CSCB63 A2

Saad Makrod

July 2022

Q1 Extracting Maximum Values From Maximum Heap Analysis

To get the largest value in a max heap you extract the first element and then perform heapify to keep the max heap structure. The complexity of this operation is $\theta(log(n))$. Thus to get the 3 largest keys you must perform this operation 3 times. So when extracting the 3 largest keys the complexity is $3log(n)$.

Thus to extract the 3 largest keys the complexity is $3log(n) \in \theta(log(n))$.

I can generalize this when trying to extract the i^{th} largest key. When extracting the i^{th} largest key I must extract the first element and then perform heapify i times. As a result the operation will take $i * complexity\ of\ extraction\ operation = i * log(n)$.

Thus to extract the i^{th} largest key the complexity is $ilog(n) \in \theta(log(n))$.

I can improve the complexity with the following algorithm: Create a new max heap, I will refer to it as S. Copy the max element in H into S. Then repeat the following $i-1$ times: perform the extract max operation on S and copy the children of that node from H and insert them into into S. After repeating this $i-1$ times, extract max on S one more time, this value will be the ith largest value in S.

This works since for each element in H its children are smaller than it. So when I extract max from S, the children of the original element in H are possibilities for the next largest value. Note that it is not a guarantee that they are the next largest since a heap only guarantees that children are smaller than the parent, there could be a larger node from another part of the heap. When the children are inserted, heapify will restructure the heap so that it still maintains the max heap structure. As a result, this algorithm is able to keep track of the ith largest value in the heap and instead of having to work with the whole heap it only works with a section of the heap which reduces the complexity.

The complexity of this algorithm is $\theta(ilog(i))$. This is because it is a constant operation to read a node and its children and since S is never larger that i elements it takes $log(i)$ to extract or insert an element in it. Since it repeats extraction/insertion i times the total complexity of the algorithm is $\theta(1 * i * log(i)) = \theta(ilog(i))$.

Thus to extract the 3 largest keys the complexity is $3log(n) \in \theta(log(n))$, to extract the i^{th} largest key the complexity is $ilog(n) \in \theta(log(n))$, and we can improve the complexity of the algorithm to $\theta(ilog(i))$.

Q2 Prove or Disprove: Given a graph G with n vertices such that for every $v \in V$, $deg(v) \geq \frac{n}{2}$, then G is one connected component

I will prove the statement by way of contradiction.

Assume this is not possible and G is not a single connected component. In other words there exists at least 2 vertices that are not connected to each other. Let these vertices be a and b.

It is given that both $deg(a) \geq \frac{n}{2}$ and $deg(b) \geq \frac{n}{2}$. I will first consider the connected component which contains the vertex a. Since we know that $deg(a) \geq \frac{n}{2}$ the connected component including a must have at least $\frac{n}{2} + 1$ vertices. We will call this connected component A.

Now we will consider the connected component of b. Note the $deg(b) \geq \frac{n}{2}$, so similarly to a, the connected component including b must have at least $\frac{n}{2} + 1$ vertices.

However this is not possible. Since G has n vertices and A contains at least $\frac{n}{2} + 1$ vertices then there are $n - (\frac{n}{2} + 1) = \frac{n}{2} - 1$ vertices which are not connected to a. As a result, if a and b are not connected then b

can only be connected to those $\frac{n}{2} - 1$ vertices. This implies that $deg(b) < \frac{n}{2}$, however we assumed that $deg(b) \geq \frac{n}{2}$ so we have come to a contradiction.

As a result the two vertices a and b must be connected. The same argument will follow through on any two vertices in G so the graph G is one single connected component.

