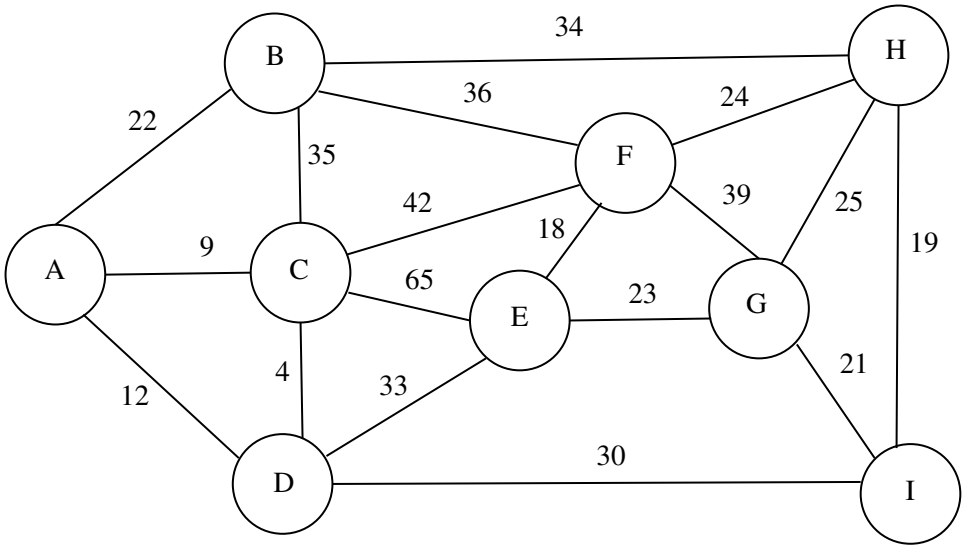
UNIVERSITY OF WASHINGTON, BOTHELL

CSS 549: ALGORITHM DESIGN AND ANALYSIS

PROBLEM SET 2

Sumit Hotchandani
Sharan Vaitheeswaran

1



(a) **Kruskal's Algorithm:** We will begin by sorting all the edges in increasing order of their weights.

1.) (C, D) = 4
2.) (A, C) = 9
3.) (A, D) = 12
4.) (E, F) = 18
5.) (H, I) = 19
6.) (G, I) = 21
7.) (A, B) = 22
8.) (E, G) = 23
9.) (F, H) = 24
10.) (G, H) = 25
11.) (D, I) = 30
12.) (D, E) = 33
13.) (B, H) = 34
14.) (B, C) = 35
15.) (B, F) = 36
16.) (F, G) = 39
17.) (C, F) = 42
18.) (C, E) = 65

Once, we have the sorted list of edges, we add the edges in order from the list as long as it doesn't form a cycle. e.g. edge (C, D) will be added first followed by edge(A, C), but the next edge (A, D) will not be added as it will form a cycle between vertices A, C and D. This process continues until $n-1$ edges have been added.

The edges added in order along with the MST formed are provided below:

1.) (C, D) = 4

2.) (A, C) = 9

3.) (E, F) = 18

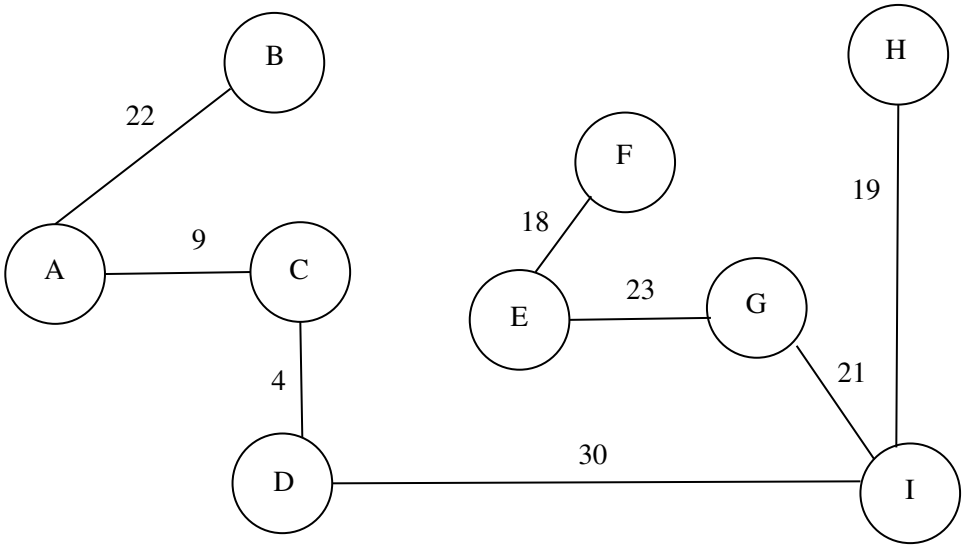
4.) (H, I) = 19

5.) (G, I) = 21

6.) (A, B) = 22

7.) (E, G) = 23

8.) (D, I) = 30



(b) **Prim's Algorithm :** For prim's algorithm we will maintain two sets of visited and unvisited nodes. Initially, all nodes are in the unvisited set.

