Recall that …

- BFS and DFS are two systematic ways to explore a graph
  - Both take time linear in the size of the graph with adjacency lists
- Recover paths by keeping parent information
- BFS can compute shortest paths, in terms of number of edges
- DFS numbering can reveal many interesting features

Shortest paths

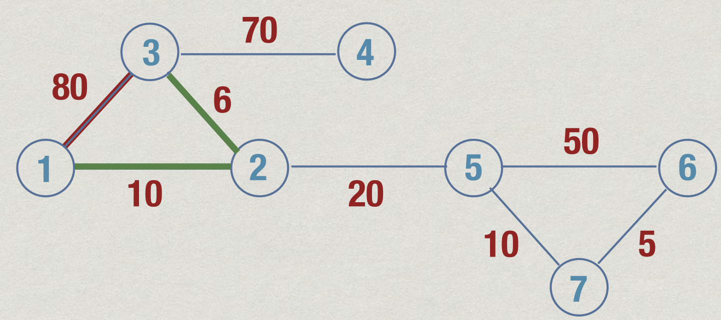
- **Weighted graph**
  - G=(V,E) together with
  - **Weight function,** w : E→Reals
- Let $e_1=(v_0,v_1)$, $e_2 = (v_1,v_2)$, …, $e_n = (v_{n-1},v_n)$ be a path from v_0 to v_n
- Cost of the path is $w(e_1) + w(e_2) + \ldots + w(e_n)$
- **Shortest path** from v_0 to v_n : minimum cost

Adding edge weights

- Label each edge with a number—**cost**
  - Ticket price on a flight sector
  - Tolls on highway segment
  - Distance travelled between two stations
  - Typical time between two locations during peak hour traffic

Shortest paths …

- BFS finds path with fewest number of edges
- In a weighted graph, need not be the shortest path

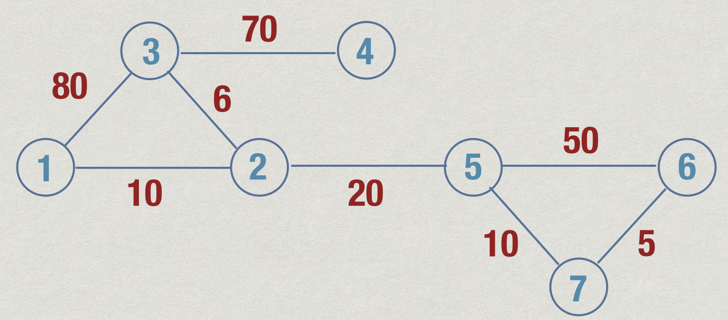


Shortest path problems

- **Single source**
  - Find shortest paths from some fixed vertex, say 1, 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

This lecture…

- Single source shortest paths
- For instance, shortest paths from 1 to 2,3,…,7



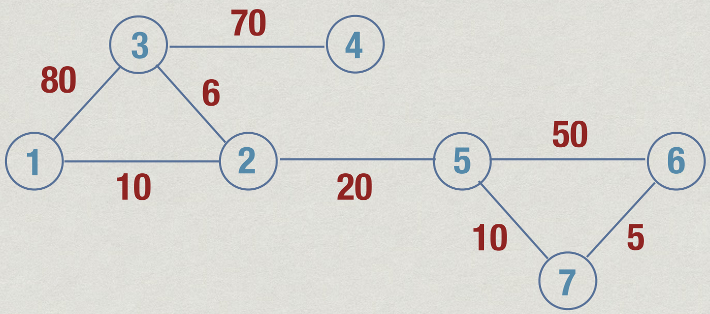
Shortest path problems

- **All pairs**
  - Find shortest paths between every pair of vertices i and j
  - Railway routes, shortest way to travel between any pair of cities

