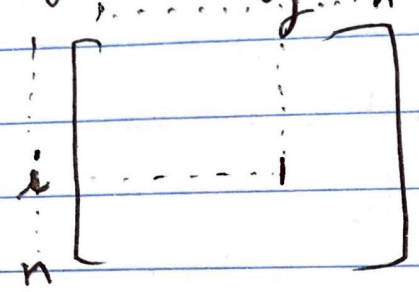
# Graph algorithms

## Adjacency matrix



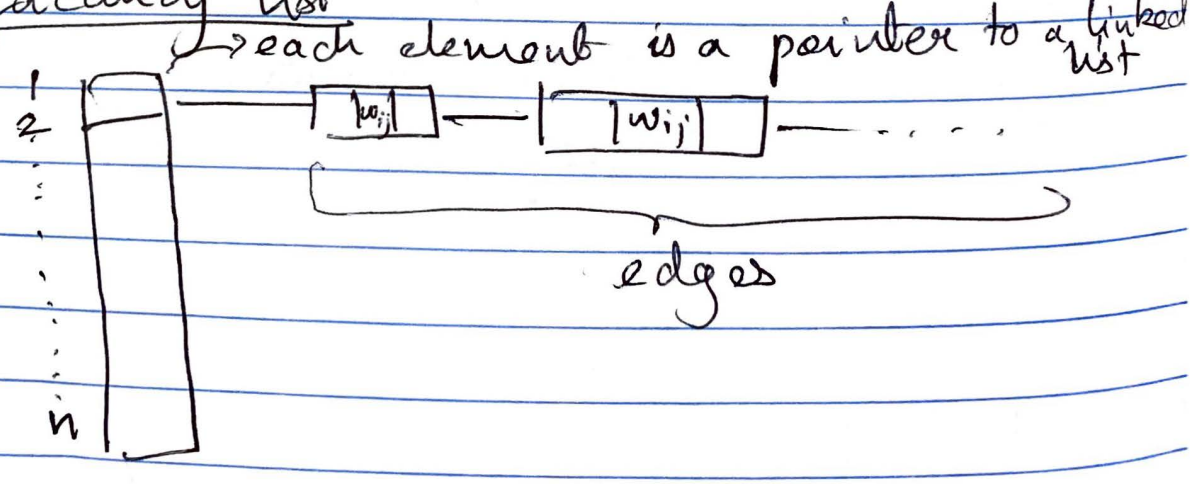

$n \times n$

there is an edge $[i,j]$ otherwise its a 0

If the graph has 0s & 1's its not weighted.

If there is a real no., then its a weighted edge.