As we are starting the algorithm from node A, we will add node to the visited set. We will now determine which vertex from A has the lowest weight to a vertex in the unvisited set, which is edge (A, C) = 9. Based on this vertex C is added to the visited set.

On each step going forward we will determine, which vertex from visited set to unvisited set has the lowest edge weight and add that edge to the MST and the corresponding vertex will be moved from the unvisited set to the visited set.

The edges added in order along with the MST formed are provided below:

1.) (A, C) = 9

2.) (C, D) = 4

3.) (A, B) = 22

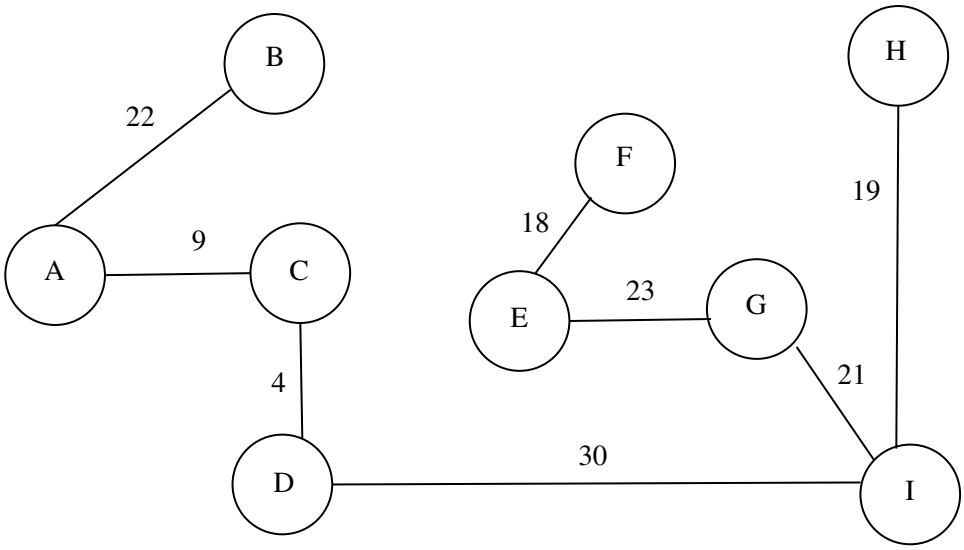
4.) (D, I) = 30

5.) (H, I) = 19

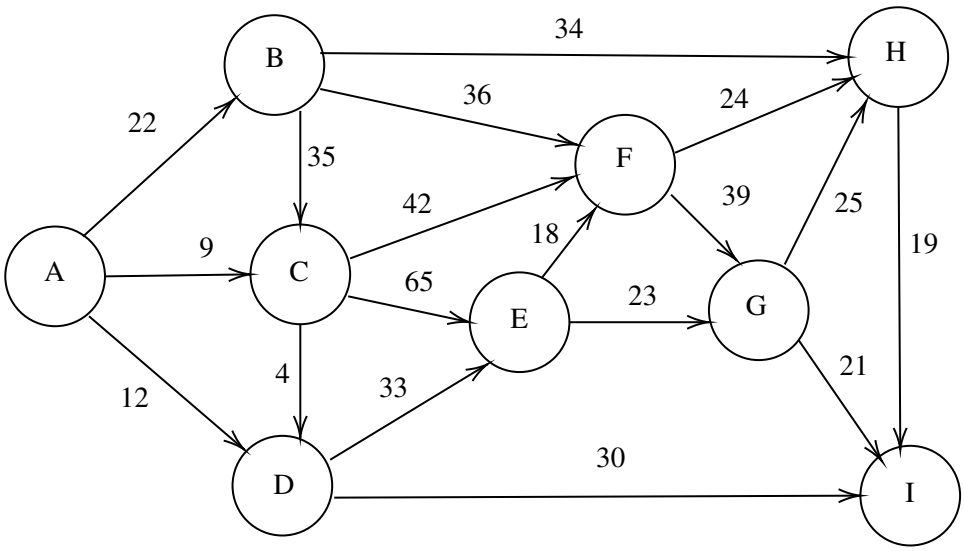
6.) (G, I) = 21

7.) (E, G) = 23

8.) (E, F) = 18



(c) **Minimum Cost Arborescence:** The original graph is converted to a directed graph, the edges are directed from the node lower in the alphabet to the node higher in the alphabet, resulting in the graph provided below.



