PDSA Notes

by

Gagneet Kaur

Date
January 24, 2022

Friends
PAGE NO.
DATE:

WEEK 5

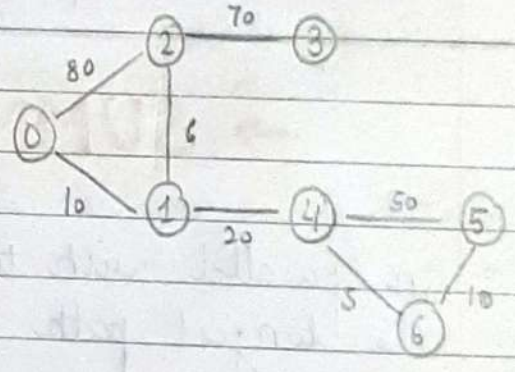
Shortest Paths In Weighted Graphs

WEIGHTED GRAPHS

→ Recall that BFS explores a graph level by level

→ BFS computes shortest path, in terms of number of edges, to every reachable vertex

→ May assign values to edges
- cost, time, distance
- Weighted graph



→ $G = (V, E)$, $W : E \to \mathbb{R}$

→ ADJACENCY MATRIX : Record weights along with edge information - weight is always zero if no edge

→ ADJACENCY LIST : Record weights along with edge information

→ In weighted graph, each edge has a cost.
→ Length of a path is the sum of weights

PAGE NO.
DATE:

SHORTEST PATHS IN WEIGHTED GRAPHS

→ BFS computes shortest path, in terms of number of edges, to every reachable vertex

→ In a weighted graph, add up the weights along a path

→ Weighted shortest path need not have minimum no. of edges
  . shortest path from 0 to 2 is via 1.

SHORTEST PATH PROBLEMS

SINGLE SOURCE SHORTEST PATHS

→ Find shortest paths from a fixed vertex to every other vertex

→ Transport finished product from factory (single source) to all retail outlets

→ Courier company delivers items from distribution centre (single source) to addressees

ALL PAIRS SHORTEST PATHS

→ Find shortest paths between every pair of vertices i & j

→ Optimal airline, railway, road routes between cities

Negative Edge Weights
→ should not have negative cycles
→ w/o negative cycles, shortest paths still well defined

PAGE NO.
DATE:

NEGATIVE EDGE WEIGHTS

→ Can negative edge weights be meaningful?

→ Taxi driver trying to head home at the end of the day
- Roads with few customers, drive empty (+ve weight)
- Roads with many customers, make profit (-ve weight)
- Find a route toward home that minimizes the cost

NEGATIVE CYCLES

→ A negative cycle is one whose weight is negative
- Sum of the weights of edges that make up the cycle

→ By repeatedly traversing a negative cycle, total cost keeps decreasing

→ If a graph has a negative cycle, shortest paths are not defined

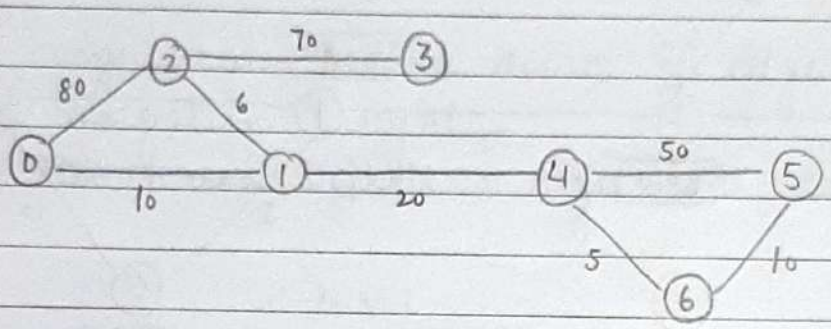
→ Without negative cycles, we can compute shortest paths even if some weights are negative

Date
January 25, 2022

Shrinesh
PAGE NO.
DATE:

Single Source Shortest Paths

SINGLE SOURCE SHORTEST PATHS



→ Weighted graph: $G = (V, E)$
$W : E \rightarrow \mathbb{R}$

→ Single Source Shortest Paths
- find shortest paths from a fixed vertex to every other vertex

→ Assume, for now, that edge weights are all non-negative

→ Compute shortest paths from 0 to all the other vertices

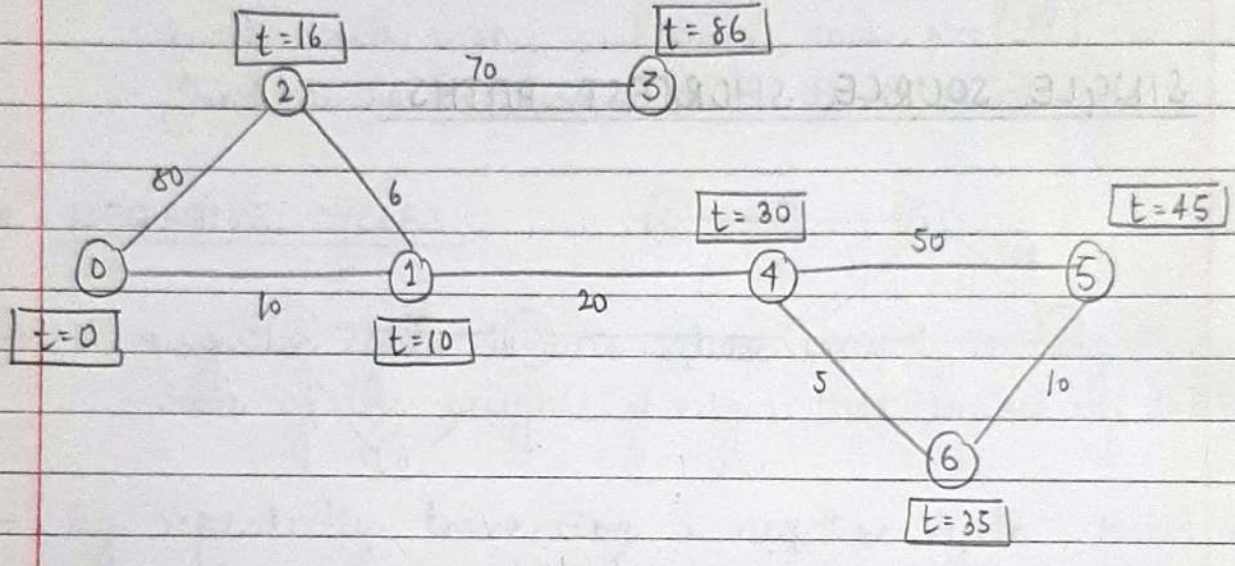
→ Imagine vertices are oil depots, edges are pipelines

