CSCI 570 HW2: Homework set 0

Mounika Mukkamalla - 1001219359

1

a) Algorithm for finding minimum number of rooms required to allocate all meetings

1. Sort the given meetings based on start time; $meetings = [\{st_1, et_1\}, \{st_2, et_2\}, ...\{st_n, et_n\}]$
2. Use a min heap to store end times of meeting; *min_heap*
3. Initialise $current\_free\_rooms$ and $total\_new\_rooms$ to 0.
4. for meet in meetings:
4.i. while *min_heap* is not empty and *min_heap.top*() < start time of *meet*:
4.i.a. Increment $current\_free\_rooms$ by 1.
4.i.b. $DeleteMin()$
4.ii. if($current\_free\_rooms$ is not 0):
4.ii.a. Decrement $current\_free\_rooms$ by 1. // Using existing free rooms for the new meet if available.
4.iii. else
4.iii.a. Increment $total\_new\_rooms$ by 1. // If there are no available rooms use a new room.
4.iv. Insert *end_time* of *meet* to the *min_heap*.
4.v. Go to step 4 and repeat till the all meets are iterated.
5. end;
6. return $total\_new\_rooms$

b) Complexity of the above algorithm is number of times we call $deleteMin()$ function.
In worst case, maximum number of times we call $deleteMin()$ is size of $meetings = n$.
Complexity of $deleteMin()$ is $O(log(n))$.
Hence overall complexity = $O(nlogn)$.

2

a. Algorithm for merging all n sorted departments containing all together m research papers using *min_heap*.

1. Select all the 1st elements of n departments and push into heap.
2. Heapify all n elements.
3. Get the top element of the heap, *min_heap.top*(). Store it in the $result\_array$. And then delete the top element from heap, $min\_heap.deleteMin()$
4. Get the next element from the department in which top element is removed. If department is empty then do nothing.
5. Add to the *min_heap*.
6. Go to step 3. Repeat till all departments are empty.
7. return $result\_array$.

b.
Step1: O(n)
Step2: O(n)
Step3: O(log(n))
Step4: O(1)
Step5: O(log(n))
Step 3 to 5 are repeated almost m number of times – almost is used because in step 1, we already selected 1st elements from n departments.

Hence assymptotic complexity of overall algorithm: O(mlogn).

3

Algorithm to find the minimum number of space station refueling stops required to reach target space station with initial fuel capacity, FC
Assume given array of pair of distances and fuel capacities as $[\{0,0\},\{d_1,fc_1\},\{d_2,fc_2\},...,\{d_n,fc_n\},\{targetDistance,0\}$
Where $\{0,0\}$ represents the celestial origin, and $\{targetDistance,0\}$ represents target star.
We maintain a max_heap of fuel capacities of space stations.
Let current fuel capacity be CFC
Algo:
0. Sort the given array based on distances, d_i
1. Initialise $CFC = FC, i = 0, numOfStations = 0$;
2. Define max_heap
3. while i != n+1:
3.i. if($CFC >= fc_i$)
3.i.a. increment i by one;
3.ii. else :
3.ii.a. if max_heap is empty
3.ii.b. return -1;
3.ii.c. else:
3.ii.c.1. $CFC = CFC + max\_heap.top()$
3.ii.c.2. $DeleteMax()$
3.ii.c.3. increment $numOfStations$ by one.
4. return $numOfStations$

Time complexity of above algorithm:
Loop runs O(n) times and inside each loop, costliest operation is $DeleteMax()$ which take $O(log(n))$ time.
Hence overall worst case complexity = $O(nlog(n))$.