Our root r for the arborescence is node A. For every vertex $v \neq r$, let $y(v)$ denote the minimum cost of any edge entering v.

Once we have $y(v)$, we need to calculate reduced cost of edge (u,v) as $c'(u,v) = c(u,v) - y(v) \geq 0$.

B	C	D	E	F	G
(A, B) = 22	**(A, C) = 9**	**(C, D) = 4**	**(D, E) = 33**	**(E, F) = 18**	**(E, G) = 23**
	(B, C) = 35	(A, D) = 12	(C, E) = 65	(B, F) = 46	(F, G) = 39
				(C, F) = 42	

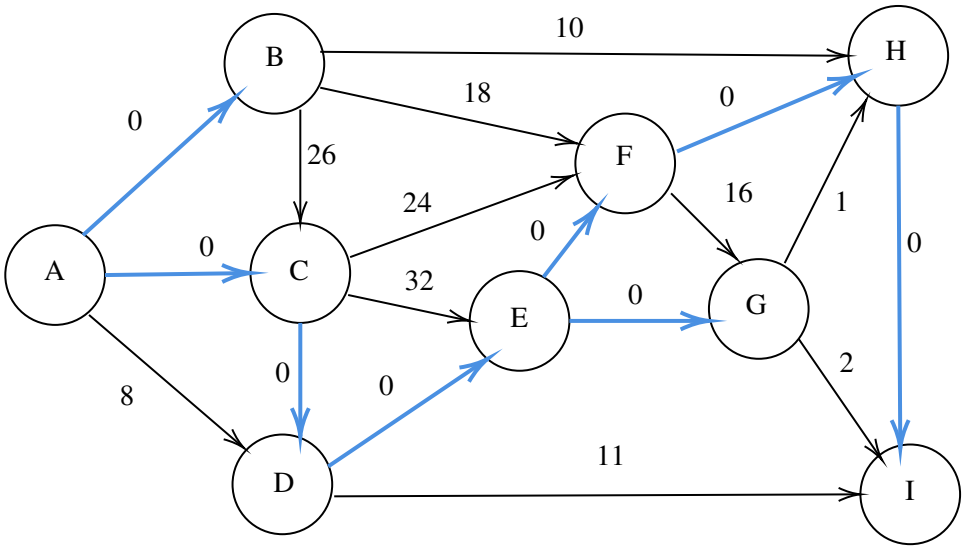
H	I
(F, H) = 24	**(H, I) = 19**
(G, H) = 25	(G, I) = 21
(B, H) = 34	(D, I) = 30

The edges highlighted in bold are the min cost edge $y(v)$ for each vertex v. We will then calculate reduced cost edges by subtracting $y(v)$ from each edge incoming to vertex v.

B	C	D	E	F	G
(A, B) = 0	(A, C) = 0	(C, D) = 0	(D, E) = 0	(E, F) = 0	(E, G) = 0
	(B, C) = 26	(A, D) = 8	(C, E) = 32	(B, F) = 18	(F, G) = 16
				(C, F) = 24	

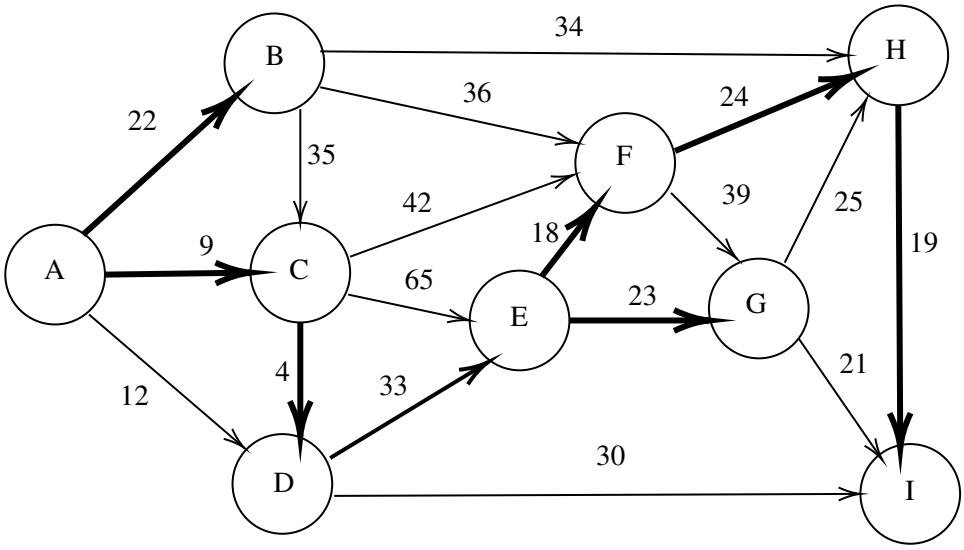
H	I
(F, H) = 0	(H, I) = 0
(G, H) = 1	(G, I) = 2
(B, H) = 10	(D, I) = 11

