Soham Chatterjee
Email: soham.chatterjee@tifr.res.in
Course: Algorithms

Assignment - 1
Dept: STCS
Date: September 27, 2024

Problem 1 P2 (10 marks)

Show that $n-1$ comparisons are necessary and sufficient to find the minimum element in an unsorted array of n elements.

Solution: Suppose there is an algorithm which can find minimum element with less than $n-1$ comparisons. Let $A = (a_1, \ldots, a_n)$ be the unsorted array of n elements. Create a graph of vertices $[n]$ and put an edge between i and j if the elements a_i and a_j are compared. Since there are , $n-1$ comparisons in the graph we constructed there are at most $n-2$ edges. Hence the graph has at least 2 connected components. Let C_1 and C_2 be two connected components. No two elements one from each component is compared. Let the concluded minimal element from C_1 is x and the concluded minimal element from C_2 is y. Hence we can make any of them bigger than the other depending on the algorithm output to make the algorithm wrong. Hence at least $n-1$ comparisons are necessary to find the minimum element.

Now we can also find the minimum element of an unsorted array of n elements with $n-1$ comparisons by the following algorithm:

Algorithm 1: Find-Minimum

Input: Array $A = (a_1, \ldots, a_n)$
Output: Minimum element of A

```
1 begin
2     MinVal ⟵ A[1]
3     for i = 2, ..., n do
4         if MinVal > A[i] then
5             MinVal ⟵ A[i]
6     return MinVal
```

This algorithm uses only $n-1$ comparisons to find the minimum element of the array. Therefore $n-1$ comparisons are sufficient to find the minimum element in an unsorted array of n elements. ■

Problem 2 P3 (15 marks)

Show a comparison based algorithm for finding the minimum and maximum in an unsorted array of n elements using $\left\lfloor \frac{3n}{2} \right\rfloor - 2$ comparisons. Also show that $\left\lfloor \frac{3n}{2} \right\rfloor - 2$ comparisons are necessary to find the minimum and maximum.

Solution: First assume that all the elements are distinct. We will show the algorithm with which you only need to do $\left\lceil \frac{3n}{2} \right\rceil - 2$ comparisons.

So first we partition the elements into pairs. If n is odd then the last element will not in any groups. Hence there are total $\left\lfloor \frac{n}{2} \right\rfloor$ pairs. Now for each pair we compare then. Hence we have done now $\left\lfloor \frac{n}{2} \right\rfloor$ many comparisons. Now in each pair we have an element which was lesser than the other. So from each pair we take the lesser element and make a new set S. So S has $\left\lfloor \frac{n}{2} \right\rfloor$ elements. And from each pair we take the greater element and make another new set T. If n is odd then the last element which didn't take part in the pairs is added to both the sets S and T. So S and T both have now $\left\lceil \frac{n}{2} \right\rceil$ elements.

Now since S has the lesser elements the minimum element of the array is in S. And since T has the greater elements, the maximum element of the array is in T. So by the Problem 1: P2 we have an algorithm to find out the minimum element of S using $\left\lfloor \frac{n}{2} \right\rfloor - 1$ comparisons. And since finding maximum element of array is very similar as finding minimum element using the same algorithm but using less than instead of greater than we get the maximum element of T using $\left\lceil \frac{n}{2} \right\rceil - 1$ comparisons. Hence now we have the minimum and the maximum element of the array.

Algorithm 2: FIND-MINIMUM-MAXIMUM(A)

Input: Array $A = (a_1, \ldots, a_n)$
Output: Minimum and Maximum element of A

```
1  begin
2      n ⟵ A.length
3      if n == 1 then
4          return (A[1], A[1])
5      k ⟵ ⌊n/2⌋, S ⟵ ∅, T ⟵ ∅
6      for i = 1, ..., k do
7          if A[2i − 1] < A[2i] then
8              S ⟵ S ∪ {A[2i − 1]}, T ⟵ T ∪ {A[2i]}
9          else
10             S ⟵ S ∪ {A[2i]}, T ⟵ T ∪ {A[2i − 1]}
11     if n mod 2 == 1 then
12         S ⟵ S ∪ {A[n]}, T ⟵ T ∪ {A[n]}
13     Minval ⟵ ∞, Maxval ⟵ −∞
14     for i ∈ S do
15         if i = 1 then
16             Minval ⟵ S[1]
17         else if Minval > S[i] then
18             Minval ⟵ S[i]
19     for j ∈ T do
20         if j = 1 then
21             Maxval ⟵ T[1]
22         else if Maxval < T[i] then
23             Maxval ⟵ T[i]
```

Hence total comparisons needed to find the minimum and maximum element is

$$\left\lfloor \frac{n}{2} \right\rfloor + \left(\left\lceil \frac{n}{2} \right\rceil - 1 \right) + \left(\left\lceil \frac{n}{2} \right\rceil - 1 \right) = \left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{n}{2} \right\rceil - 2 = \left\lceil \frac{3n}{2} \right\rceil - 2$$

Hence this algorithm can find out the minimum and maximum using $\left\lceil \frac{3n}{2} \right\rceil - 2$ comparisons. Now we need to prove that at least this many comparisons are needed.