4

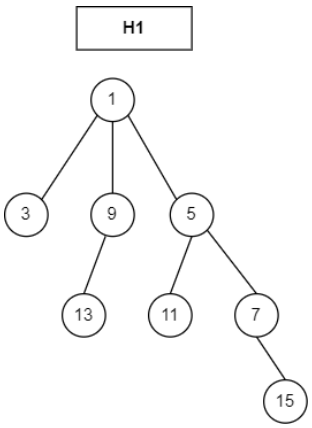

Figure 1: H1

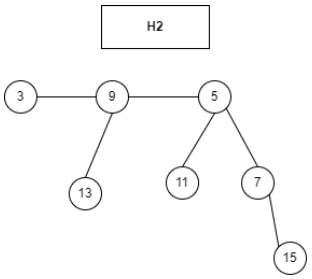

Figure 2: H2

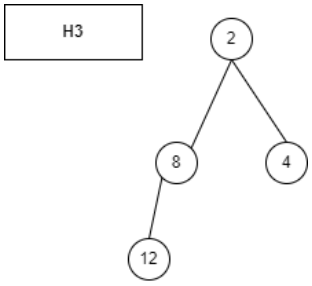

Figure 3: H3

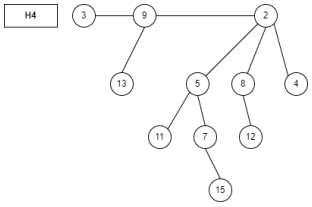

Figure 4: H4

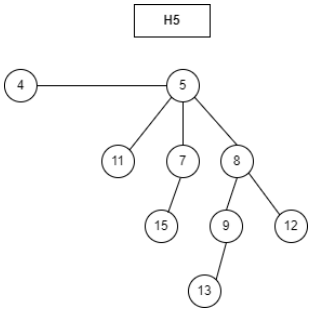

Figure 5: H5

5

BaseCase: Consider a 2^{nd} order binomial tree.
For a 2^{nd} order binomial tree, number of nodes = 3 and number of children for the root node = 2.
When root node is deleted we are left with 2 individual nodes not connected with each other. and hence a forest of 2 1^{st} order binomial trees with one node.
Given statement is true for base case.

Inductive Hypothesis: Assume that the given statement is true for k = r.
i.e., For a B_r^{th} binomial tree, when root node is deleted, we get forest of r small order binomial trees.

Inductive Step: We need to prove statement is true for $r + 1$.
Consider we have 2 r^{th} order binomial trees. $tree\_1$, $tree\_2$
Merging these 2 trees results in $tree\_0$:
a) Get the minimum of roots of 2 binomial trees, min_root. Assume $tree\_1$ has minimum root value.
b) Add the other root, $tree\_2$ as a child to min_root.
Hence $tree\_0$ is formed with min_root as root.
Hence number of children to the root node of $tree\_0 = 1 + r$.
When root node is deleted, all the edges to the children are deleted leading to forest of r+1 graphs.
From Inductive step, we know that r children from $tree\_1$ are already binomial trees with smaller orders than r, and the additional child added to the $tree\_0$ is $tree\_2$ which is also a binomial tree of r order.
Hence, we can safely say that r+1 graphs are r+1 binomial trees of smaller orders.
Therefore, Inductive Step is proved.

6

Let's take krushkal's algorithm for building a Minimum Spanning Tree.
Let's take cyclic property, if there is a cycle in a graph, and if there is an edge e in the cycle with highest weight compared to all other individual weights in the graph then that edge e does not exist in MST.
We can modify it as:
Consider 2 vertices, v,w. Edge e does not exist in the path from v and w in MST until all the edge costs in the path are less than edge e.

Algorithm:
a) Traverse through the edges and delete all the edges with cost greater than or equal to e.
b) By DFS traverse through the vertices from v to w, if there exists a path then edge e is not in MST else edge e is in the MST.

Complexity:
Step a: O(E)
Step b: O(V+E)
Overall Complexity: O(V+E).

7

Given E = V+10
Using cyclic property, if there is a cycle in a graph, and if there is an edge e in the cycle with highest weight compared to all other individual weights in the graph then that edge e does not exist in MST.
This implies, we can eliminate highest weighted edge whenever we find a cycle. Since edge weights are distinct, there would be only one highest weighted edge.
We use DFS to detect cycle.

