Graph Traversal Algorithms

→ Breadth-First Search (BFS)

Given a graph G(V,E) & source vertex s, BFS 'discovers' every vertex reachable from s

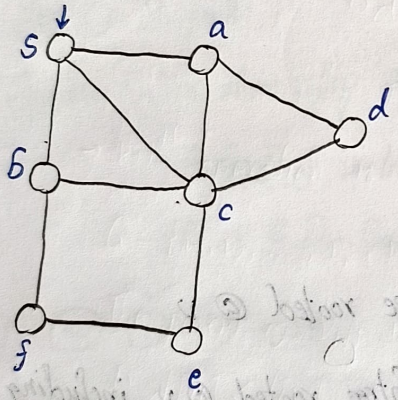
BFS colours nodes using 3 colours

↳ White: undiscovered node

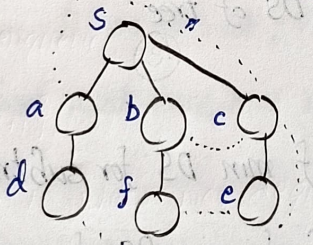
↳ Grey: Discovered node with white neighbours

↳ Black: Discovered node with neighbours either black or grey

Eg: Graph



Traversal : s, a, b, c, d, f, e



Traversal tree

* Edges in a traversal tree are tree edges
* Non-tree edges can connect nodes who are in the same level or one level apart

↳ If the edge connects nodes 2 levels apart, the bottom node must have been discovered ~~before~~ by the top node itself making the level difference as 1.

Algorithm:

```
BFS(G, s) {:
    for each vertex u in V(G) - {s} {:
        colour(u) = white   → colour of vertex
        d[u] = ∞            → distance from s
        π[u] = NIL          → parent node in traversal tree
    colour(s) = grey
    d[s] = 0
    π[s] = NIL
    Q = ϕ                   → queue of grey nodes.
    Enqueue(Q, s)
    while Q ≠ ϕ :
        u = ~~dequeue~~ dequeue(Q)
        for v in adjlist[u]:
            if colour[v] = white:
                colour[v] = grey
                ~~Enqueue~~ d[v] = d[u] + 1
                π[v] = u
                Enqueue(Q, v)
        colour[u] = black
```

preprocessing

$O(n)$

* Every vertex is enqueued & dequeued once $\Rightarrow O(n)$.
* ~~Eve~~ For every vertex, we traverse through all edges connected to it
  ↳ Total time taken $\Rightarrow O(m)$

Overall: $O(n+m)$ [$n \Rightarrow$ ~~edges~~ nodes, $m \Rightarrow$ edges]

Properties of BFS

① Shortest path

$d[v]$ holds the value of shortest path from s to v

↳ To prove this, we need property 2.

② Suppose that during execution of BFS on graph (V, E), the queue Q contains $\{v_1, v_2, \ldots, v_r\}$ where v_1 is head of Q & v_r is tail, then

$\Rightarrow d[v_i] \leq d[v_{i+1}]$ $\quad$ $\Rightarrow d[v_1]+1 \geq d[v_r]$

$1 \leq i \leq r-1$

↳ Proof by induction on queue operations

Base case: 1 operation ⇒ Enqueue ⇒ $Q = \{s\}$

↳ Trivially true

Hypothesis: True for k operations

$Q = \{v_1, v_2, \ldots, v_r\}$

Inductive step: $k+1^{th}$ operation

→ $k+1^{th}$ operation is dequeue ⇒ v_1 is removed

We know that $d[v_r] \leq d[v_1]+1 \leq d[v_2]+1$ $\quad (d[v_1] \leq d[v_2])$

$\therefore d[v_r] \leq d[v_2]+1$

Proved

→ $k+1^{th}$ operation is enqueue

$Q = \{v_1, v_2, \ldots, v_r, v_{r+1}\}$

The parent of v_{r+1} is u which was dequeued earlier

↳ $d[u] \leq d[v_1]$ & $d[v_{r+1}] = d[u]+1$

$\therefore d[v_{r+1}] \leq d[v_1]+1$

Now we need to show $d[v_r] \leq d[v_{r+1}]$

As $d[v_r] \leq d[u]+1$ (Before u was dequeued)