→ Set fire to oil depot at vertex 0

Shrinath
PAGE NO.
DATE:

- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex

.....



- Compute 'expected burn time' for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours
- Algorithm due to Edsger W Dikjstra

DIKJSTRA'S ALGORITHM : proof of correctness

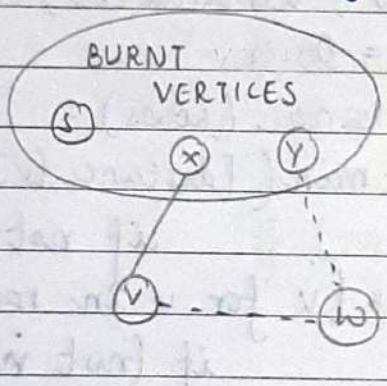
- Each new shortest path we discover extends an earlier one
- By induction, assume we have found shortest paths

to all vertices already burnt.

→ Next vertex to burn is v, via x

→ Cannot find a shorter path later from y to v via w
- Burn time of w ≥ burn time of v
- Edge from w to v has weight ≥ 0

→ This arguments breaks down if edge (w,v) can have negative weight
- Can't use Dijkstra's Algo with negative edge wts.



IMPLEMENTATION

→ Maintain 2 dictionaries with vertices as keys
- visited, initially False for all v (burnt vertices)
- distance, initially infinity for all v (expected burn time)

→ Set distance[s] to 0

→ Repeat until all reachable vertices are visited
- Find unvisited vertex nextv with min. distance

- set visited[nextv] to True
- Recompute distance[v] for every neighbour v of nextv

CODE : (1) Using Adjacency Matrix

```
def dijkstra (WMat , s) :
    (rows , cols , x) = WMat.shape
    infinity = np.max (WMat) * rows + 1
    (visited , distance) = ( {} , {} )
    for v in range (rows) :
        (visited[v] , distance[v]) = (False, infinity)
    distance[s] = 0
    for u in range (rows) :
        nextd = min ( [distance[v] for v in range (rows)
                       if not visited[v]] )
        nextvlist = [v for v in range (rows)
                     if (not visited[v]) and
                        distance[v] == nextd ]
        if nextvlist == [ ] :
            break
        nextv = min (nextvlist)
        visited[nextv] = True
        for v in range (cols) :
            if WMat[nextv , v, 0] == 1 and (not visited[v]
                distance[v] = min (distance[v] , distance[nextv]
                                   WMat[nextv, v, 1])

    return (distance)
```

initialization

next distance

Shrinath
PAGE NO.
DATE:

(2) Using Adjacency List

```
def dijkstralist(Wlist, s):
    infinity = 1 + len(Wlist.keys()) *
                max([d for u in Wlist.keys()
                     for (v,d) in Wlist[u]])

    (visited, distance) = ({}, {})
    for v in Wlist.keys():
        (visited[v], distance[v]) = (False, infinity)

    distance[s] = 0
    for u in Wlist.keys():
        nextd = min([distance[v] for v in Wlist.keys()
                     if not visited[v]])
        nextvlist = [v for v in Wlist.keys()
                     if (not visited[v]) and
                        distance[v] == nextd]

        if nextvlist == []:
            break
        nextv = min(nextvlist)
        visited[nextv] = True
        for (v,d) in Wlist[nextv]:
            if not visited[v]:
                distance[v] = min(distance[v],
                                  distance[nextv]+d)

    return (distance)
```

Shrinath
PAGE NO.
DATE:

COMPLEXITY

→ Setting infinity takes $O(n^2)$ time

→ Main loop runs n times
- Each iteration visits one more vertex
- $O(n)$ to find next vertex to visit
- $O(n)$ to update distance[v] for neighbours

→ Overall $O(n^2)$

→ If we use an adjacency list
- Setting infinity & updating distances both $O(m)$, amortized
- $O(n)$ bottleneck remains to find next vertex to visit
- Better data structure?

- Dijkstra's algorithm computes single source shortest paths
- Use a greedy strategy to identify vertices to visit
  - → Next vertex to visit is based on shortest distance computed so far
  - → Need to prove that such a strategy is correct
  - → Correctness requires edge weights to be non-negative
- Complexity is $O(n^2)$
  - → Even with adj. list
  - → Bottleneck is identifying unvisited vertex with min. distance

Date
uary 26, 2022

Shrinath
PAGE NO.
DATE:

Single Source Shortest Paths with Negative Wts.