Replacing the costs of the edges with the reduced costs in the following graph.



In the next phase select all the 0-cost directed edges (highlighted in blue in the above graph) and check if there exists any cycle.

As clearly seen in the above graph, there is no directed cycle. In addition to the above there are exactly $n-1$ edges and for every vertex $v \neq r$, the indegree or the no. of incoming vertices = 1, thus we have obtained the min cost arborescence (bold edges) shown in the graph below



2

Master theorem. Let $a >= 1$, $b >= 2$ *and* $c > 0$ and suppose that $T(n)$ is a function on the non-negative integers that satisfies the recurrence $T(n) = a.T(\frac{n}{b}) + \theta(n^c)$ with $T(0) = 0$ and $T(1) = \theta(1)$ where $\frac{n}{b}$ means either $[\frac{n}{b}]$ or $[\frac{n}{b}]$. Then,

Case 1. If $c < log_b a$, then $T(n) = \theta(n^{log_b a})$.
Case 2. If $c = log_b a$, then $T(n) = \theta(n^c\ logn)$
Case 3. If $c > log_b a$, then $T(n) = \theta(n^c)$

(a) Performs 5 recursive calls on problems half the size of the input and performs $\theta(n^2)$ work per recursive call.

$$T(n) = 5.T(\frac{n}{2}) + \theta(n^2)$$

based on this recurrence relation $a = 5,\ b = 2,\ c = 2 \therefore a > b^c\ (5 > 2^2)$
using case 3 of Master's theorem $T(n) = \theta(n^{log_b a})$
Hence, $\boldsymbol{T(n) = \theta(n^{log_2 5})}$

(b) Performs 2 recursive calls on problems a quarter the size of the input and performs $\theta(n)$ work per recursive call.

$$T(n) = 2.T(\frac{n}{4}) + \theta(n)$$

based on this recurrence relation $a = 2,\ b = 4,\ c = 1 \therefore a < b^c\ (2 < 4^1)$
using case 1 of Master's theorem $T(n) = \theta(n^c)$
Hence, $\boldsymbol{T(n) = \theta(n)}$

(c) Performs 8 recursive calls on problems half the size of the input and performs $\theta(n^3)$ work per recursive call.

$$T(n) = 8.T(\frac{n}{2}) + \theta(n^3)$$

based on this recurrence relation $a = 8,\ b = 2,\ c = 3 \therefore a = b^c\ (8 = 2^3)$
using case 2 of Master's theorem $T(n) = \theta(n^c\ logn)$
Hence, $\boldsymbol{T(n) = \theta(n^3\ logn)}$

3

The divide and conquer algorithm to find the closest pair of points can be generalized to find the two closest pair of points as follows:

1. **Sorting the points:**
    - We can start by maintaining two lists, Px where the points are sorted by their x-coordinate and Py with the same points sorted by their y-coordinate.
    - This step will take $\boldsymbol{O(n\ logn)}$ time.
2. **Divide:**
    - Now we can divide the points in two halves using the median x-coordinate x_{mid}.
    - The first subarray contains points from $Px[0]$ to $Px[n/2]$ and the second subarray contains points from $Px[n/2+1]$ to $Px[n-1]$.
    - We will also divide points Py by comparing their x-coordinate with the median x-coordinate. All points with x-coordinates less than equal to the midpoint will be stored in the first subarray and points with x-coordinates greater than midpoint will be stored in the right subarray.

3. **Recursively finding smallest distances:**

   - Recursively find the two smallest distances in both subarrays. Let the distances be dl_1 and dl_2 (from left subarray) and dr_1 and dr_2 (from the right subarray).
   - Find the minimum of dl_1 and dr_1 and dl_2 and dr_2 as two minimum distances d_1 and d_2 such that $d_1 \leq d_2$.

4. **Combine:**

   - Create a list $strip$ which contains all points from Py whose x-coordinates are within distance d_2 of the median x-coordinate.
   - Find two smallest distances sd_1 and sd_2 between points in $strip$. This operation is $\boldsymbol{O(n)}$ as we need to check at the most 7 points (since points were already sorted by y-coordinate).