$d[v_r] \leq d[v_{r+1}]$ $\quad (d[u]+1 = d[v_{r+1}])$

Hence proved

Proof of property ①

$\delta(u,v)$ = shortest path from u to v. We need to show $d[v] = \delta(s,v)$

Let $u, v \in V[G]$ & $(u,v) \in E[G]$, $\delta(s,v) \leq \delta(s,u)+1$

Clearly, $d[v] \geq \delta(s,v)$

↳ If $d[v] < \delta(s,v)$, there exists a shorter path from s to v than $\delta(s,v)$

↳ Contradiction

Claim: $d[v] = \delta(s,v)$

Let v be the first vertex from s where $d[v] > \delta(s,v)$ [All prev vertices follow $d[u] = \delta(s,u)$]

Say (u,v) is the last edge in shortest path from s to v

$d[v] > \delta(s,v) = \delta(s,u)+1 = d[u]+1$

↳ $d[v] > d[u]+1 \Rightarrow$ Contradiction

When u is dequeued, ~~v can't be black as u is neighbour~~ v can't be black

if v was white, $d[v] = d[u]+1$

if v was grey, $d[v] = d[w]+1 \leq d[u]+1$

where w is the node that discovers v before u.

↳ By prop ②, $d[w] \leq d[u]$ as w comes before u

→ Depth-First Search

$\pi[v]$ ⇒ predecessor of v

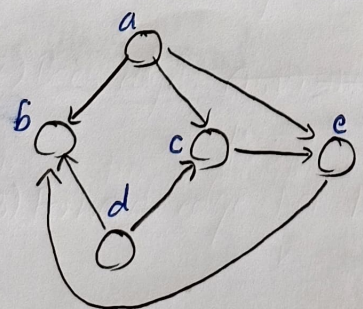
$d[v]$ ⇒ time when v is discovered (when v turns grey)

$f[v]$ ⇒ time when all neighbours of v are visited (when v turns black)

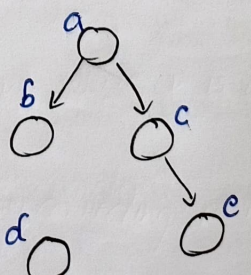
```
DFS(G):
    for each vertex u ∈ V[G]:
        colour[u] = white
        π[u] = NIL
    time = 0
    for each vertex u ∈ V[G]:
        if colour[u] = white:
            DFS_visit(u)

DFS_visit(u):
    colour[u] = grey
    time = time + 1
    d[u] = time
    for each vertex v ∈ adjList[u]:
        if colour[v] = white:
            π[v] = u
            DFS_visit(v)
    colour[u] = black
    f[u] = time
    time = time + 1
```

Eg:



Traversal tree :



time:	1	2	3	4	5	6	7	8	9	10
node:	a	b	b	c	e	e	c	a	d	d

d[a] d[b] f[b] d[c] d[e] f[e] f[c] f[a] d[d] f[d]

Properties of DFS

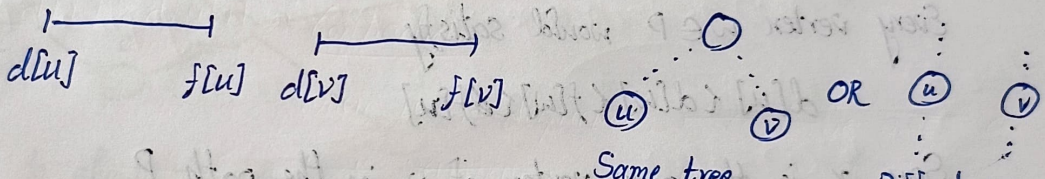
① We get multiple trees in DFS. They form a ~~forrest~~ forest

② $\pi[v] = u$ if & only if DFS-visit(v) was called during a search of u's adjList.