QED

Q3 Prove or Disprove if the Following Algorithm Constructs a MST: Assume that we have n points on a plane and that the distance between two points is the ordinary Euclidean distance. Also, assume the distances between all pairs of points are distinct. We decide to sort all the points in order of x-coordinate. Assume we have distinct x-coordinates. Let $(s_1, s_2, \ldots, s_n)$ be the sorted sequence of points. For each such point $s_i, 2 \leq i \leq n$, connect s_i with its closest neighbour among $s_1, s_2, \ldots, s_{i-1}$

This algorithm cannot construct a minimum spanning tree. Consider a plane with the following 3 points $a = (1, 7)$, $b = (2, 1)$, and $c = (3, 5)$. The distance between the following points are as follows:

Distance between a and b: $\sqrt{37}$

Distance between b and c: $\sqrt{17}$

Distance between a and c: $2\sqrt{2}$

The distance between these points are unique, the points all have distinct x-coordinates, and n in this case is 3, so the algorithm above should apply to them. By the algorithm we have $s_1 = a$, $s_2 = b$ and $s_3 = c$.

When using the algorithm to construct a MST (s_2, s_1) and (s_3, s_1) are the edges chosen to construct the MST. This is because the closest vertex to s_2 which is before s_2 in the sequence (s_1, s_2, s_3) is s_1 and the closest vertex to s_3 which is before s_3 in the sequence (s_1, s_2, s_3) is s_1. Thus the weight of this tree is $2\sqrt{2} + \sqrt{37}$.

However, this is not the MST of these 3 points. It is easy to see that the following edges construct the real MST: (s_2, s_3) and (s_3, s_1). This tree has a weight of $2\sqrt{2} + \sqrt{17}$. Clearly $2\sqrt{2} + \sqrt{17} < 2\sqrt{2} + \sqrt{37}$ so the algorithm proposed did not construct a MST.

Thus this counter example proves that the algorithm proposed cannot always construct a MST.

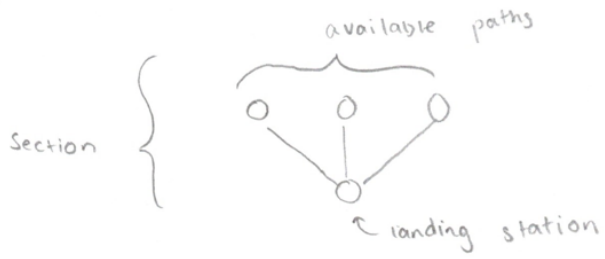
QED

Q4 Create a graph such that for every natural number N, there is an undirected graph of $cn + k$ vertices such that for some pair of vertices s and t in the graph, there are 3^n shortest paths from s to t

I need to make a graph such that the number of shortest paths from s to t increases by 3^n where $n \in N$ and $cn + k$ is the number of vertices. To do this I will keep s and t as two "end-points" in my graph and I will build a "body" that increases the number of shortest paths between the two endpoints as a function of 3^n. As a result I will let $k = 2$ for the two endpoints.

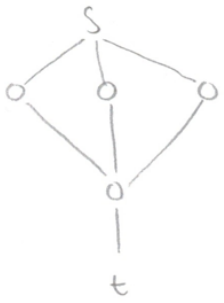
Now I need to decide on a value for c, to do this I must decide how I want to build the "body" of my graph. I know I want the number of shortest paths to increase as a function of 3^n. Consider the idea of a "landing station". The idea is to funnel all the paths to one specific vertex and from that vertex branch out again to next set of possible paths. As a result, when we move from one "landing station" to the next the paths multiply by the number of available paths. I know I want the paths to multiply by a factor of 3 so I will let the number of available paths be 3. So for each section of the "body" I will need one vertex for a "landing station" and 3 for the available paths. As a result I will let $c = 4$.

The idea of the design can be seen below:



Now I know that $c = 4$ and $k = 2$, all that is left is to implement the graph. I will draw a series of graphs to get the idea across.