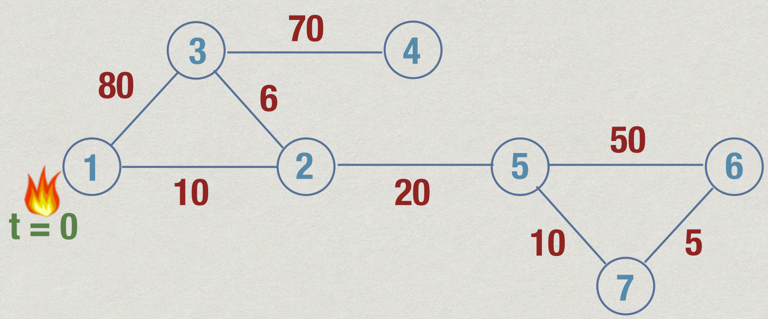
Single source shortest paths

- Imagine vertices are oil depots, edges are pipelines
- Set fire to oil depot at vertex 1
  - Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 1 is nearest vertex
- Next oil depot is second nearest vertex
- …

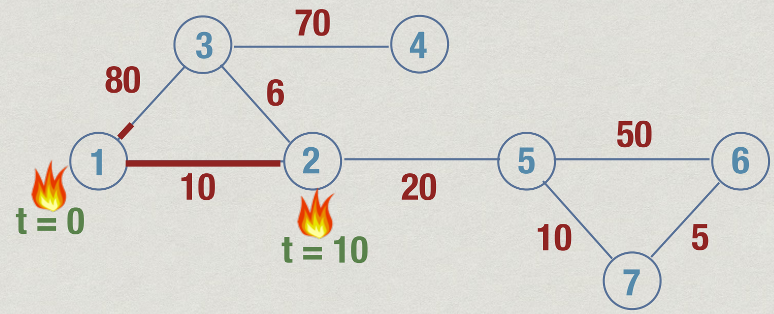
Single source shortest paths



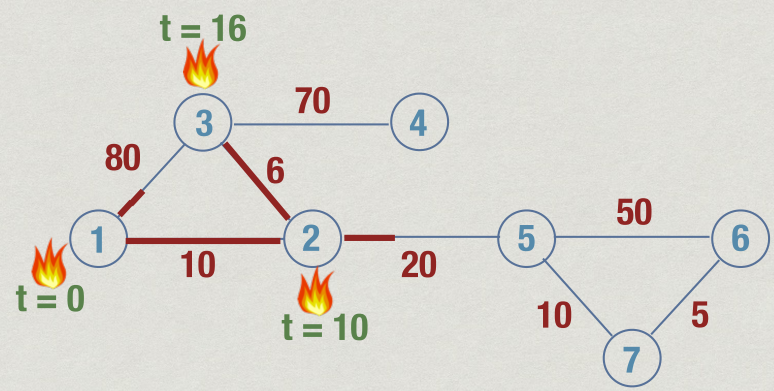