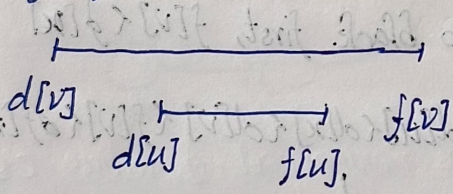
③ Parenthesis thm

In any DFS of a directed or undirected graph $G=(V,E)$, for any 2 vertices u & v, exactly 1 of the 3 conditions holds:

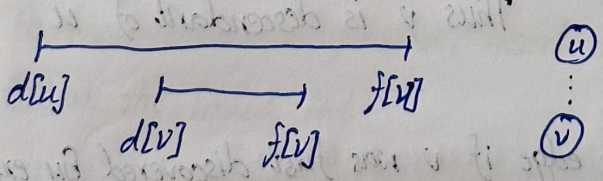
1) Intervals $[d[u], f[u]]$ & $[d[v], f[v]]$ are completely disjoint. Neither u nor v is a descendant of ~~or~~ the other in the DFS forest.



2) $[d[u], f[u]]$ is entirely contained in ~~[d[v], f[v]]~~ $[d[v], f[v]]$ & u is descendant of v



3) $[d[v], f[v]]$ is entirely contained in $[d[u], f[u]]$ & v is descendant of u



Provable by simple proof by contradiction

④ v is a proper descendant of u if & only if $d[u] < d[v] < f[v] < f[u]$

⑤ White Path thm ***

v is descendant of u iff at the the time $d[u]$, v can be reached from u along a path consisting entirely of white vertices.

↳ Proof

Assume v is descendant of u

↳ This means there is a path from u to v, in traversal tree. This path would be white (undiscovered) when we first discover u (at time $d[u]$)

Assume there exists a white path from u to v in G at $d[u]$ time

↳ Let path be P

Assume v is not descendant of u

However every verten in P would a descendant of u

Every verten $w \in P$ would satisfy

$$d[u] < d[w] < f[w] < f[u]$$

Say x is the prev verten of v in the path P.

As x would be discovered first, $d[x] < d[v]$

For x to go black, v must go black first, $f[v] < f[x]$

As $d[u] < d[x] < f[x] < f[u] \rightarrow d[u] < d[x] < d[v] < f[v] < f[x] < f[u]$

$$\downarrow$$

$$d[u] < d[v] < f[v] < f[u]$$

Thus v is descendant of u

⑥ Classification of Edges

1) Tree edges : (u,v) is a tree edge if v was first discovered by exploring edge (u,v)

2) Back edge : ~~(u,v) where~~ Non-tree edge (u,v) where v is ancestor of u

3) Forward edge: Non-tree edge (u,v) ~~con nher~~ connecting u to descendant v

4) Cross Edge: ~~Edge~~ Non-tree edge b/w diff DFS trees or edges b/w vertices in the same tree, when they aren't ascendant / descendant of one-another.

~~Baseia~~ Basically all other edges apart from 1, 2, & 3

~~Applications~~

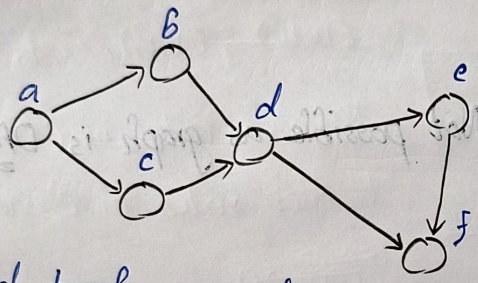
→ Applications of BFS & DFS

① Topological Sort

directed graph (directed acyclic graph)

A topological sort of a DAG G(V,E) is a linear ordering of vertices of G s.t. if G contains an edge (u,v), then u appears before v in the ordering.