DIJKSTRA'S ALGORITHM

→ Recall the burning pipeline analogy

→ We keep track of the following.
- The vertices that have been burnt
- The expected burn time of vertices

→ Initially
- No vertex is burnt
- Expected burn time of source vertex is 0
- Expected burn time of rest is ∞

→ While there are vertices yet to burn
- Pick unburnt vertex with minimum expected burn time, mark it as burnt
- Update the expected burn time of its neighbours

INITIALIZATION (assume source vertex 0)

- $B(i)$ = False, for $0 \leq i < n$

$UB = \{ k \mid B(k) = \text{False} \}$

- $EBT(i) = \begin{cases} 0 & , \text{ if } i = 0 \\ \infty & , \text{ otherwise} \end{cases}$

UPDATE, if $UB \neq \emptyset$

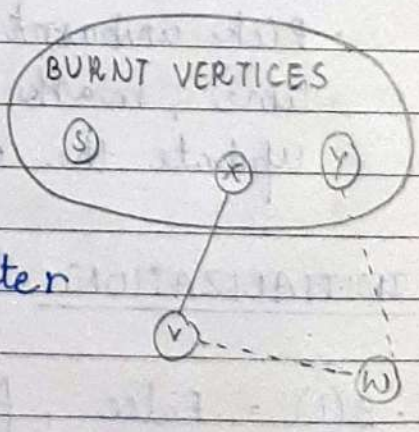
- Let $j \in UB$ such that $EBT(j) \leq EBT(k)$ for all $k \in UB$
- Update $B(j) = True$, $UB = UB \setminus \{j\}$
- For each $(j,k) \in E$ such that $k \in UB$, $EBT(k) = \min(EBT(k), EBT(j) + W(j,k))$

CORRECTNESS REQUIRES NON-NEGATIVE EDGE WEIGHTS

→ Each new shortest path we discover extends an earlier one

→ By induction, assume we have found shortest paths to all vertices already burnt

→ Next vertex to burn is v, via x



→ Cannot find a shorter path later from y to v via w

- Burn time of w ≥ burn time of v
- Edge from w to v has weight ≥ 0

→ This argument breaks down if edge (w,v) can have negative weight

EXTENDING TO NEGATIVE EDGE WEIGHTS

→ The difficulty with negative edge weights is that we stop updating the burn time once a vertex is burnt

→ What if we allow updates even after a vertex is burnt?

→ Recall, negative edge weights are allowed, but no negative cycles

→ Going around a cycle can only add to the length

→ Shortest route to every vertex is a path, no loops

→ Suppose minimum weight path from 0 to k is

$$0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \cdots \xrightarrow{w_{l-1}} j_{l-1} \xrightarrow{w_l} k$$

Need not be minimum in terms of no. of edges

→ Every prefix of this path must itself be a minimum weight path

- $0 \xrightarrow{w_1} j_1$
- $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2$

Shrisagh
PAGE NO.
DATE:

- $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} j_3$
- ....
- $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} j_3 \xrightarrow{w_4} \cdots \xrightarrow{w_{k-1}} j_{k-1}$

→ Once we discover shortest path to j_{k-1}, next update will fix shortest path to k

→ Repeatedly update shortest distance to each vertex based on shortest distance to its neighbours
- Update can't push this distance below actual shortest distance

→ After l updates, all shortest paths using $\leq l$ edges have stabilized
- Minimum weight path to any node has at most $n-1$ edges
- After $n-1$ updates, all shortest paths have stabilized

BELLMAN - FORD ALGORITHM

INITIALIZATION (source vertex 0)

$D(j)$: minimum distance known so far to vertex j

$$D(j) = \begin{cases} 0 & , \text{ if } j = 0 \\ \infty & , \text{ otherwise} \end{cases}$$

Repeat $n-1$ times

For each vertex $j \in \{0, 1, \ldots, n-1\}$ for each edge $(j, k) \in E$,

$$D(k) = \min(D(k), D(j) + w(j, k))$$

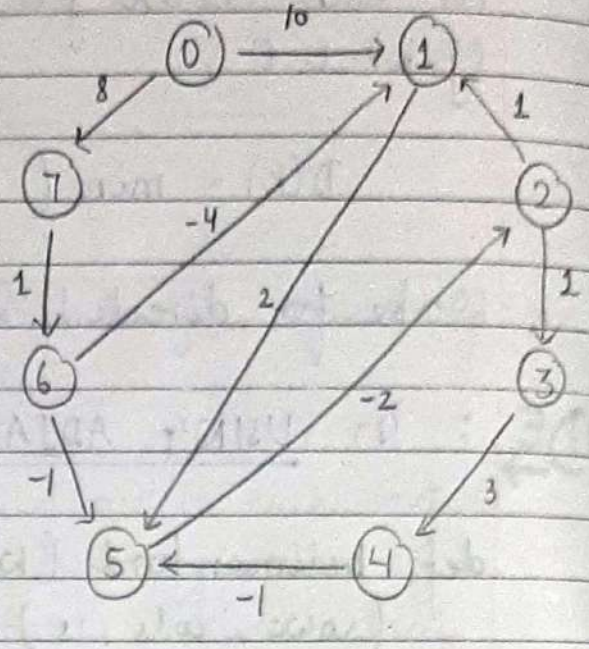
Works for directed & undirected graphs

CODE : (1) USING ADJACENCY MATRIX

```
def bellmanford (WMat, s) :
    (rows, cols, s) = WMat.shape
    infinity = np.max(WMat) * rows + 1
    distance = {}
    for v in range (rows) :
        distance [v] = infinity

    distance [s] = 0

    for i in range (rows) :
        for u in range (rows) :
            for v in range (cols) :
                if WMat [u, v, 0] == 1 :
                    distance [v] = min (distance [v],
                                        distance [u] + WMat [u,v,1])

    return (distance)
```

Initialize $D(0) = 0$
For each $(j, k) \in E$, update $D(k) = \min(D(k), D(j) + w(j, k))$

V	D(v)							
0	0	0	0	0	0	0	0	0
1	∞	10	10	5	5	5	5	5
2	∞	∞	∞	10	6	5	5	5
3	∞	∞	∞	∞	11	7	6	6
4	∞	∞	∞	∞	∞	14	10	9
5	∞	∞	12	8	7	7	7	7
6	∞	∞	9	9	9	9	9	9
7	∞	8	8	8	8	8	8	8



What if there was a negative cycle?