## Adjacency list



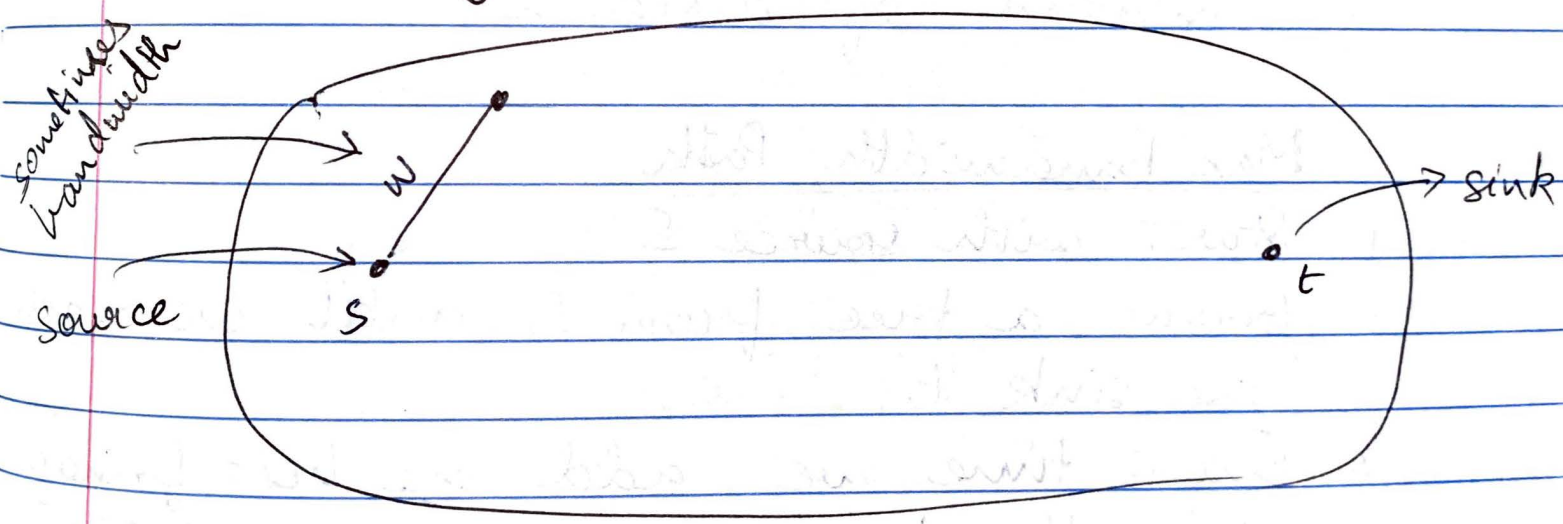

- Space complexity for adjacency matrix is $O(n^2)$

- Complexity for adjacency list is

$$O(n + m)$$

$m$ ↓ # of edges

When $n$ is larger, we prefer an adjacency list.




Band width of a pass = smallest edge along the path

## Dijkstra's Algorithm

* Maximum Bandwidth path (MBP)
Given a weighted graph G and two vertices s and t, find a s-t path of max bandwidth.

To solve the MBP problem we use Dijkstra's algo he we use bandwidth as the measure instead of distance.

Max Bandwidth Path

1 Start with source S
2 Grow a tree from S until we reach the sink t.
3 Each time we add the best fringe into the tree.

fringe → vertexes that are one step away from the tree.

status[n] - intree / fringes / unseen

wt[n] → records the measure of fringe
(weight)

wt[v] is the bandwidth of the path from s to v

dad[v] :- dad[v] is the father of v in the tree.

## Dijkstra's algorithm (G, s, t)

```
1  for (v=1; v<=n; v++)
       status[v] = unseen;
2  status[s] = intree;
3  for (each edge [s, w])
       status[w] = fringe
       wt[w] = weight (s, w)
       dad[w] = s
4  while (there are fringes)        → No more than n times
       pick the best fringe v; // i.e with max wt[v]
       status[v] = intree;
       for (each edge [v,w])        ← n times so O(n)
           if (status[w] == unseen)
               status[w] = fringe
               dad[w] = v
               wt[w] = min(wt[v], weight(v,w))
```

O(n) for lines 1–3

O(n) → because the status is already intree, we just have to traverse from s to t

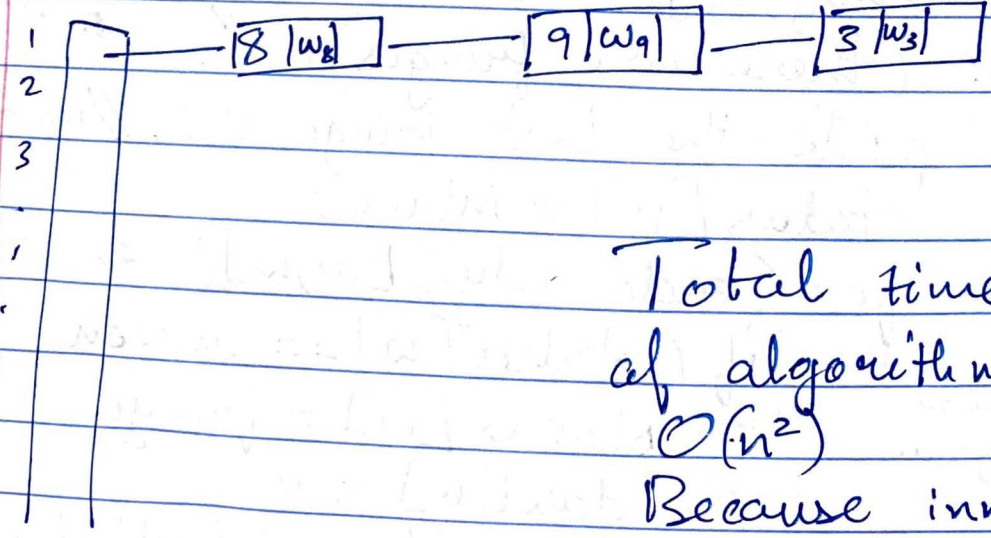
```
else (status[w] == fringe
                && wt[w] <
                min(wt[v], weight(v,w
    dad[w] = v;
    wt[w] = min(wt[v], weight(v,w
```

5 out [ dad [n], wt[n])

For this algorithm adjascency matrix is NOT good. Cuz we have to go through each row to find edges of the vertic We use an adjasceney list for this algorithm.