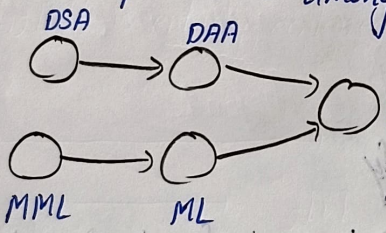
Eg:



a, b, c, d, e, f

Used to show precedence among events

Eg:



DSA, MML, ML, DAA, NLP

Logic

⇒ Call DFS(G) to compute f[v] for each v

⇒ As each vertex is finished, insert it to the front of a linked list

⇒ Return the linked list

A directed graph G is acyclic iff a DFS of G yields no back edges.

↳ Proof

Assume (u,v) is a back edge

↳ Path from v to u in DFS tree exists. With (u,v) we get a cycle. Contradiction as G is acyclic.

Assume cycle $C(v_1, v_2, \ldots, v_k, v_1)$ exists

↳ Let v_1 be the first vertex in C to be discovered.

At $d[v_1]$, $v_2 - v_3 - \ldots - v_k$ are all white.

$\exists$ a white path $v_1 \rightsquigarrow v_k \Rightarrow v_k$ is descendant of v_1

$\Rightarrow (v_k, v_1)$ is a back edge

Proof of correctness of Topological Sort logic

If $(u,v) \in E$, then $f(u) > f(v)$

When the edge (u,v) is explored

Case 1) v is white

↳ v_2 is descendant of $u \Rightarrow f(u) > f(v)$

Case 2) v is grey

↳ (u,v) is back edge $\Rightarrow$ Not possible as graph is DAG

Case 3) v is black

↳ $f(v) < f(u)$

(2) Strongly Connected Components

This makes sense in directed graphs only

A SCC of a directed graph $G(V, E)$ is maximal set of vertices $C \subseteq V$ s.t for every pair of vertices u & v in C, v is reachable from u & u is reachable from v.

Algorithm ~~Logic~~ used

1) Call DFS(G) & compute $f[u]$ for every vertex u

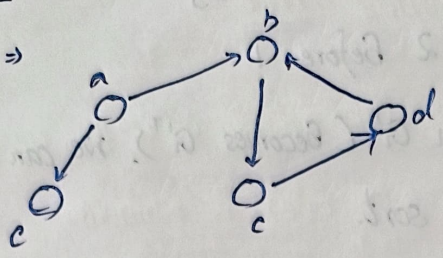
2) Compute $G^T \Rightarrow G^T$ is the same graph with the edges reversed.

3) Call DFS(G^T), consider the vertices in order of decreasing order of $f[u]$

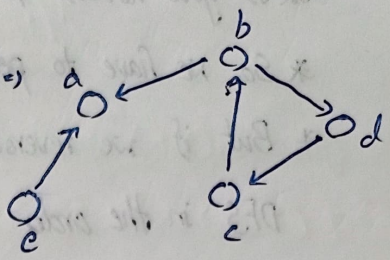
↳ Each tree in the DFS forest of DFS(G^T) is a separate SCC.

Dry run

$G(V,E) \Rightarrow$



$G^T(V,E) \Rightarrow$



Step 1: DFS tree of $G(V,E) \Rightarrow$ $a \rightarrow b \rightarrow c \rightarrow d$, $a \rightarrow e$

f[u] ordered in decreasing order : d a e b c d.

Step 2 : Done above

Step 3: DFS (G^T) on a $\Rightarrow$ (a)

DFS (G^T) on e $\Rightarrow$ (e)

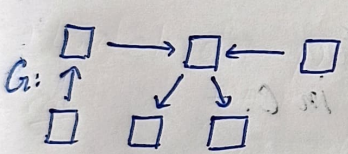
DFS (G^T) on b $\Rightarrow$ (b) → (d) → (c)

} These are the 3 SCCs in the graph G

Logic behind the algorithm

Say we have the following graph (we compiled all the SCCs together)

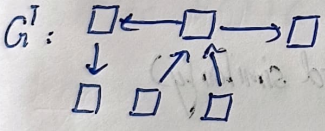
G:



Here, □ is a SCC (It has nodes inside)

In this graph, if there is an edge b/w 2 SCCs, one node in the first SCC has an edge to a node in the second SCC

G^T:

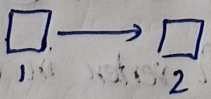


If we reverse the edges, the ~~edg~~ SCCs won't be affected. The edges would be reversed.