Algorithm:

1. initialise $i = 0, visited\_array = [], edges\_stack$
2. while $i < 11$:
2.i. start from any vertex
2.ii. traverse the graph using DFS, and store the visited nodes in *visited_array*.
2.ii.a. While traversing, if we find a node, u, which is already visited then a cycle is detected.
2.iii. Start from u.
2.iv. traverse the graph using dfs until we reach u again.
2.iv.a. While traversing push the edges of the path in *edges_stack*.
2.iv.b. When you backtrack from child to parent node, pop that edge from the *edges_stack*.
2.v. delete one maximum weighted edge from the detected cycle path.
2.vi. Empty *edges_stack* and *visited_array*.
2.vii. increment i by 1.

We run the loop 11 times because, for an MST, number of edges, E = V-1 and total number of edges are V + 10, hence 11 edges are additional which we remove by detecting cycles.

Time Complexity:
Detecting the cycle required O(V+E) time complexity. and since cycle detection is run limited number of times, 11, and E is O(V), Overall time complexity remains O(V).

8

Algorithm:
1. Sort the weights. Sorted array, $W\_sorted = [w_1, w_2, ..., w_n]$
2. Maintain 2 pointers, one for left to right traversal, i and one for right to left, j.
3. initialise $j = n - 1, count = 0$
4. Traverse from left to right: for i in range(0, n-1):
4.a. while $j >= i$:
4.a.i. if i==j:
4.a.i.1. increment *count* by one.
4.a.i.2. decrement j by one.
4.a.ii. if $w_i + w_j < M$
4.a.ii.1. increment *count* by one
4.a.ii.2. break j loop.
4.a.iii. else:
4.a.iii.1. increment *count* by one
4.a.iii.2. decrement j by 1 .
4.a.iv. Go to 4.a
5. return *count*.

Algorithm Correctness:
After step 1, we are trying to get best possible pair for each element while traversing from left to right.
Base Case: Let n = 1,
for i = 0, j = 0, count = 0, algorithm goes to 4.a.i, incrementing count by one, count=1;
Let n = 2:
for i=0, j=1, count = 0: algorithm goes to 4.a.ii:
If $w_i + w_j < M$,
i. then count becomes 1, and i becomes -1.
ii. returns 1
else :
i. count becomes 1 and j becomes 0.
ii. i = 0, j = 0, count = 1, algorithm goes to 4.a.i,
a. count becomes 2
Hence for base cases, n=1, n=2, algorithm works correctly.

Inductive Hypothesis: Assume the algorithm works for $n <= k$.
That is, for all arrays of size $n <= k$, we find best possible pair for every element and hence minimum number of boats required satisfying given conditions.

Inductive Step: We need to prove that algorithm also works for n = k+1

Case1: If the additional element, p is somewhere in the middle of the sorted array
Consider array be: $[a_1, a_2, .., a_q, ...p, a_l, ..., a_m, ..., a_{n-1}, a_n]$
Suppose, if pointer j reaches the element p first, then consider sub array: $[a_q, ...p]$
if pointer i reaches the element p first, then consider sub array, $[p, a_l, ..., a_m]$
In both cases, sub arrays are of size $<= k$.
From Inductive Hypothesis, we definitely know that this sub array provides minimum number of boats.
And the part of the array, we traversed till now has selected the best possible pair(from inductive hypothesis).
Hence providing the minimum number of boats for the whole k+1 array
and hence Inductive Step for case 1 is true

Case2: if p is on the right most side of the sorted array.
Let array be: $[a_1, a_2, ..., a_{n-1}, a_n, p]$
From base case and inductive hypothesis, we select best possible pair for element a_1
Acc to the algo, if $p + a_1$ is less than M, then one boat is selected.
Now, the subarray becomes, $[a_2, ..., a_n]$ which is of size $<= k$. and hence from IH, we get minimum number of boats for this array.
if $p + a_1$ is greater than M, there is no pair for p with which we can send p in the same boat(since a_1 is the least weight).
Hence according to the algo, we get an additional boat for p(according to the intuition even, we need to get additional boat).
now, we are left with sub array, $[a_1, ..., a_n]$ which is of size $<= k$ and hence we get minimum number of boats in this case.