■

Problem 3 P4 (10 marks)

Let $G = (V, E)$ be a directed acyclic graph $G = (V, E)$. Additionally, you are given a nonnegative, integral weight w_e on each edge $e \in E$, and two special vertices $s, t \in V$. Give an algorithm to find a max-weight path from s to t.

Solution: We can solve this problem using dynamic programing. Let $dist(v)$ denotes the weight of maximum weight path from $s \rightsquigarrow v$ for any $v \in V$. $dist(s) = 0$. Now for any $v \in V$ we have the following relation:

$$dist(v) = \max_{u:(u,v)\in E} \{dist(u) + w(u, v)\}$$

Hence we have the following algorithm:

Algorithm 3: FIND-MAX-PATH(G, s, t, W)

Input: A directec acyclic graph $G = (V, E)$ with 2 vertices s, t and weights on edges W.
Output: Maximum weight path from $s \rightsquigarrow t$

```
1  begin
2      dist(s) ⟵ 0, s.parent ⟵ Null
3      for v ∈ V − {s} do
4          dist(v) ⟵ −∞
5          v.parent ⟵ Null
6      U ⟵ {s}
7      while U ≠ ∅ do
8          u ⟵ Extract first element of U
9          for (u, v) ∈ E do
10             if MaxPath[v] ≤ MaxPath[u] + w(u, v) then
11                 MaxPath[v] ⟵ MaxPath[u] + w(u, v)
12                 v.parent ⟵ u
13             U ⟵ U ∪ {v}
14     P ⟵ ∅, p ⟵ t
15     while True do
16         P ⟵ P ∪ {p}
17         if p.parent == Null then
18             Break
19         p ⟵ p.parent
20     P ⟵ Reverse(P)
21     return P
```

Time Complexity: The lines 2-3 takes $O(n)$ time and line 5 takes constant time. The while loop at line 7 picks a vertex and then goes through all its neighbors and adds them to U. So the while loop visits at most every vertex and and every edges. In each iteration it takes constant time. Therefore the while loop takes $O(|V| + |E|) = O(n^2)$ time. Now for the while loop at line 14 it can be at most n many iterations and in each iteration it takes constant time. The Reverse function also take linear time. Therefore the algorithm runs in $O(n^2)$ time or $O(|V| + |E|)$ time. ■

Problem 4 P5 (15 marks)

Given a matroid $(S, \mathcal{I})$, show that $(S, \mathcal{I}')$ is also a matroid, where $A \in \mathcal{I}'$ if $S \setminus A$ contains a maximal independent in $\mathcal{I}$.

Solution:

① Downward Closure: Let $A \in \mathcal{I}'$ and $B \subseteq A \implies S \setminus A \subseteq S \setminus B$. Since $A \in \mathcal{I}', \exists\ X \in \mathcal{I}$ such that X is a maximal independent set such that $X \subseteq S \setminus A$. Therefore we have $X \subseteq S \setminus A \subseteq S \setminus B$. Therefore $B \in \mathcal{I}'$. Hence $\mathcal{I}'$ follows the downward closure property.

② Exchange Property: Let $A, B \in \mathcal{I}'$ and $|B| < |A|$. Let $X, Y \in \mathcal{I}$ be maximal independent sets such that $X \subseteq S \setminus A$ and $Y \subseteq S \setminus B$. First we prove a lemma:

Lemma 1. *E, F are maximal independent sets in a matroid M. Then $\exists\ e \in E \setminus F$ and $f \in F \setminus E$ such that $(E \cup \{f\}) \setminus \{e\}$ and $(F \cup \{e\}) \setminus \{f\}$ are also maximal independent sets in M*

Proof: We have $|E| = |F|$. Let $e \in E \setminus F$. Then consider the set $E \setminus e$. Now since $|F| > |E \setminus \{e\}| = |E| - 1$ there exists $f \in F \setminus (E \setminus \{e\}) = F \setminus E$ such that $(E \cup \{f\}) \setminus \{e\}$ is an independent set. Since $|(E \cup \{f\}) \setminus \{e\}| = |E|$ it is a maximal independent set. Now we will show $(F \cup \{e\}) \setminus \{f\}$ is also a maximal independent set. Now it suffices to show that $(F \cup \{e\}) \setminus \{f\}$ is independent set since $|(F \cup \{e\}) \setminus \{f\}| = |F|$. Therefore if $(F \cup \{e\}) \setminus \{f\}$

is independent set then it is also a maximal independent set. □

■

Problem 5 P6 (15 marks)

In class, we showed that if $(S,\mathcal{I})$ is a matroid, then for any nonnegative weights w no the elements of S, the greedy algorithm obtains a maximum weight independent set. Show that this is only true if $(S,\mathcal{I})$ a matroid. That is, for a fixed downward-closed set system $(S,\mathcal{I})$, if the greedy algorithm obtains a maximum weight element of $\mathcal{I}$ for every assignment of nonnegative weights to elements of S, then $(S,\mathcal{I})$ is a matroid.