Let $n = 1$,

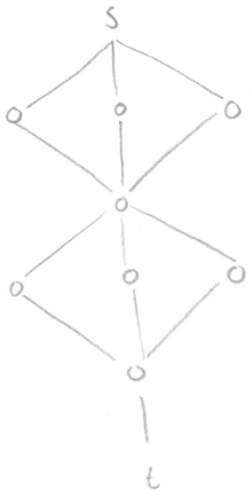


$n = 1$

$cnk = 4 \cdot 1 + 2 = 6$ vertices

$3^1 = 3$ shortest paths

Let $n = 2$,

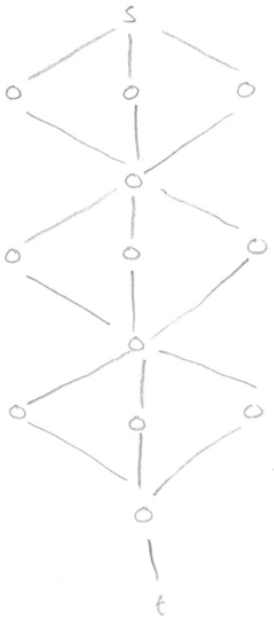


$n = 2$

$cnk = 4 \cdot 2 + 2 = 10$ vertices

$3^2 = 9$ shortest paths

Let $n = 3$,

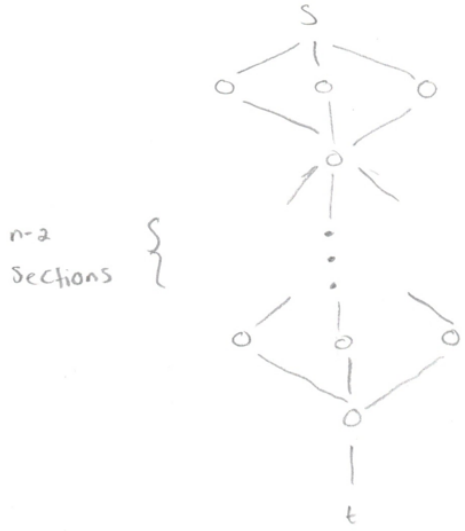


n = 3

Cnt K = 4·3+2 = 14 vertices

3^3 = 27 shortest paths

Let $n = n$,



n = n

Cnt K vertices

3^n Shortest paths

Thus for each 4 vertices added the number of shortest paths multiplies by 3. As a result this is a graph such that for every natural number N, there is an undirected graph of $4n + 2$ vertices such that for some pair of vertices s and t in the graph, there are 3^n shortest paths from s to t.

As a result this design satisfies the requirements.

Q5 Suppose we have a directed graph $G = (V, E)$ and each edge represented by (u, v) has an associated value $d(u, v)$, which is a real number and has range $0 \leq d(u, v) \leq 1$ that represents the dependability of a communication channel from vertex u to v. We interpret $d(u, v)$ as the probability that the channel from u to v will not fail, and we also assume that these probabilities are independent of each other. Based on this information, give an efficient algorithm that finds the most dependable/reliable path between any two given vertices

I know the following:

- Each edge in $G = (V, E)$ has an associated value $d(u, v)$ where u and v are two connected vertices in G

- $d(u, v)$ is a real number and has range $0 \leq d(u, v) \leq 1$
- $d(u, v)$ is the probability that the channel from u to v will not fail, and these probabilities are independent of each other

Since $d(u, v)$ is the probability that the channel from u to v will not fail, the higher it is the more likely a move from u to v is successful. Furthermore, since the probabilities are independent then if I want find the probability that a path from vertex a to b will not fail, I can form the product of the $d(u, v)$ values of the vertices in the path. Thus I want to maximize the product.

I can express the product as follows:

Let a and b be vertices in the directed graph G, the most reliable path from a to b is the path P such that the following product is maximized: $\prod_{(u,v)\in P} d(u, v)$. Where (u, v) is an edge in the path P.

In other words the most reliable path is the one which has the highest probability of being successful.

I want to convert this to a shortest path problem so that I can apply a shortest path algorithm to it. To do this I must choose how to define the weights of the edges in the graph G.

I know I want to maximize $\prod_{(u,v)\in P} d(u, v)$. To get closer to a shortest path problem I need to turn the product into a sum. The best way to do this is to *log* the product and apply *log* rules. I can do this since *log* is a monotonic increasing function so maximizing an expression is equivalent to maximizing the *log* of the expression.

Thus maximizing $\prod_{(u,v)\in P} d(u, v) \iff$ maximizing $log(\prod_{(u,v)\in P} d(u, v))$.