Not all vertices will have many edges; so adj matrix will have a lot of zeros which is a waste.

adj list




Total time complexity of algorithm is $O(n^2)$

Because inner loop (inside while loop) runs n times and outer loop runs n times.

→ First state the theorem
→ Then prove.

## The correctness

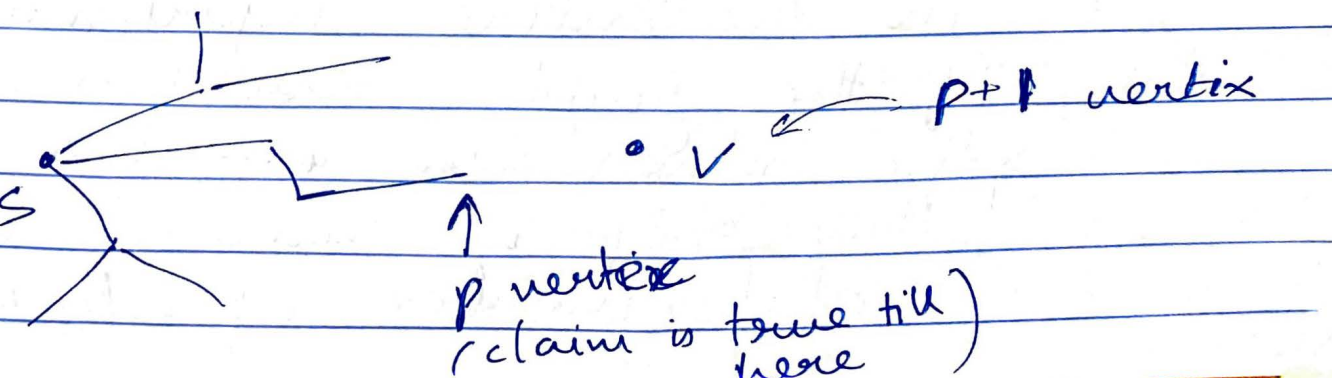
Once a vertex $v$ is added to the tree, then the path in the tree from $s$ to $v$ is the (globally) max bandwidth pass from $s$ to $v$

H.W Proof by induction on the number of vertices in the tree

P ← * For $p = 1 \Rightarrow s$ is the only vertex

(Be careful while doing this for shortest path. Distance from $s$ to $s$ may not be 0 if weights are ~~the~~ negative  )

* We assume ~~general~~ claim for general $p$
* Consider $p+1$ vertices in the tree
  → Let $v$ be the $(p+1)$ st tree vertix



can be $O(\log n)$ if you use a 2-3 tree

$v = \text{Max}(F)$ → 2-3 tree

Dijkstra (For MBP)

1-3 Initialization ⟶ $O(n)$ time → can be $O(n \log n)$ if we use 2-3 trees

4 while (there are fringes) ⟶ $n-1$ times

4.1 pick the best fringe $v$ ⟶ $O(n)$ time
status $(v)$ = in tree

4.2 for (each edge $[v, w]$) ⟶ $O(n)$ time
if (status of $w$ = = unseen)
status $[w]$ = fringe, dad $[w] = v$
$wt[w] = \min\{wt[v], \text{weight}(v,w)\}$ - -
else if (status $[w]$ = = fringe &&
$wt[w] < \min\{wt(v), \text{weight}(v,$
dad $(w) = v$
$wt(w) = \min\{wt[v], \text{weight}($

5 output (dad $[u]$)

time $= O(n^2)$ w/o 2-3 trees

How many times is step 4.2 executed in the entire algorithm?

Ans If the graph is directed, then the ~~step runs~~ number of times the step is run is equal to total number of edges. If

edges are undirected, each edge is counted at most 2 times

directed $\leq m$

undirected $\leq 2m$ where m is the number of edges.