Distance would continue to decrease

Check if update n reduces any $D(v)$

COMPLEXITY

→ Initializing infinity takes $O(n^2)$ time

→ The outer update loop runs $O(n)$ times

→ In each iteration, we have to examine every edge in the graph.
- This take $O(n^2)$ for an adjacency matrix

→ Overall, $O(n^3)$

- If we shift to ADJACENCY LISTS
  - Initializing infinity is O(m)
  - Scanning all edges in each update iteration is O(m)

CODE: (2) USING ADJACENCY LIST

```
def bellmanfordlist (~~wmat~~ WList, s) :
    infinity = 1 + len(WList.keys()) *
                   max([d for u in WList.keys()
                        for (v, d) in WList[u]])

    distance = {}
    for v in WList.keys() :
        distance[v] = infinity

    distance[s] = 0

    for i in WList.keys() :
        for u in WList.keys() :
            for (v, d) in WList[u] :
                distance[v] = min(distance[v],
                                  distance[u] + d)

    return (distance)
```

→ Now, overall O(mn)

All-Pairs Shortest Paths

SHORTEST PATHS IN WEIGHTED GRAPHS

→ Two types of shortest path problems of interest

SINGLE SOURCE SHORTEST PATHS

- Find shortest paths from a fixed vertex to every other vertex
- Transport finished product from factory (single source) to all retail outlets
- Courier company delivers items from distribution centre (single source) to addresses
- Dijkstra's algorithm (non-negative weights), Bellman-Ford algorithm (allows negative weights)

ALL PAIRS SHORTEST PATHS

- Find shortest paths between every pair of vertices i & j
- Optimal airline, railway, road routes between cities
- Run Dijkstra or Bellman-Ford from each vertex
- Is there another way?

TRANSITIVE CLOSURE

→ Adjacency matrix A represents paths of length 1

→ Matrix multiplication, $A^2 = A \times A$

- $A^2[i,j] = 1$, if there is a path of length 2 from i to j
- For some k, $A[i,k] = A[k,j] = 1$

→ In general, $A^{l+1} = A^l \times A$

- $A^{l+1}[i,j] = 1$, if there is a path of length $l+1$ from i to j
- For some k, $A^l[i,k] = 1$, $A[k,j] = 1$

→ $A^+ = A + A^2 + \ldots + A^{n-1}$

AN ALTERNATIVE APROACH

→ $B^k[i,j] = 1$, if there is path from i to j via vertices $\{0, 1, \ldots, k-1\}$

- Constraint applies only to intermediate vertices between i and j
- $B^0[i,j] = 1$ if there is a direct edge
- $B^0 = A$

→ $B^{k+1}[i,j] = 1$ if

- $B^k[i,j] = 1$ — can already reach j from i via

PAGE NO.
DATE:

$\{0, 1, \ldots, k-1\}$

- $B^k[i,k] = 1$ and $B^k[k,j] = 1$ - use $\{0, 1, \ldots, k-1\}$ to go from i to k and then from k to j

WARSHALL'S ALGORITHM

→ $B^k[i,j] = 1$, if there is path from i to j via vertices $\{0, 1, \ldots, k-1\}$

→ $B^0[i,j] = A[i,j]$
- Direct edges, no intermediate vertices

→ $B^{k+1}[i,j] = 1$ if :
- $B^k[i,j] = 1$, or
- $B^k[i,k] = 1$ and $B^k[k,j] = 1$

→ The algorithm on the left also computes transitive closure - Warshall's Algorithm

→ $B^n[i,j] = 1$, if there is some path from i to j with intermediate vertices in $\{0, 1, \ldots, n-1\}$

→ $B^n = A^+$

→ We adapt Warshall's algorithm to compute all-pairs shortest paths

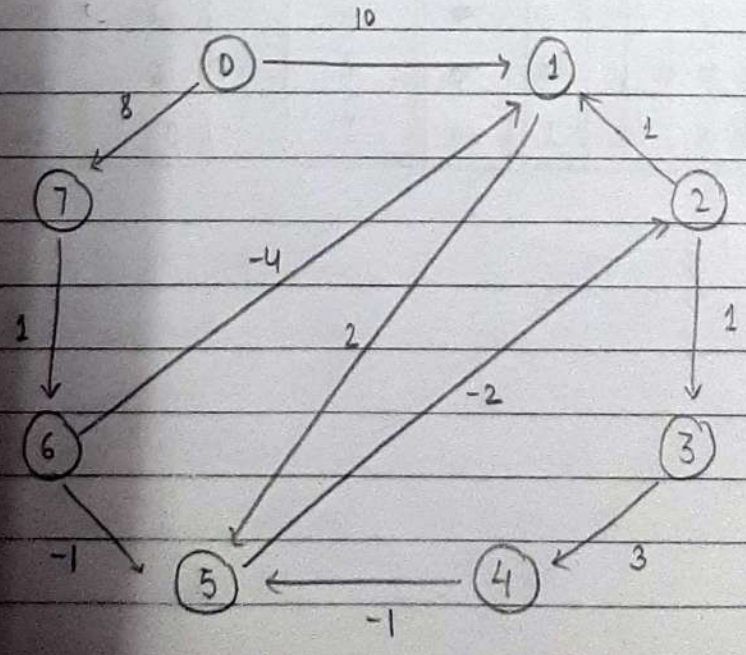
FLOYD - WARSHALL ALGORITHM

→ Let $sp^k[i,j]$ be the length of the shortest path from i to j via vertices $\{0, 1, \ldots, k-1\}$