Single source shortest paths



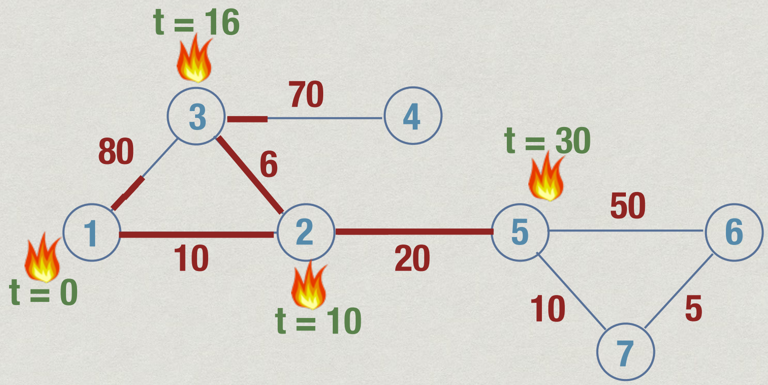
Single source shortest paths



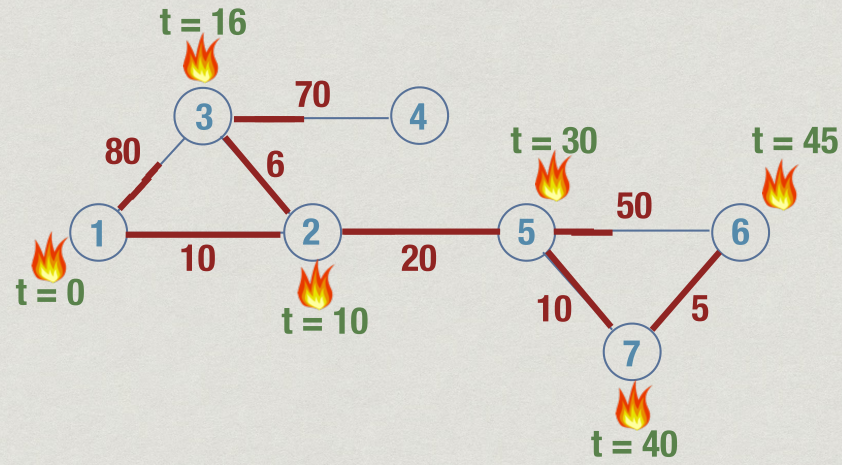
Single source shortest paths



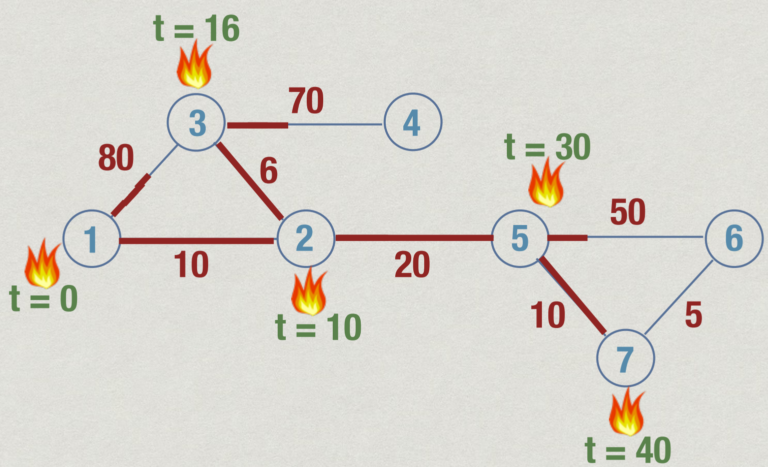
Single source shortest paths



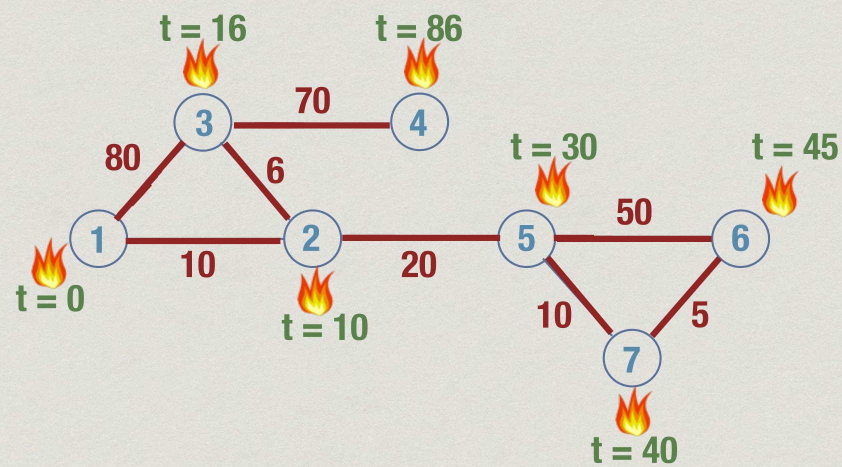
Single source shortest paths



Single source shortest paths

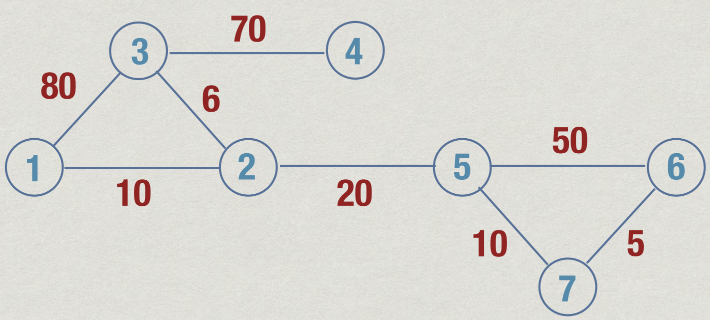


Single source shortest paths