∴ total time executed in step 4.2 = $O(m)$

Step 4.1 can be done using 2-3 tree to get time $O(\log n)$

→ v = Max (F)
Delete (F, v)
] $O(\log n)$
F = 2-3 tree for fringes

Step 4.2 can also make use of 2-3 tree

→ Insert (F, w)
→ Delete (F, w)
→ Insert (F, w)

Then step 4.2 becomes $O(m \log n)$ ~~which is also the total~~

↓

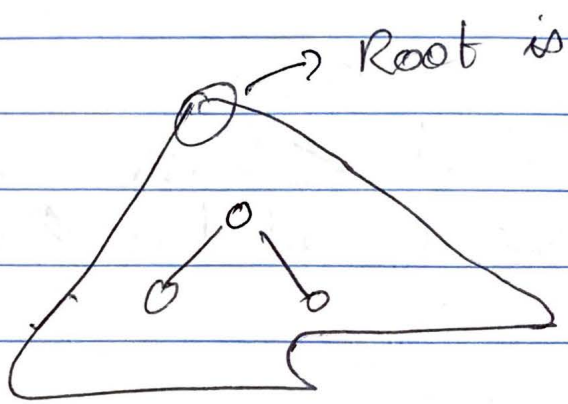
No. of edges.

∴ total time becomes

$O(n \log n + m \log n) = O((n+m) \log n)$

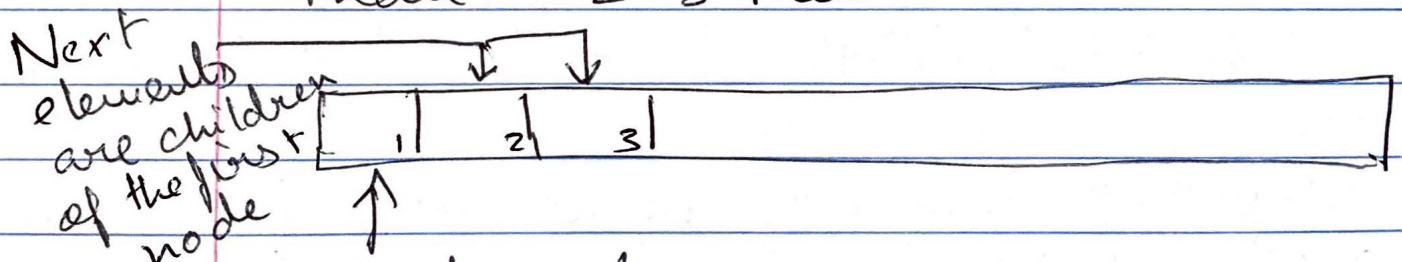
2-3 tree for purges F

Instead of using a 2-3 tree for purges, we use a heap. (max heap)




parent is always larger than ~~children~~ children

Max heap is a simpler data structure than 2-3 tree




Children index of a parent node indexed at $n = 2n, 2n+1$ (if indexing starts at 1)

Parent of a node = $\left\lfloor \frac{n}{2} \right\rfloor$

floor func →

If we use a max heap

fixHeap(H, i)
while (H[i] < max(H[2i], H[2i+1]))   (children of H[i])
    swap(H[i] w/ larger child)
    and make i this child.
& H[i] is not a leaf

If we use a max heap,
Step 4.1 in the algorithm becomes
    v = H[1]  → root
    H[i] = H[end--]; fixHeap(H, 1)

Dijkstras is a "single" source ~~shortest~~ path, source s is important, t is int.

Dijkstras

1.3 initialization

4 while (there are fringes)

v = largest fringe → if we use a max heap, just use the first element

status[v] = in tree

for (each edge [v, w])

if (status[w] == unseen)

status[w] = fringe, dad[w] = v

wt[w] = min{wt[v], weight[~~w,v~~ v,w]}

else if (status[w] == fringe &

wt[w] < min(wt[v], weight[v,w])

dad[w] = v

wt[w] = min{wt[v], weight[~~w,w~~ v,w]}

Dijkstras (using Max heap) $O((n+m)\log n)$

1-3 initialization

4 while (there are fringes)

v = H[1] , H[1] = H[end--]

status[v] = intree ; fixHeap(H, 1) *

for(each edge[v, w])

if (status[w] == unseen)

status[w] = fringe , dad[w] = v.

else if (status[w] == fringe & wt[w] < min(wt[v], weight[v, w]))

wt[w] = min{wt[v], weight[v, w]}

insert[H, w]

fixHeap[H, w]

fixHeap[H, w]