→ $sp^0[i,j] = w[i,j]$
- No intermediate vertices, shortest path is weight of direct edge
- Assume $w[i,j] = \infty$, if $(i,j) \notin E$

→ $sp^{k+1}[i,j]$ is the minimum of
- $sp^k[i,j]$ → shortest path using only $\{0, 1, \ldots, k-1\}$
- $sp^k[i,k] + sp^k[k,j]$ → combine shortest path from i to k & k to j

→ $sp^n[i,j] = 1$ is the length of the shortest path overall from i to j
- Intermediate vertices lie in $\{0, 1, \ldots, n-1\}$



sp^0	0	1	2	3	4	5	6	7
0	∞	10	∞	∞	∞	∞	∞	8
1	∞	∞	∞	∞	∞	∞	∞	∞
2	∞	1	∞	1	∞	∞	∞	∞
3	∞	∞	∞	∞	3	∞	∞	∞
4	∞	∞	∞	∞	∞	-1	∞	∞
5	∞	∞	-2	∞	∞	∞	∞	∞
6	∞	-4	∞	∞	∞	-1	∞	∞
7	∞	∞	∞	∞	∞	∞	1	∞

sp^1	0	1	2	3	4	5	6	7
0	∞	10	∞	∞	∞	∞	∞	8
1	∞	∞	∞	∞	∞	∞	∞	∞
2	∞	1	∞	1	∞	∞	∞	∞
3	∞	∞	∞	∞	3	∞	∞	∞
4	∞	∞	∞	∞	∞	-1	∞	∞
5	∞	∞	-2	∞	∞	∞	∞	∞
6	∞	-4	∞	∞	∞	-1	∞	∞
7	∞	∞	∞	∞	∞	∞	1	∞

sp^2	0	1	2	3	4	5	6	7
0	∞	10	∞	∞	∞	12	∞	8
1	∞	∞	∞	∞	∞	2	∞	∞
2	∞	1	∞	1	∞	3	∞	∞
3	∞	∞	∞	∞	3	∞	∞	∞
4	∞	∞	∞	∞	∞	-1	∞	∞
5	∞	∞	-2	∞	∞	∞	∞	∞
6	∞	-4	∞	∞	∞	-2	∞	∞
7	∞	∞	∞	∞	∞	∞	1	∞

sp^3	0	1	2	3	4	5	6	7
0	∞	10	∞	∞	∞	12	∞	8
1	∞	∞	∞	∞	∞	2	∞	∞
2	∞	1	∞	1	∞	3	∞	∞
3	∞	∞	∞	∞	3	∞	∞	∞
4	∞	∞	∞	∞	∞	-1	∞	∞
5	∞	-1	-2	-1	∞	1	∞	∞
6	∞	-4	∞	∞	∞	-2	∞	∞
7	∞	∞	∞	∞	∞	∞	1	∞

sp^7	0	1	2	3	4	5	6	7
0	∞	10	10	11	14	12	∞	8
1	∞	1	0	1	4	2	∞	∞
2	∞	1	1	1	4	3	∞	∞
3	∞	1	0	1	3	2	∞	∞
4	∞	-2	-3	-2	1	-1	∞	∞
5	∞	-1	-2	-1	2	1	∞	∞
6	∞	-4	-4	-3	0	-2	∞	∞
7	∞	-3	-3	-2	1	-1	1	∞

sp^8	0	1	2	3	4	5	6	7
0	∞	5	5	6	9	7	9	8
1	∞	1	0	1	4	2	∞	∞
2	∞	1	1	1	4	3	∞	∞
3	∞	1	0	1	3	2	∞	∞
4	∞	-2	-3	-2	1	-1	∞	∞
5	∞	-1	-2	-1	2	1	∞	∞
6	∞	-4	-4	-3	0	-2	∞	∞
7	∞	-3	-3	-2	1	-1	1	∞

IMPLEMENTATION

- Shortest path matrix SP is $n \times n \times (n+1)$
- Initialize $SP[i, j, 0]$ to edge weight $W(i, j)$ or ∞ if no edge
- Update $SP[i, j, k]$ from $SP[i, j, k-1]$ using the Floyd Warshall update rule
- Time complexity is $O(n^3)$
- We only need $SP[i, j, k-1]$ to compute $SP[i, j, k]$
- Maintain two "slices" $SP[i, j]$, $SP'[i, j]$, compute SP' from SP, copy SP' to SP, save space

CODE:

```
def floydwarshall(WMat):
    (rows, cols, x) = WMat.shape
    infinity = np.max(WMat) * rows * rows + 1
    SP = np.zeros(shape = (rows, cols, cols + 1))
    for i in range(rows):
        for j in range(cols):
            SP[i, j, 0] = infinity

    for i in range(rows):
        for j in range(cols):
            if WMat[i, j, 0] == 1
                SP[i, j, 0] = WMat[i, j, 1]
```

```
for k in range(1, cols+1):
    for i in range(rows):
        for j in range(cols):
            sp[i,j,k] = min(sp[i,j,k-1],
                            sp[i,k-1,k-1] + sp[k-1,j,k-1])

return(sp[:,:,cols])
```

→ SUMMARY ←

- Warshall's algorithm is an alternative way to compute transitive closure
  - → $B^k[i,j] = 1$ if we can reach j from i using vertices in $\{0, 1, \ldots, k-1\}$
- Adapt Warshall's algorithm to compute all pairs shortest paths
  - → $sp^k[i,j]$ is the length of the shortest path from i to j using vertices in $\{0, 1, \ldots, k-1\}$
  - → $sp^n[i,j]$ is the length of the overall shortest path
  - → Floyd-Warshall algorithm
- Works with negative edge weights, assuming no negative cycles
- Simple nested loop implementation, time $O(n^3)$
- Space can be limited to $O(n^2)$ by reusing 2 "slices" SP and SP'.