Aim: We need to develop an algorithm that outputs each SCC (each □) separately.

↳ For achieving this we ~~need~~ will perform DFS.

↳ The challenge here is there might be a chance to get □→□ which is wrong.

Graph: G') 

Solution to this challenge:

Call DFS on SCC 2 before SCC 1.

This way we get $\square_1$ & $\square_2$ separately.

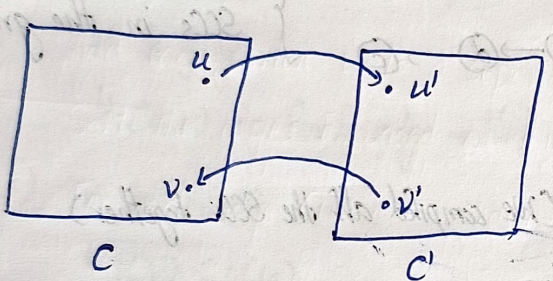
* If you notice, topological sort of G' is $\{1, 2\}$
* So, we have to perform DFS on 2 before 1
* But if we reverse the edges in G' (G' becomes G'^T), we can perform DFS in the order of topological sort.

This is essentially what we are doing.

Proof of correctness

Claim: B/w 2 disjoint SCCs C & C', if there is a path from C to C', there can't exist a path from C' to C

↳ Prove claim using proof by contradiction



(u, u') & (v, v') exists

Now each vertex in C is connected to each vertex in C' via (u, u')

↳ ~~Each ver~~

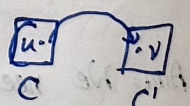
Each vertex in C' is connected to each vertex in C.

This means C & C' aren't separate & disjoint.

↳ Contradiction

So, there are 2 cycles in the graph of SCCs (can be proved similarly)

Claim: If $(u, v) \in E$ s.t. $u \in C$ & $v \in C'$, then $f(C) > f(C')$



↳ Proof:

$f(C) = \max_{vec} f(v)$; $d(C) = \min_{vec} d(v)$

Case I: $d(C) < d(C')$

Say $d(C) = d[x]$ (x is first node in C' to be discovered

Every vertex in C & C' (except x) is white @ $d[x]$

There ∃ white path from x to each vertex in C & C'

For C to turn black, C' must turn black as $C \to C' \Rightarrow f(C) > f(C')$

Case II: $d(C) > d(C')$

$d(C') = d[y]$

$\exists$ white path from y to every node in C'

$f(C') < d(C) < f(C)$

Claim: Suppose $(u, v) \in E^T$ where $u \in C$, $v \in C'$, then $f(C) < f(C')$

Now proof by induction of on DFS trees k

↳ Base case: $k = 0$, trivially true

↳ Inductive hypothesis: k trees returned corresponding to SCC.

↳ Constructed in ↓ order of $f(u)$

↳ Inductive step

Let u be root of $(k+1)^{th}$ DFS tree.

Let S be the SCC which contains S.

The $(k+1)^{th}$ tree must be S

$V(S) \subseteq V(T)$ & $V(T) \subseteq V(S)$

↳ As all vertices in S are connected to u via a white path. So $V(S) \subseteq V(T)$

Now for $V(T) \subseteq V(S)$

↳ Assume not, $\exists n \in T$ s.t $x \notin S$

Say $x \in S'$, there is an edge from S to S'.

But $f(S') < f(S)$ as $S \to S'$

This means S' would be one of the k trees before & we won't visit it again

Hence, $V(T) \subseteq V(S) \Rightarrow S = T$

Hence Proved

Minimum Spanning Tree

Given $G(V,E)$ graph & the weights of the edges $c: E \to \mathbb{N}$, construct tree T $(T \subseteq E)$ s.t. $G(V,T)$ is connected & $\sum_{e \in T} c(e)$ is minimum

This can be computed with a greedy strategy.

