left half. For every integer k≥ 0 using the fact that the integers are distinct and sorted, A[m + k] $\geq A[m]$ + k > m + k.
- If A[m] < m, then we can discard the left half and continue the algorithm recursively on the right half. For every integer k≥ 0 using the fact that the integers are distinct and sorted, A[m - k] $\leq A[m]$ + k < m + k.

- Thus, for the number of comparisons we get the following recurrence relation, T(n) = T(n/2) + O(1).
- By using the Master Theorem, the time complexity is O(log n).

Problem 2

A polygon is called convex if all of its internal angles are less than 180º and none of the edges cross each other. We represent a convex polygon in the form of a coordinate pair (x,y). Consider V[1] is the vertex with the least x coordinate and that the vertices V[1], V[2], …, V[n] are ordered counter-clockwise. Assuming that the x and y coordinates of the vertices are distinct:

Give a divide and conquer algorithm to find the vertex with the largest x coordinate in O(log n) time.

Give a divide and conquer algorithm to find the vertex with the largest y coordinate in O(log n) time.

Solution

Since V[1] is known to be the vertex with the minimum x-coordinate (leftmost point), moving counter-clockwise to complete a cycle must first increase the x-coordinates and then after reaching a maximum (rightmost point), should decrease the x-coordinate back to that of V[1]. Thus, Vx[1 : n] is a unimodal array, and so it is synonymous with detecting the location of the maximum element in the array.

Algorithm:

(1) If $n = 1$, return Vx[1].
(2) If $n = 2$, return max{Vx[1], Vx[2]}.
(3) $k = n/2$.
(4) If Vx[k] > Vx[k − 1] and Vx[k] > Vx[k + 1], then return Vx[k].
(5) If Vx[k] < Vx[k − 1] then call the algorithm recursively on Vx[1 : k − 1], else call the algorithm recursively on Vx[k + 1 : n].

Let p denote the index at which the x-coordinate was maximized. Observe that joining V[1]
and V[p] by a straight line divides the polygon into an upper half and a lower half and the vertex
achieving the maximum y-coordinate must lie above this line. The counter-clockwise traversal
$Vy[p] \rightarrow Vy[p+1] \rightarrow \cdots \rightarrow Vy[n] \rightarrow Vy[1]$ is unimodal and we need to detect the location of the
maximum element of this array. So, considering the same algorithm as above with this new array
$[Vy[p], Vy[p+1], \ldots, Vy[n], Vy[1]]$ will return the vertex with the maximum y-coordinate.

The recurrence equation is $T(n) \leq T(n/2) + \Theta(1)$.
Invoking Master's Theorem gives $T(n) = O(\log n)$.

USC University of Southern California

CSCI 570 Exam 1 Review

Greedy

The Slides for Greedy Part are credit to Alex Wang.

Greedy: 2 Common Methods of Proof of Optimality

- **stay-ahead**: an inductive method to show that for each step you take, you are always ahead of any other solution according to some metric

- **exchange**: show that for your greedy solution, any swaps between steps/elements cannot achieve a better result

The Slides for Greedy Part are credit to Alex Wang.

Problem 1

For a graph G = (V, E), a vertex cover of G is a set of vertices that includes at least one endpoint of each edge of G. Assuming our graph G is a tree, find an algorithm that finds a minimum vertex cover of G.

Solution 1

Let S be the set of vertices that will be the vertices we return. If our graph G has no vertices or only 1 vertex, return S. Otherwise, add all vertices 1 layer above all our leaf vertices in the tree to S, and remove the vertices and its adjacent edges from G, as well as the leaf vertices. Recursively apply this step until we end up with 1 or no vertices in G. Return S.

This is optimal because we have to cover the leaf edges regardless, so our cover will always either include the leaf node itself or its parent. If we include its parent, we cover just as much if not more than if we select the leaf node. The problem then reduces to a vertex cover for a subset of the original tree. This subset is smaller than the one we would have gotten from choosing the leaf node, and a vertex cover of tree is at least as large as the vertex cover of a subset of it.

The Slides for Greedy Part are credit to Alex Wang.

Problem 2

Given an array of distinct integers A = [a1, a2, …, an] and a sequence p of numbers p = [1, 2, …, n], rearrange the numbers in p such that $\sum_{i=1}^{n} |a_i - p_i|$ is minimized.

The Slides for Greedy Part are credit to Alex Wang.

Solution 2

Consider 2 real numbers ai and aj, and 2 other real numbers, pi and pj, where ai < aj and pi < pj. WLOG, assume that all 4 of these numbers are distinct. There are 6 possible orderings given these constraints: ai < aj < pi < pj, ai < pi < aj < pj, ai < pi < pj < aj, pi < ai < aj < pj, pi < ai < pj < aj, and pi < pj < ai < aj. For each of these cases, we want to show that |ai - pi| + |aj - pj| ≤ |ai - pj| + |aj - pi|, which basically means that we cannot get a smaller distance if we exchange elements pi and pj. Therefore if we have ai < aj if and only if pi < pj for all i, j, we will have the minimum distance. We can store the index of the elements in A in a hashmap h. We then sort A and pair it with p, and reorder p according to indices of h. Since our algorithm produces elements ai and pi with these properties, our algorithm produces a minimum distance.

The Slides for Greedy Part are credit to Alex Wang.

In the previous proof, we left out details about showing

$|a_i - p_i| + |a_j - p_j| \leq |a_i - p_j| + |a_j - p_i|$

for each of 6 cases. Here is an example of showing that this is true for the case for

$a_i < p_i < a_j < p_j$.

Define the distance between ai and pi, pi and aj, and aj and pj, as d1, d2, and d3 respectively. Then

$|a_i - p_i| + |a_j - p_j| = d_1 + d_3 \leq d_1 + 2*d_2 + d_3 = |a_i - p_j| + |a_j - p_i|$.

You can verify that the other cases are true as well. In the exam, there will not be this much detail involved in a proof, and if there is, we will state which parts can be skipped or assumed to be true.

The Slides for Greedy Part are credit to Alex Wang.