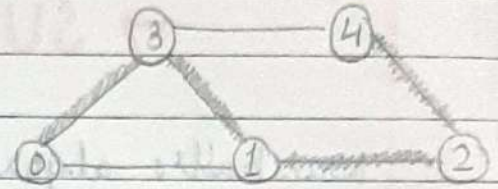
Date January 27, 2022

Shrinath
PAGE NO.
DATE:

Minimum Cost Spanning Trees

SPANNING TREES

→ Retain a minimal set of edges so that graph remains connected



→ Recall that a minimally connected graph is a 'tree'.
- Adding an edge to a tree creates a loop
- Removing an edge disconnects the graph

→ Want a tree that connects all the vertices - <u>spanning tree</u>

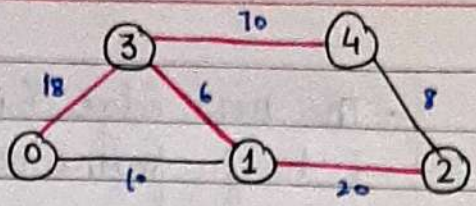
→ More than one spanning tree, in general.

SPANNING TREES WITH COSTS

→ Restoring a road or laying a fibre optic cable has a cost

→ Minimum Cost Spanning Tree
- Add the cost of all edges in the tree
- Among the different spanning trees, choose one with

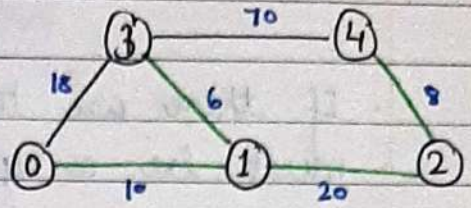
minimum cost.



→ Example

- Spanning tree, Cost is 114 - not minimum cost spanning tree

- Another spanning tree, Cost is 44, - minimum cost spanning tree



SOME FACTS ABOUT TREES

Definition : A tree is a connected acyclic graph

FACT 1 - A tree on n vertices has exactly n-1 edges

- Initially, one single component
- Deleting edge (i,j) must split component (otherwise there is still path from i to j, combine with (i,j) to form cycle)
- Each edge deletion creates one more component
- Deleting n-1 edges creates n components, each an isolated vertex

FACT 2 - Adding an edge to a tree must create a cycle.

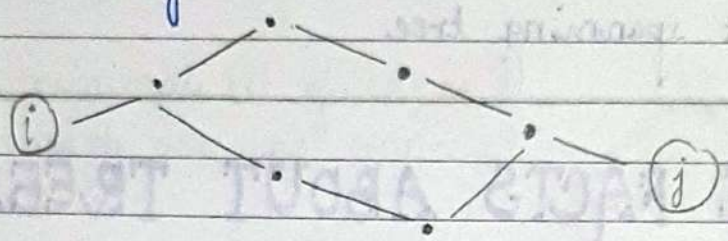
- Suppose we add an edge (i,j)
- Tree is connected, so there is already a path from i to j

Shrinath
PAGE NO.
DATE:

- The new edge (i, j) combined with this path from i to j forms a cycle

FACT 3 - In a tree, every pair of vertices is connected by a unique path.

- If there are two paths from i ~~and~~ to j; these must be a cycle



Observation : Any 2 of the following facts about a graph G implies the third

→ G is connected
→ G is acyclic
→ G has n-1 edges

BUILDING MIN. COST SPANNING TREES

→ We will use these facts about trees to build minimum cost spanning trees.

→ Two natural strategies

→ Start with the smallest edge & 'grow' a tree
  · Prim's Algorithm

→ Scan the edges in ascending order of weight to connect components without forming cycles
  · Kruskal's Algorithm

PRIM's ALGORITHM

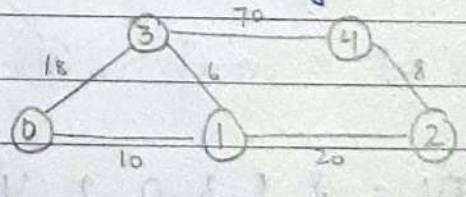
Minimum Cost Spanning Tree (MCST)

· Weighted Undirected Graph, $G = (V, E)$, $W : E \to \mathbb{R}$
G assumed to be connected

· Find a minimum cost spanning tree
  - Tree connecting all vertices in V

· STRATEGY 1
  → Incrementally grow the min. cost spanning tree
  → Start with a smallest weight edge overall
  → Extend the current tree by adding the smallest edge from the tree to a vertex not yet in the tree

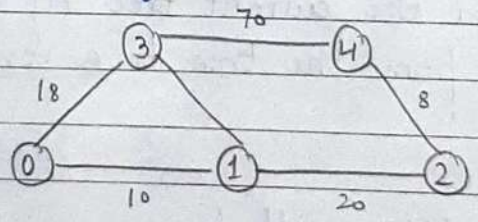
· EXAMPLE
  → start with smallest edge (1,3)
  → Extend the tree with (1,0)
  → Can't add (0,3), forms a cycle
  → Instead, extend the tree with (1,2)
  → Extend the tree with (2,4).