Since $\prod_{(u,v)\in P} d(u, v)$ is a product, by using *log* rules we can split the product into the sum as follows $log(\prod_{(u,v)\in P} d(u, v)) = \sum_{(u,v)\in P} log(d(u, v))$. So maximizing $\prod_{(u,v)\in P} d(u, v) \iff$ maximizing $\sum_{(u,v)\in P} log(d(u, v))$.

At this point I am close to a shortest path problem since now I have a sum. However, I am still trying to maximize the sum. If I want a shortest path problem I want to minimize the sum not maximize it.

To have a sum which I want to minimize I can change the $log(d(u, v))$ to $-log(d(u, v))$, since attempting to maximize a function is the same as attempting to minimize its negative. Thus maximizing $\sum_{(u,v)\in P} log(d(u, v)) \iff$ minimizing $\sum_{(u,v)\in P} -log(d(u, v))$ and maximizing $\prod_{(u,v)\in P} d(u, v) \iff$ minimizing $\sum_{(u,v)\in P} -log(d(u, v))$.

Consider the relation between the mapping of $d(u, v)$ and $-log(d(u, v))$. As $d(u, v)$ approaches 1, $-log(d(u, v))$ approaches 0 and as $d(u, v)$ approaches 0, $-log(d(u, v))$ approaches ∞. Thus this expression can be used in a shortest path problem since as $d(u, v)$ increases, $-log(d(u, v))$ decreases but remains non-negative. However $-log(0)$ is undefined. I will set weight of an edge with $d(u, v) = 0$ to be ∞ to make this work in a shortest path problem.

Thus I will define the weight of the edges in G by the function below:

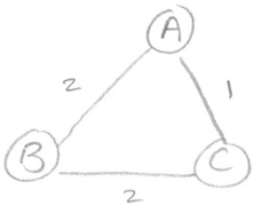
$$w(u, v) = \begin{cases} \infty & d(u, v) = 0 \\ -log(d(u, v)) & 0 < d(u, v) \leq 1 \end{cases}$$

Now I can use a shortest path algorithm to solve for the most reliable path since as stated earlier if $w(u, v) = \begin{cases} \infty & d(u, v) = 0 \\ -log(d(u, v)) & 0 < d(u, v) \leq 1 \end{cases}$ then maximizing $\prod_{(u,v)\in P} d(u, v) \iff$ minimizing $\sum_{(u,v)\in P} w(u, v)$ and I know that maximizing $\prod_{(u,v)\in P} d(u, v)$ gives the most reliable path.

Thus, consider using Dijkstra's Algorithm on G with the weights of the edges as follows, $w(u, v) = \begin{cases} \infty & d(u, v) = 0 \\ -log(d(u, v)) & 0 < d(u, v) \leq 1 \end{cases}$. Since Dijkstra's Algorithm is an efficient method to solving for the shortest path it will give the most reliable path between two vertices with efficient results.

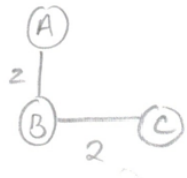
Q6 Prove or Disprove: Consider an undirected graph $G = (V, E)$ with non-distinct, non-negative edge weights. If the edge weights are not distinct, it is possible to have more than one MST. Suppose we have a spanning tree $T \subset E$ with the guarantee that for every $e \in T$, e belongs to some minimum-cost spanning tree in G. Can we conclude that T itself must be a minimum-cost spanning tree in G?

I will disprove the statement with a counter example. Consider the graph below, I will refer to it as G,



G has the following two MSTs,

From the figure above it is easy to see that all the edges in G belong to some MST of G. As a result, let $T = \{(A, B), (B, C)\}$. This is a valid choice of T since all the edges in T belong to a MST of G. However if we let T form a spanning tree of G,



It is clear that T is not an MST since the weight of T is 4 and the weight of a MST of G is 3.Therefore this counter example shows that we cannot conclude that T is a MST of G.