Case 3: If p is on the left most side of the sorted array.
Let array be: $[p, a_1, ..., a_l, ..., a_{n-1}, a_n]$
for p, according to base case and inductive hypothesis, we get the best possible pair.
After p is done, we are left with sub array $[a_1, ...a_l]$ which of size less than k. and hence provides minimum number of boats required
Hence Inductive step is true for case 3 even.
Hence, we proved Inductive step to be true in all the cases.
Therefore, algorithm correctness is proved.

9

Algorithm:
1. Given set of arrays, $arrays = [arr_0, arr_1, ...arr_{n-1}]$
2. Initialise min_value = min of elements from arr_0.
3. Initialise max_value = max of elements from arr_0.
4. let max_i and min_i be maximum and minimum elements of i^{th} element in the $arrays$.
5. Initialise int i = 1, result_difference = 0
6. $result\_difference = max(result\_difference, (max\_i - min\_value), (max\_value - min\_i))$
7. Update $min\_value = min(min\_value, min\_i)$.
8. Update $max\_value = max(max\_value, max\_i)$.
9. if i == n-1 then Go to step 12.
10. Increment i by one.
11. Go to step 6.
12. return result_difference.

Algorithm Correctness:
Let's prove the algorithm by contradiction.
Let's assume the algorithm is incorrect
In step6, Since we assumed that the algorithm is incorrect,
$\exists p, q$ in the arrays such that maximum difference between any pair of elements is not covered by either $(max\_i - min\_value)$ or $(max\_value - min\_i)$.
implying, $result\_difference < max(p - min\_value, max\_value - q)$
If we reach step 12 without finding a contradiction, that means we iterated through all the arrays.
and hence, min_value and max_value are updated to cover the minimum and maximum values across all arrays.
But for any difference to be maximum, minimum value should be as minimum as possible and maximum value should be as maximum as possible.
Hence, while calculating, $(p - min\_value)$, min_value is anyways the minimum possible value. p should be the maximum possible value. Our algorithm takes max_i for every iteration. but since p is not equal to max_i p must be less than max_i.
Therefore $(p - min\_value)$ cannot be greater than $(max\_i - min\_value)$.
Similarly, $(max\_value - q)$ cannot be greater than $(max\_value - min\_i)$ since q cannot be less than min_i.
Hence, $max(p - min\_value, max\_value - q)$ cannot be greater than $result\_difference$.
hence contradicting our statement.

10

Algorithm can be divided into 2 parts:
Part1: Define Input.
1. Define all the cities as vertices(1 to N).
2. Draw an edge between $i -> 2 * i$ and $i -> i - 1$ with weights 1.
3. Add all vertices to the heap with costs infinity except for vertex a whose cost is 0.
Part2: Shortest Path algorithm.
Apply Dijkshtra.
1. Start from a. Add to solution tree.
2. $DecreaseKey$: Update heap costs of directly connected vertices from solution tree.
3. Get least cost vertex from heap.
4. If that vertex is city b return cost of city b.
5. $DeleteMin$: Else delete from heap and add to solution tree.
6. Go back to step 2.

Complexity of the above algorithm, is **O(n(log(n)))**.

We can reduce the complexity to **O(n)**:
1. Repeat Part1 of Algorithm 1 : Define Input.
Apply BFS.
2. Declare queue, Q 3. Start from a.
4. add all the vertices connected to vertex a to Q.
5. Dequeue() from queue and enqueue() all the unvisited vertices connected to the dequeued vertex to Q.
6. Repeat 5 until we reach vertex b.
7. If Q is empty() and vertex is not reached return -1

We can reduce the complexity to **O(log(n))**
1. initialise $numOfFlights = 0$

2. while $a != b$:
3. if($a > b$)
3.i. increment $numOfFlights$ by $a - b$
3.ii break from while loop.
4. else if b is odd:
4.i. if $b + 1 > N$: $numOfFlights = -1$;
4.ii.a. break the while loop
4.ii. else:
4.ii.a update b by $(b + 1)/2$ // b is always an integer
4.ii.b increment $numOfFlights$ by two.
5. else if b is even:
5.i. update b by b/2 // b is always an integer
5.ii. increment $numOfFlights$ by one.
6. return $numOfFlights$.