Shriansh
PAGE NO.
DATE:

PRIM's ALGORITHM

- $G = (V, E)$, $W : E \to \mathbb{R}$
- Incrementally build an MCST
  - $TV \subseteq V$: tree vertices, already added to MCST
  - $TE \subseteq E$: tree edges, already added to MCST
- Initially, $TV = TE = \phi$
- Choose minimum weight edge $e = (i, j)$
  - Set $TV = \{i, j\}$
    $TE = \{e\}$ MCST
- Repeat n-2 times
  - Choose min. weight edge $f = (u, v)$ such that $u \in TV$, $v \notin TV$
  - Add v to TV, f to TE
- EXAMPLE :

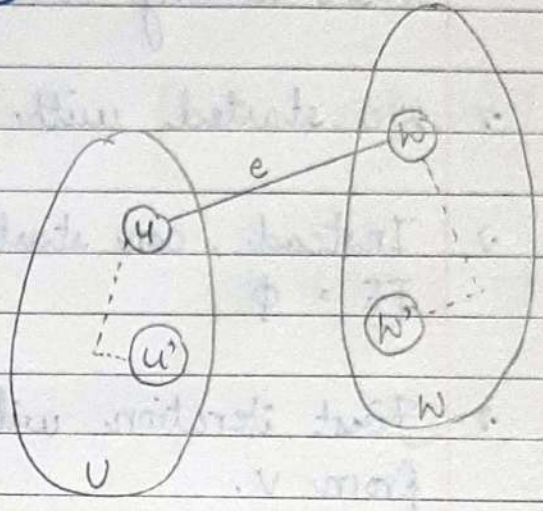


$TV = \{1, 3, 0, 2, 4\}$
$TE = \{(1,3), (1,0), (1,2), (2,4)\}$

CORRECTNESS OF PRIM's ALGORITHM

- Minimum Separator Lemma :

- Let V be partitioned into two non-empty sets U & $W = V \setminus U$
- Let $c = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$
- Every MCST must include e



→ Assume for now, all edge weights distinct

→ Let T be an MCST, $e \notin T$

→ T contains a path p from u to w
- p starts in U, ends in W
- Let $f = (u', w')$ be the first edge on p crossing from U to W.
- Drop f, add e to get a cheaper spanning tree

→ What if two edges have the same weight?

→ Assign each edge a unique index from 0 to m-1

→ Define $(e, i) < (f, j)$ if $W(e) < W(j)$ or $W(e) = W(j)$ and $i < j$.

→ In Prim's algorithm, TV and $W = V \setminus TV$ partition V

→ Algorithm picks smallest edge connecting TV & W, which must belong to every MCST

Shrianeb
PAGE NO.
DATE:

- In fact, for any $v \in V$, $\{v\}$ and $V \setminus \{v\}$ form a partition
- The smallest weight edge leaving any vertex must belong to every MCST
- We started with overall minimum cost edge
- Instead, can start at any vertex v, with $TV = \{v\}$ $TE = \phi$
- First iteration will pick minimum cost edge from v.

IMPLEMENTATION

→ Keep track of :
- visited[v] - is v in the spanning tree?
- distance[v] - shortest dist. from v to tree
- Tree Edges - edges in the current spanning tree

→ Initialize visited[v] to False, distance[v] to infinity

→ first add vertex 0 to tree

→ Find edge (u, v) leaving the tree where distance[v] is minimum, add it to the tree, update distance[w] of neighbours

Shrinath
PAGE NO.
DATE:

CODE :

```
def primlist (WList):
    infinity = 1 + max([d for u in WList.keys()
                        for (v,d) in WList[u]])
    (visited, distance, TreeEdges) = ({}, {}, [])

    for v in WList.keys():
        (visited[v], distance[v]) = (False, infinity)

    visited[0] = True
    for (v,d) in WList[0]:
        distance[v] = d

    for i in WList.keys():
        (mindist, nextv) = (infinity, None)
        for u in WList.keys():
            for (v,d) in WList[u]:
                if visited[u] and (not visited[v]) and
                    d < mindist:
                    (mindist, nextv, nexte) = (d, v, (u,v))

        if nextv is None:
            break
        visted[nextv] = True
        TreeEdges.append(nexte)
        for (v,d) in WList[nextv]:
            if not visited[v]:
                distance[v] = min(distance[v], d)

    return (TreeEdges)
```

COMPLEXITY

→ Initialization takes $O(n)$
→ Loop to add nodes to the tree runs $O(n)$ times
→ Each iteration takes $O(m)$ time to find a node to add
→ Overall time is $O(mn)$, which could be $O(n^3)$!
→ Can we do better ?

IMPROVED IMPLEMENTATION

→ For each v, keep track of its nearest neighbour in the tree
  - visited[v] - is v in the spanning tree?
  - distance[v] - shortest distance from v to the tree
  - nbr[v] - nearest neighbour of v in tree.

→ Scan all non-tree vertices to find nextv with minimum distance

→ Then (nbr[nextv], nextv) is the tree edge to add

→ Update distance[v] and nbr[v] for all neighbours of nextv.

COMPLEXITY :→ Now the scan to find next vertex to add is $O(n)$

→ Very similar to Dijkstra's algorithm, except for the update rule for distance

→ Like Dijkstra's algorithm, this is still $O(n^2)$ even for adjacency lists

→ With a more clever data structure to extract the minimum, we can do better

CODE:

```
def primlist2(WList):
    infinity = 1 + max([d for u in WList.keys()
                          for (v,d) in WList[u]])
    (visited, distance, nbr) = ({}, {}, {})

    for v in WList.keys():
        (visited[v], distance[v], nbr[v]) = (False, infinity, -1)

    visited[0] = True
    for (v,d) in WList[0]:
        (distance[v], nbr[v]) = (d, 0)

    for i in range(1, len(WList.keys())):
        nextd = min([distance[v] for v in WList.keys()
                        if not visited[v]])
        nextvlist = [v for v in WList.keys()
                        if (not visited[v]) and
                           distance[v] == nextd]

        if nextvlist == []:
            break

        nextv = min(nextvlist)
```

```
        visited[nextv] = True
        for (v,d) in Wlist[nextv]:
            if not visited[v]:
                (distance[v], nbr[v]) = (min(distance[v],d),
                                         nextv)

    return (nbr)
```

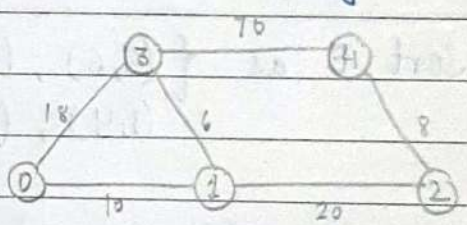
→ SUMMARY ←

- Prim's algorithm grows an MCST starting with any vertex
- At each step, connect one more vertex to the tree using minimum cost edge from inside the tree to outside the tree
- Correctness follows from Minimum Separator Lemma
- Implementation similar to Dijkstra's algorithms
  - Update rule for distance is different
- Complexity is $O(n^2)$
  - Even with adj. lists
  - Bottleneck is identifying unvisited vertex with min distance.