QED

Q7 We can model the build process of an object as a directed graph. For example, suppose we are building a house. The walls can't be painted until the drywall is installed which cannot happen until after the studs are built. Suppose we model the process with a vertex representing the components of the object and a directed edge from a to b if a must be completed before b can be completed. Notice that if this directed graph has a cycle, then there is no way to construct the object.

a) Give an algorithm to determine whether a directed graph has a cycle. What should your complexity be?

Consider a DFS, we know that whenever a DFS has a backedge then the graph has a cycle. The algorithm can be simple, if the DFS discovers a backedge return true to indicate that a cycle was found otherwise if the DFS completes and no backedge was found return false to indicate a cycle does not exist.

Since the algorithm is a DFS, in the worst case it will traverse the whole graph. As a result the complexity of this algorithm is the same as it is for a regular DFS. The complexity will be $\theta(|V| + |E|)$ where $|V|$

indicates the number of vertices and $|E|$ indicates the number of edges.

b) Using DFS, construct an algorithm that either returns a valid ordering of the vertices to build the object or a cycle confirming no such ordering exists. Again, what should your complexity be?

I will modify the regular DFS by adding two more Stacks to the algorithm. The algorithm itself will return one of the Stacks upon completion. One Stack will keep track of the visited vertices (call it Visit_Stack) and the other will keep track of the finished vertices (call it Finish_Stack).

The algorithm is as follows: Carry out a regular DFS. When a vertex in the DFS is finished push the vertex onto the Finish_Stack and as vertices are visited place them on the Visit_Stack. If the DFS discovers a backedge then a cycle has occurred, in this case the algorithm will push the vertex that it found has already been visited onto the Visit_Stack and return the Visit_Stack. If no backedge is discovered then the algorithm will just return the Finish_Stack after the DFS is finished.

Since a Stack is last in first out this means the vertex which finished last will be at the top of the Finish_Stack. To be more specific the Finish_Stack arranges the vertices in descending order of their finish time. The desired sequence has the vertices arranged this way since vertices that finish last are the same vertices that must be used when starting the build process. As a result the Stack returned is either a valid ordering of the vertices to build the object (the Finish_Stack) or a cycle confirming no such ordering exists (the Visit_Stack). I do not need to worry about the ordering in the Visit_Stack since all I need to know is if the same vertex appears twice in the stack to know if it is a cycle. Thus this algorithm performs the desired task. **Note that the pseudocode for the algorithm can be found at the end of the answer**.

Pushing an item onto the Stack is a constant time operation since we only need to update a pointer. In the worst case we must perform this operation $|V|$ times since we push all the vertices in the graph onto the Stack. Thus in total the Stack will add $|V|$ to the complexity of the operation. The DFS complexity is $\theta(|V| + |E|)$, so if we add the two Stacks in then we get the total complexity to be $\theta(3|V| + |E|)$. As a result the complexity of the algorithm is $\theta(3|V| + |E|)$.

Pseudocode for Algorithm:

Note that this is a modified version of the DFS algorithm found on the lecture slides.

```
Stack DFS(G=(V,E),s)
    for v in V: // initialize the arrays
        state[v] = not_visited; d[v] = infinity
        f[v] = infinity; p[v] = NULL
    new stack S
    new stack Finish_Stack //keep track of way to build model
    new stack Visit_Stack //keep track of visited vertices
    time = 0
    state[s] = visited; d[s] = time; p[s] = NULL
    Visit_Stack.push(s) //s has been visited so push
    S.push(NULL)
    for edge (s,v) in E:
        S.push(s,v)
    while not S.is_empty():
        (u,v) = S.pop()
        if (v == NULL): // Done with u
            time += 1
```

```
    f[u] = time
    state[u] = finished
    Finish_Stack.push(u) //finished so push onto the finish stack
else if (state[v] == not_visited):
    Visit_Stack.push(v) //push since visited
    state[v] = visited
    time += 1
    d[v] = time
    p[v] = u
    S.push((v,NULL)) // end of v's neighbours
    for edge (v,w) in E:
        S.push((v,w))
else if (state[v] == visited): //we have a backedge
    Visit_Stack.push(v) //push node where backedge was reached
    return Visit_Stack //cycle returned
return Finish_Stack //object plan returned
```