Problem session 13

Dibran Dokter 1047390

December 12, 2019

13

13.1

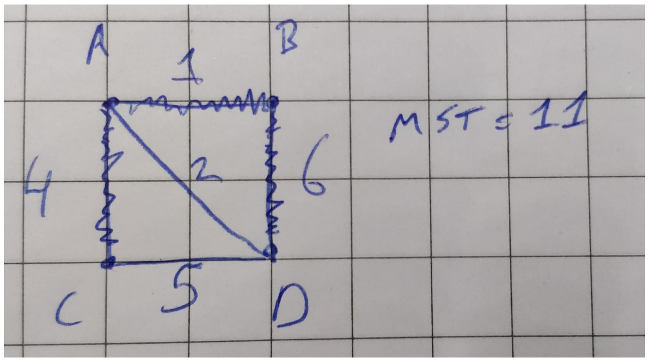
13.1.1

If there is an edge in the graph that is not in the MST and has a lower weight than one of the edges in the MST it would be added in the MST. Thus the MST should have at least one edge of minimum weight in it to form the MST.

Lets assume we have a MST M, and a graph G. If there exists an edge $E \in G$ that has the minimum weight of G that is not in M. Is M still a MST?
The answer is no, it is not a MST. Since if we replace the edge that connects the vertex at the end of E to the MST with edge E we get an MST that has a lower cost.
Thus, the MST should always have the edge with the minimum weight in the graph in it.

13.1.2

Looking at the example below, we see that the minimum weight is 1, on the edge from A to B. When we add this edge to the MST we can get the MST that contains edges (A,B), (A,C) and (B,D). Giving a cost of 11.
However this is not the MST of the graph. Thus having at least one edge of minimum weight does not always give the MST. Because it still depends on the other edges you choose.



13.2

13.2.1

Prim's algorithm:

We start with node a.

Iteration 0:
We add vertex a
Priority queue :

b	e	c

Predecessor and key :

Vertex	**Predecessor**	**Key**
A	a	a

Iteration 1:
We add vertex b
Priority queue:

e	d	c

Predecessor and key :

Vertex	**Predecessor**	**Key**
A	a	0
B	a	3

Iteration 2:
We add vertex e
Priority queue:

c	d

Predecessor and key :

Vertex	**Predecessor**	**Key**
A	a	0
B	a	3
E	b	2

Iteration 3:
We add vertex c
Priority queue:

d

Predecessor and key :

Vertex	**Predecessor**	**Key**
A	a	0
B	a	3
E	b	2
C	e	3
D	c	5

13.2.2

Kruskal's algorithm:

We start with vertex a:

Iteration 1:
We add vertex b with edge $a \rightarrow b$.
Iteration 2:
We add vertex e with edge $b \rightarrow e$.
Iteration 3:
We add vertex c with edge $e \rightarrow c$.
Iteration 4:
We add vertex d with edge $c \rightarrow d$.

13.2.3

We can sort an array using Prim's algorithm by converting the array to a graph where every vertex is connected to every other vertex. Thus we create the fully connected graph K_n where n is the number of values in the array. Then run Prim's algorithm and add the vertices to a list. This list is now sorted.

13.3

These algorithms both solve the problem since they select the same jobs. One selects the jobs from the beginning by finding the earliest finishing time and the other selects the jobs from the end by finding the latest starting time. They do this as long as the jobs are compatible with the jobs already selected.
They select the same jobs because if this is not the case there exists a job that has an earlier finishing time than starting time. And such a job cannot exist.

13.4

Sort the set of numbers in ascending order. Now we look at the numbers from lowest to highest. We take the lowest number and add a single unit. Then we remove all the numbers that this encompasses from the list. We do this until there are no numbers left in the list.

This algorithm is correct because if we do it in a different way we do not include all the numbers that are required. It gives the minimal number of intervals since if we remove an interval it no longer encompasses all the values.

This algorithm gives the time complexity $\mathcal{O}(nlogn)$ for sorting the list and in the worst case we need to create an interval for every value. This gives the complexity $\mathcal{O}(n)$. Which combined gives the complexity $\mathcal{O}(n + (n\ log\ n))$, which can be simplified to being $\mathcal{O}(n)$.

13.5

To do this we take the sub-sequence and look for its elements in the sequence. If we find all the elements of the sub-sequence in the sequence it is indeed a sub-sequence. If there are elements left over it is not a sub-sequence.

To do this we take the first element of the sub-sequence and walk through the sequence until we find the element. Then we take the next element and find it. We do this until we either have not more elements in the sub-sequence or we have run through the entire sequence. If there are no elements left in the sub-sequence the sub-sequence is correct. Otherwise it is not a valid sub-sequence of the given sequence.

This algorithm gives the time complexity of $\mathcal{O}(m + n)$ since in the worst case we need to run through the entire sub-sequence and the given sequence.

This algorithm is based on the fact that the sub-sequence is defined as the sequence that is obtained by deleting zero or more elements from the sequence, in order. Thus, if the sub-sequence is valid the elements should be in the sequence with zero or more elements in between, in order.