KRUSKAL's ALGORITHM

Minimum Cost Spanning Tree (MCST)

- Weighted undirected graph, $G = (V, E)$, $W: E \to R$
  G assumed to be connected
- Find a minimum cost spanning tree
  → Tree connecting all vertices in V
- STRATEGY 2
  → Start with n components, each a single vertex
  → Process edges in ascending order of cost
  → Include edge if it does not create a cycle
- EXAMPLE
  → start with smallest edge (1,3)
  → Add next small edge (2,4)
  → Add next smallest edge (0,1)
  → Can't add (0,3) forms a cycle
  → Add next smallest edge (1,2)



KRUSKAL's ALGORITHM

→ $G = (V, E)$, $W: E \to R$

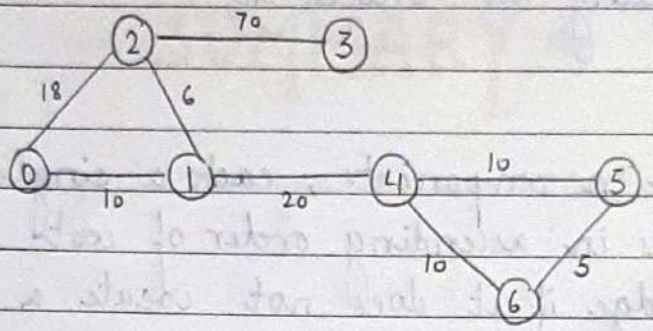
→ Let $E = \{e_0, e_1, \ldots, e_{m-1}\}$ be edges sorted in ascending order by weight

→ Let $TE \subseteq E$ be the set of tree edges already added to MCST

→ Initially, $TE = \phi$

→ Scan E from e_0 to e_{m-1}
- If adding e_i to TE creates a loop, skip it
- Otherwise, add e_i to TE

EXAMPLE



Sort E as $\{(5,6), (1,2), (0,1), (4,5), (4,6), (0,2), (1,4), (2,3)\}$

Set $TE = \phi$

Add (5,6), (1,2), (0,1), (4,5); skip (4,6), (0,2); Add (1,4) (2,3)

$TE = \{(5,6), (1,2), (0,1), (4,5), (1,4), (2,3)\}$

CORRECTNESS OF KRUSKAL'S ALGORITHM

→ Minimum Separator Lemma
- Let V be partitioned into two non-empty sets U and ~~W~~ $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U$, $w \in W$

→ Every MCST must include e

- Edges in TE partition vertices into connected components
  → Initially each vertex is a separate component
- Adding e = (u, w) merges components of u and w
  → If u and w are in the same component, e forms a cycle & is discarded
- Let U be component of u, W be V\U
  → U, W form a partition of V with u ∈ U & w ∈ W
  → Since we are scanning edges in ascending order of cost, e is minimum cost edge connecting U and W, so it must be part of any MCST.

IMPLEMENTATION

- Collect edges in a list as (d, u, v)
  → Weight as first component for easy sorting
- Main challenge is to keep track of connected components
  → Dictionary to record component of each vertex
  → Initially each vertex is an isolated component
  → When we add an edge (u, v), merge the components of u & v

→ Analysis

→ Sorting the edges is $O(m \log m)$ since m is at most n^2 equivalently, $O(m \log n)$

→ Outerloop runs m times
- Each time we add a tree edge, we have to merge components - O(n) scan
- n-1 tree edges, so this is done O(n) times

→ Overall, $O(n^2)$

→ Complexity is $O(n^2)$

→ Bottleneck is naive strategy to label & merge components

→ Components partition vertices
- collection of disjoint sets

→ Data structure to maintain collection of disjoint sets
- find (v) - return set containing v
- union (u, v) - merge sets of u, v.

→ Efficient union-find brings complexity down to $O(m \log n)$

CODE:

```
def kruskal (WList):
    (edges, component, TE) = ([], {}, [])
    for u in WList.keys():
        edges.extend([(d,u,v) for (v,d) in WList[u]])
        component[u] = u
    edges.sort()

    for (d,u,v) in edges:
        if component[u] != component[v]:
```

Shrinath
PAGE NO.
DATE:

```
        TE.append((u,v))
        c = component[u]
        for w in Wlist.keys():
            if component[w] == c:
                component[w] = component[v]
    return(TE)
```

→ SUMMARY ←

- Kruskal's algorithm builds an MCST bottom up
  - → start with n components, each an isolated vertex.
  - → scan edges in ascending order of cost
  - → Whenever an edge merges disjoint components, add it to the MCST
- Correctness follows from Minimum separator lemma.
- Complexity is $O(n^2)$ due to naive handling of components
  - → will see how to improve to $O(m \log n)$
- If edge weights repeat, MCST is not unique
- "Choose minimum cost edge" will allow choices
  - consider a triangle on 3 vertices with all edges equal
- Different choices lead to different spanning trees
- In general, there may be a very large number of MCST