There are 2 strategies:

1) Kruskal's Algorithm

Insert edges in increasing order of cost so that no cycles are formed.

```
Krushal (V,E)
    A = ϕ
    Add each vertex v to a separate component of A
    Sort the edges E based on thier weights
    For each edge (u,v) in order:
        if u & v are not in same component:
            A = A ∪ (u,v)
```

Runtime Analysis

Sorting the edges → $O(m \log m)$

Checking whether to 2 elements are of same component & merge → $O(\log n)$

↳ Done m times → $O(m \log n)$ — Union Find DS

Adding vertices to A → $O(n)$

So, $O(n + m\log n + n\log n)$ ∎

2) Prim's Algorithm

Start @ a root node s & try to greedily grow a tree from s outwards

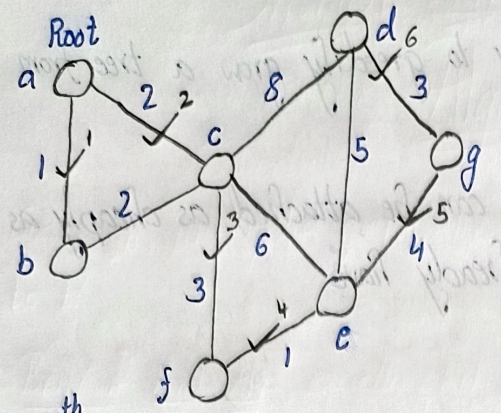
At each step, add the node that can be attached as cheaply as possible to the partial tree we already have

```
Prims (V, E):
    For each u ∈ V:
        key[u] = ∞   (Min cost to add u to the soln)
        π[u] = NIL   (The neighbour of u that realised the key)
    key[r] = 0
    Q = V  (ordered by key; Priority Queue with heap implementation)
    While Q ≠ ϕ:
        u = ExtractMin(Q)
        Add edge (u, π(u)) to solution tree
        For each edge v ∈ AdjList[u]:
            if v ∈ Q & w(u,v) < key[v]:
                π[v] = u
                key[v] = w(u,v)
```

Runtime Analysis

	Binary heap	Fibo heap
Finding Min Element from Q → n times	~~$O(n^2)$~~ → $O(n \log n)$	→ $O(n \log n)$
Updating the value of key → m times	→ $O(m \log n)$	→ $O(m)$

* Dry Run of Prim's Algorithm:



$\checkmark^n$: n^{th} edge added to MST

Vertex	Key	π
a	0 ✓	NIL
b	$\infty \xrightarrow{a} 1$ ✓	a
c	$\infty \xrightarrow{a} 2$ ✓	a
d	$\infty \xrightarrow{c} 8 \xrightarrow{e} 5 \xrightarrow{g} 3$	g
e	$\infty \xrightarrow{c} 6 \xrightarrow{f} 1$ ✓	f
f	$\infty \xrightarrow{c} 3$ ✓	c
g	$\infty \xrightarrow{e} 4$ ✓	e

→ Proof of Correctness of Kruskal's & Prim's Algorithm

* Property: Some edges are always safe to be added to a MST

* Cut Property

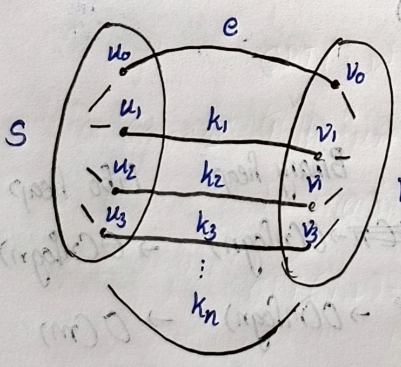
Assume all edge costs are distinct, $S \subset V$ & $S \neq \phi$