5. **Result:**

   - Compare d_1 with sd_1 and d_2 with sd_2 and select the minimum from both to get the two closest distances and the two closest pair of points corresponding to these distances.

Guarantee of correctness:

- The algorithm's correctness is established through a sound recursive approach and careful selection of minimum distances. In the recursive step, assuming correctness for left and right subarrays, the algorithm accurately identifies the two smallest distances, denoted as dl_1, dr_1, dl_2, and dr_2. The recursive assumption ensures the correct identification of the closest pairs within each subarray.
- Moving to the combine step there can be two cases:

  1. Let's assume points p_1, q_1 are the closest pair of points and p_2, q_2 are the second closest pair where $p_1 \; or \; q_1 \neq p_2 \; or \; q_2$. If p_1, q_1 and p_2, q_2 both lie in either left subarray or right subarray, but do not span across it i.e., to say either point in the pair does not lie in the $strip$, then in the combine step, calculating the minimum between pair of distances dl_1, dr_1 and dl_2, dr_2 will give us the two closest pair of points.
  2. In case where at least one point or one of the closest pair of points spanning across the subarrays, the strip array is formed with points within distance d_2 of the median x-coordinate. Within this strip, any pair of points has a distance at most d_2. The algorithm proceeds to find the smallest distances, sd_1 and sd_2, within the strip, correctly identifying the closest pair within this region as (p_s, q_s) with distance d_s.

The final step involves comparing the distances d_1 and d_2 with $sd1$ and $sd2$. By selecting the minimum from both comparisons, the algorithm guarantees the correct identification of the two closest distances. If d_1 is the minimum, then (p_1, q_1) is the correct closest pair, and if d_2 is the minimum, then (p_2, q_2) is the correct closest pair.

Time Complexity:

- Each function call performs 2 recursive calls where the problem is divided in half and the combine step of the finding the two closest pair of points is done in $O(n)$ time. Therefore the overall time complexity $T(n)$ can be represented as

$$T(n) = 2.T(\frac{n}{2}) + O(n)$$

where $a = 2,\ b = 2\ c = 1$ which satisfies case 2 of master theorem i.e., $a = b^c$
Therefore, overall time complexity $\boldsymbol{T(n) = O(n\ logn)}$

4

(a) **Recurrence Relation**

Let's denote $V(i)$ as the maximum value that can be achieved through day i. Then the recurrence relation can be expressed as follows:

$$V(i) = max(V(i-1) + s_i,\ V(i-2) + l_{i-1})$$

where,
s_i = the value of the short job on day i.
l_i = the value of the long job on day i.

The base cases will be

$$V(0) = 0$$
$$V(1) = s_i$$

(b) **Algorithm**

Algorithm 1 Algorithm to find max value of jobs

```
initialize an array V of size n+1, where n is the number of days
V[0] ← 0                                                    ▷ base case
V[1] ← s_i                                                  ▷ base case
for i = 2 to n step 1 do
    V[i] ← Max(V[i−1] + s_i, V[i−2] + l_{i−1})
end for
return V[n]
```

(c) **Time and Space Complexity**

- The algorithm iterates over each day once, hence the running time of the algorithm is $O(n)$, where n is the number of days.
- The memory usage is also $O(n)$, because we are storing a value for each day in the array V.
- Time and space complexity in big-theta notation is $\theta(n)$

5

(a) **Recurrence Relation**

Let's denote $P(i)$ as the maximum profit that can be achieved by selling i pieces of candy. Then the recurrence relation can be expressed as follows:

$$P(i) = max(P_j + P(i-j)) \text{ for } 1 \leq j \leq i$$

where P_j = the profit of selling j pieces of candy.

The base case is

$$P(0) = 0$$

(b) **Algorithm**

Algorithm 2 Algorithm to find max profit by selling candies

```
initialize an array P of size n+1, where n is the number of candies
P[0] ← 0                                                    ▷ base case
for i = 1 to n step 1 do
    max_profit ← −1
    for j = 1 to i step 1 do
        max_profit ← Max(max_profit, P[j] + P[i − j])
    end for
    P[i] ← max_profit
end for
return P[n]
```

(c) **Time and Space Complexity**

- The algorithm iterates over each candy once for each possible number of candies, hence the running time of the algorithm is $O(n^2)$, where n is the number of candies.

- The memory usage is also $O(n)$, because we are storing the profit for each possible number of candies in the array P.
- Time complexity in big-theta notation is $\theta(n^2)$ and space complexity is $\theta(n)$