* fixHeap(H, i)

1 while (i is smaller than some of the children)

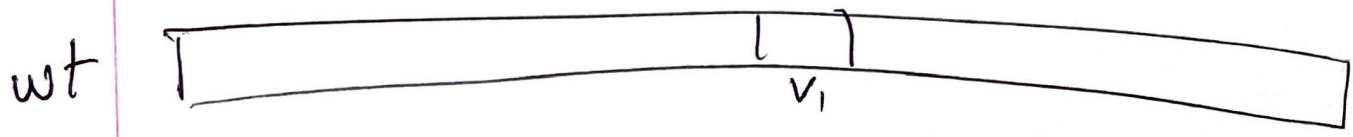
Swap i and the large child.

Let i be that child.

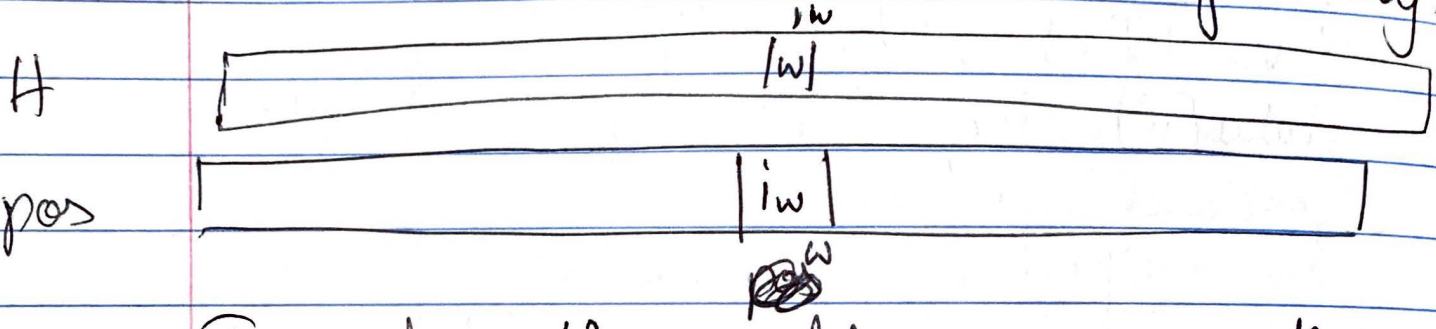
2 while (i is larger than its parent)

Swap i and its parent.

Let i be its parent.




While inserting or deleting, we need the position or index of vertex w in the Heap array.




To solve the problem use another array. to hold the index of vertex w.

$$pos[wt[H[i]]] = i$$

In heap, the no. of leaves is more than half of no. of non leaves.

Heap doesn't support searching.

# Alternate way to solve M.B.P (only M.B.P and not shortest path)

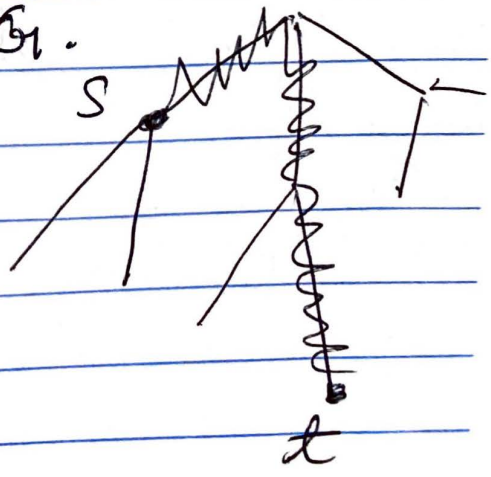
1. Construct a max spanning tree T.
2. Output the s-t path in tree T

For a spanning tree, the path from one node to another is always unique

Spanning trees work only on undirected graphs.

✗ Conditions for a tree (Also the definition according to Tianer chen)
- acyclic
- connected.

Spanning tree of a graph G is a subgraph of G that is a tree and contains all vertices of G.



Max spanning tree :- Given a weighted and undirected graph G, find a spanning tree of the largest weight

~~Algorithm for max spanning tree~~

Using Dijkstras

```
1,3 initialize
4   while (there are fringes)
    {                          }

    for (each edge[v, w])
        if (
        wt[w] = weight[v, w]

    else if (status[w] == fringe &
             wt[w] < weight[v, w])

        wt[v] = weight[v, w]
```