We have 2 disjoint set of vertices: S & ~~V~~ $V \setminus S$

Let e be the edge of min. weight, connecting S & $V \setminus S$

Claim: e is a safe edge, aka, every MST contains e

↳ Proof by contradiction



Say there exists an MST with the connecting edges b/w S & $V \setminus S$ be $\{k_1, k_2, \ldots, k_n\}$ s.t no $k_i = e$

Let us add e & remove any k_i (w.l.o.G k_1)

↳ This set of edges would cost less than the original MST

↳ We have to show this set of edges is ~~an~~ MST

1) There must be no loops ✓

↳ Original MST had no loops, adding u formed a loop, removing k_1 broke that loop

2) Every 2 vertices are connected ✓

↳ Say 2 vertices were connected via k_1 in original MST

These 2 vertices now would be connected via c or $k_2 k_3 \ldots k_n$

bridge

↳ The other vertices pairs remain undisturbed.

So, the given set of edges is a ST with less weight than original MST

↳ Contradiction

Hence Proved

Optimality of Prims

It would be enough if we show that every edge added is a safe edge.

Apply Cut Property : S ⇒ Partial Solution

Every edge added is edge b/w S & V\S which has least weight

Hence Proved

Optimality of Kruskals

Similar logic

Every edge added is min weight edge connecting 2 disjoint components (they don't form a cycle)

Hence Proved

Shortest Path Algorithms

If G is an unweighted graph, how do we find the shortest path from s to t?

→ Bellman-Ford Alg.

Condition to work: G may contain -'ve edges, BUT NO -'ve cycles.

↳ Now that G has no -'ve cycles, the algorithm won't loop in the -'ve cycle forever

↳ So, the ~~shortest~~ shortest path from s to t would simple & have atmost n-1 edges.

Let us solve this using DP.

Step 1: Define the subproblem & final answer

$OPT(i,v) \Rightarrow$ min cost to reach v from t with atmost i edges.

$OPT(n-1,s) \Rightarrow$ final answer

Step 2: Write the recurrence

Case I: Path from t to v has atmost i-1 edges

$$OPT(i,v) = OPT(i-1,v)$$

Case II: Path from t to v has i edges exactly.

We check paths from neighbours of v

$$OPT(i,v) = \min_{w \in Adjlist(v)} [weight(v,w) + OPT(i-1,w)]$$

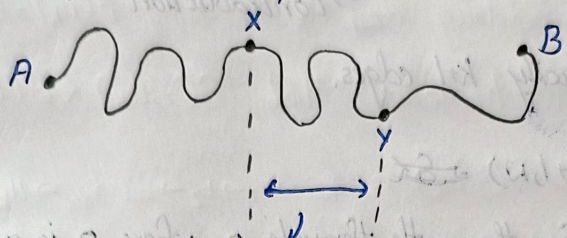
So, $$OPT(i,v) = \min \left\{ OPT(i-1,v), \min_{w \in N(v)} [weight(v,w) + OPT(i-1,w)] \right\}$$

Step 3: Proof of correctness

Let us apply induction upon i & use optimal substructure property.

What is optimal substructure property?

↳ Say the shortest path b/w nodes A & B is as follows:



This path is the shortest path b/w X & Y

Optimal substructure guarantees that the sub-path b/w X & Y in the shortest path from A to B is the shortest path b/w X & Y.

This is trivial & can be proved by contradiction. If there exists a shorter path b/w X & Y, shortest path b/w A & B would take it.

Proof of Bellman-Ford Algo using induction on i:

Base Case: i=0

$OPT(0,t) = 0, OPT(0,w) = \infty$ for any $w \neq t$

True

Inductive hypothesis: i = k

We have $OPT(k,w)$ for all $w \in V$

This means we have shortest path b/w t & w of atmost k edges.

Inductive step: i = k+1

We compute $OPT(k+1,w)$ using recurrense.

Proof by contradiction ↷ Say $OPT(k+1,w)$ is not optimal cost shortest path of atmost k+1 edges b/w w & t. Say S is the shortest cost path as such.