Solution: ■

Problem 6 P7 (10 marks)

Exercise 10.4-6 (on tree representations with pointers) from CLRS.

Solution: ■

Problem 7 P8 (10 marks)

Given a directed graph $G = (V,E)$ with weights on the edges, and which has a negative-weight directed cycle that is reachable from the source s, Give an efficient algorithm to list the vertices of such a cycle.

Solution: If the graph if there is a negative-weight cycle reachable from s then there is a vertex j reachable from s such that the shortest path from $j \rightsquigarrow j$ has negative weight. So we will use the Floyd Warshal Algorithm. But we will also keep track of the paths. We are assuming that G is given as an adjacency matrix.

Here we introduce a $[n+1]\times[n]\times[n]$ $D = \left(d_{i,j}^{(k)}\right)$ where $k \in [n+1]$. So $d_{i,j}^{(k)}$ is the weight of the shortest path from i to j using the vertices $[k]$. Naturally we have the relation:

$$d_{i,j}^{(k+1)} = \min\left\{d_{i,k+1}^{(k)} + d_{k+1,j}^{(k)},\ d_{i,j}^{(k)}\right\}$$

Here we also need to keep the path. So when the minimum path uses the $k+1$ vertex then we just concatenate the paths from $i \rightsquigarrow k+1 \rightsquigarrow j$ and otherwise the path is same as it was not using the vertex $k+1$. So here is the algorithm:

Algorithm 4: FIND-NEGATIVE-CYCLE(A, s, W)

Input: A directec graph $G = (V = [n], E)$ with source vertex s and weights on edges W with promise that G has a negative-weighted cycle reachable from s

Output: Find a negative-weighted cycle reachable from s

```
1  begin
2      Create a [n+1] × [n] × [n] array D = (d_{i,j}^{(k)}) for all k ∈ [n+1], i, j ∈ [n] with all entries (∞, Null)
3      for i, j ∈ [n] do
4          d_{i,j}^{(0)} = (A[i,j], [i,j])
5      for k = 1, ..., n do
6          for i = 1, ..., n do
7              for j = 1, ..., n do
8                  if d_{i,j}^{(k-1)}[1] > d_{i,k}^{(k-1)}[1] + d_{k,j}^{(k-1)}[1] then
9                      d_{i,j}^{(k)}[2] ⟵ d_{i,k}^{(k-1)}[2] ++ d_{k,j}^{(k-1)}[2]
10                 else
11                     d_{i,j}^{(k)}[2] ⟵ d_{i,j}^{(k-1)}[2]
12     Visited ⟵ {s}, U ⟵ {s}
13     Create a n length array VIS with all entries 0
14     VIS[s] = 1
15     while U ≠ ∅ do
16         u ⟵ Extract first element of U
17         for (u, v) ∈ E do
18             if VIS[v] == 0 then
19                 U ⟵ U ∪ {v}
20                 Visited ⟵ Visited ∪ {v}
21                 VIS[v] ⟵ 1
22     for v ∈ Visited do
23         if d_{i,i}^{(n)}[1] < 0 then
24             return d_{i,i}^{(n)}[2]
```

Time Complexity: The line 2 takes $O(n^3)$ time to create the array. In line 3 the for loop has n^2 iterations where each iteration takes constant time. In line 5-7 it takes total n^3 iterations and each iteration takes constant time. So the for loops in line 5-7 in total takes $O(n^3)$ time. In line 12-14 it takes linear time. In line 15 the while loop and the for loop at line 17 goes through each vertex and then also goes through each neighbor of it. Hence there are total $O(|V| + |E|) = O(n^2)$ iterations each of which takes constant time. In line 22 the for loop has $O(n)$ many iterations at most and each iteration it takes constant time. Hence the algorithm runs in $O(n^3)$ time. ■

Problem 8 P9 (15 marks)

Let us modify the "cut rule" (in the implementation of decrease-key operation for a Fibonacci heap) to cut a node x from its parent as soon as it loses its 3rd child. Recall that the rule that we studied in class was when a node loses its 2nd child. Can we still upper bound the maximum degree of a node of an n–node Fibonacci heap with $O(\log n)$.

Solution: ■

Problem 9 P10 (15 marks)

The following are Fibonacci-heap operations: *extract-min*$(\cdot)$, *decrease-key*$(\cdot,\cdot)$, and also *create-node*(x,k) which creates a node x in th root list with key value k. Show a sequence of these operations that results in a Fibonacci heap consisting of just one tree that is a linear chain of n nodes.

Solution: ■