Introduction to algorithms Fall 2017

IFT2125-6001 TA: Maëlle Zimmermann

Demonstration 6

1

Question: Run the Prim algorithm on the following graph:

Solution: By running the algorithm, we get:

Iteration	(u, v)	B
0	-	{1}
1	(1,2)	{1,2}
2	(2,3)	{1,2,3}
3	(1,4)	{1,2,3,4}
4	(4,5)	{1,2,3,4,5}
5	(4.7)	{1,2,3,4,5,7}
6	(7.6)	{1,2,3,4,5,7,6}

Thus the underweight shaft of minimum weight is of weight 17.

2

Question: Show that the Prim algorithm can, like Kruskal's, be implemented using piles. Show that it then takes a time in Θ (alog n).

Solution: Let us first consider the Prim algorithm naively implemented without monceaux:

```
def Prim (V, E)
    def weight ((u, v, c)): return c
    F = sorted (E, key = weight)
    T = []
    Set B = (V [1])

    # as long as all vertices are not covered
    while len (B)! = len (V)
        for (u, v, _) in F:
            if (! u in B) = (v in B)
                break
        T.append ((u, v))
        B.update ([u, v])
    return T
```

In the worst case, this algorithm takes a time in O (an) (while execution loop exactly n-1 times and path of F in integer which contains a edges). But there is a better implementation. Here is the algorithm of Prim implemented with heaps:

```
def Prim_heap (V, E)
    if len (V) == 0:
        return []
    x = V [0] # current top
```



```
    T = [] # minimum partial tree
    B = Set (V [1]) # summits covered by T
    H = [] # empty heap

    # build neighbors of vertices
    neighbors = [[] for v in V]
    for (u, v, c) in E:
```

```
neighbors [u] .append ((v, c))
neighbors [v] .append ((u, c))

# calculation of the tree
while len (T) <len (V) - 1:
# sets all neighbors of x in the heap
    for (y, c) in neighboring [x]
        heappush (H, (c, (x, y)))

    # remove the edge to the minimum weight c
    (c, (u, v)) = heappop (H)

    # continues to withdraw until the edge passes through B and V \ B
    while (u in B) == (v in B):
        (c, (u, v)) = heappop (H)

    # update x, T and B
    x = u if u not in B else v
    T.append ((u, v))
    B.add (x)
return T
```

In a heap, the push and pop operations take a time in Θ (log k) and Θ (log k) respectively (where uuk is the number of elements in the pile). Let n = | V | and a = | E |, then:

- The loop that builds neighbors is executed a times so takes a long time in Θ (a).
- Each edge (u, v) is added at most 2 times in the heap either via u or via v. There are thus at most 2a push operations which each require a time of Θ (log a). So the total time of the additions is in Θ (alog a).
- Similarly, there are at most 2a pop operations that each require a time of Θ (log a). So the total withdrawal time is in Θ (alog a).
- The append and add operations are executed as many times as there are iterations of the algorithm, therefore at most n - 1 times.

The execution time of the algorithm is thus in Θ (alog a), which is also written Θ (alog n) for the same reasons as for the Kruskal algorithm.