This means $|S| < OPT(k+1,w)$

Case I: δ has atmost k edges

Then $OPT(k+1, w)$ would take the value of $OPT(k, w)$

Then $OPT(k+1, w) = \delta$

Contradiction.

~~Case II~~ (OR) $\delta < OPT(k+1, w) \Rightarrow \delta < OPT(k, w)$

↳ Contradiction

Case II: δ has exactly $k+1$ edges.

$\delta < OPT(k+1, w)$ ~~$\Rightarrow \delta <$~~

Say δ takes the path through p, where p is a neighbour of w

Then, $\because \delta < OPT(k+1, w)$

$$\delta_{t-p} + c(p, w) < OPT(k, p) + c(p, w)$$

Cost of path from t to p $\Rightarrow \delta_{t-p} < OPT(k, p)$

↳ Contradiction

Thus $OPT(k+1, w)$ is correct

Hence Proved

Runtime Analysis

$O(n, m)$

↳ For a fixed (i, v), the value of $OPT(i, v)$ is used for the computation of $OPT(i+1, w)$ for degree(v) number of times.

↳ Σ degree$(v) = m$

↳ We are computing it n times (for diff is) $\Rightarrow O(n, m)$

→ Dijkstra's Algorithm

Given a graph G & ~~sour~~ source s, find shortest path from s to all vertices. (All edge weights must be positive)

Let us solve this by a greedy strategy:

Maintain a set S of vertices u for which we have determined a shortest path distance $d(u)$ from s. (or from 'explored')

Dijkstra(G):

$S = \{s\}$

$d(s) = 0$

While $S \neq V$:

Select a node $v \notin S$ s.t. $d'(v) = \min_{e=(u,v), u \in S} d(u) + \ell_e$ is min

$S = S \cup \{v\}$

$d(v) = d'(v)$

Basically,

Greedy choice: Always pick the next vertex with the smallest current distance.

Optimal Substructure: Shortest paths can be built by extending shortest paths.

Intuition behind the Alg

The alg picks the vertex u with smallest tentative distance $d[u]$ & fixes it as shortest path from u to s.

We have to prove that this path is indeed the shortest path to u.

↳ Proof by induction on size of set S.

Base case: $|S| = 1$

$S = \{s\}$, $d(s) = 0$

Trivially true.

Inductive hypothesis: $|S| = k$

$S = \{s, u_1, u_2, \ldots, u_{k-1}\}$

$d(u_i) = \delta(s, u_i)$

Here $\delta(s, u_i)$ is the actual shortest path from s to u_i

Inductive step: $|S| = k+1$

We are adding u_k to S & $d(u_k) = \delta(s, u_k)$

Assume (contradiction) that $\delta(s, u_k) < d(u_k)$

Say path $\delta(s, u_k)$ is $s \to v_1 \to v_2 \to \ldots v_m \to u_k$

In this path say v_j is the first vertex yet to be extracted from the queue. In the sense, v_{j-1} has been extracted.

So, $\delta(s, v_{j-1}) = d(v_{j-1})$

As $w(v_{j-1}, v_j) \geq 0$, $d(v_j) \leq \underbrace{d(v_{j-1}) + w(v_{j-1}, v_j) = \delta(s, v_j)}$

By optimal substructure property.

This means ~~$d(v_j) < \delta(s, v_j)$~~

$d(v_j) \leq \delta(s, u_k) < d(u_k)$

$d(v_j) < d(u_k)$

This is a contradiction as u_k is extracted from queue before v_j (v_j is still in queue) but $d(v_j) < d(u_k)$

Thus, assumption is wrong.

$d(u_k) = \delta(s, u_k)$

Hence Proved

Runtime Analysis

	Binary heaps	Fibonacci heaps
* n ExtractMin operations	$\to n \log n$	$\to n \log n$
* m Updates	$\to m \log n$	$\to m$

Binary $\Rightarrow O((m+n)\log n)$

Fibonacci $\Rightarrow O(n \log n + m)$ //