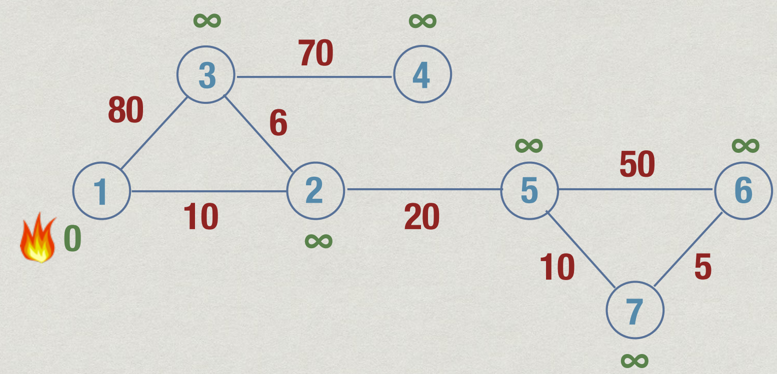
Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



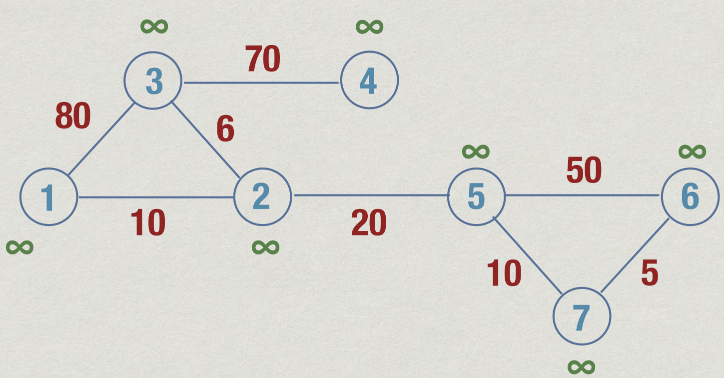
Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



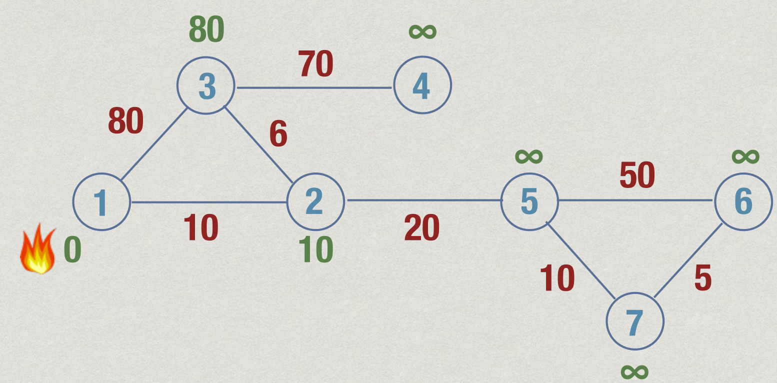
Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



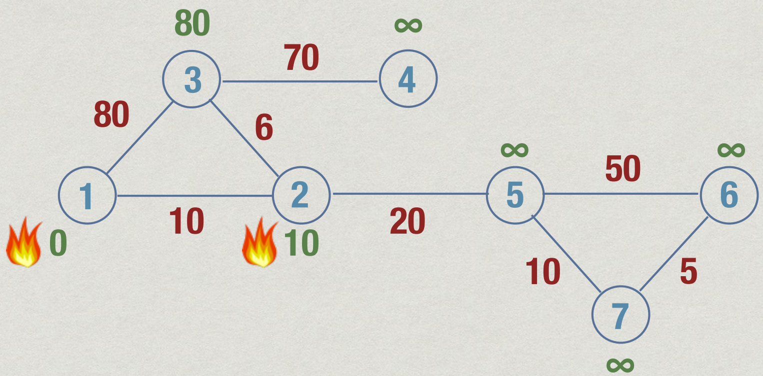
Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



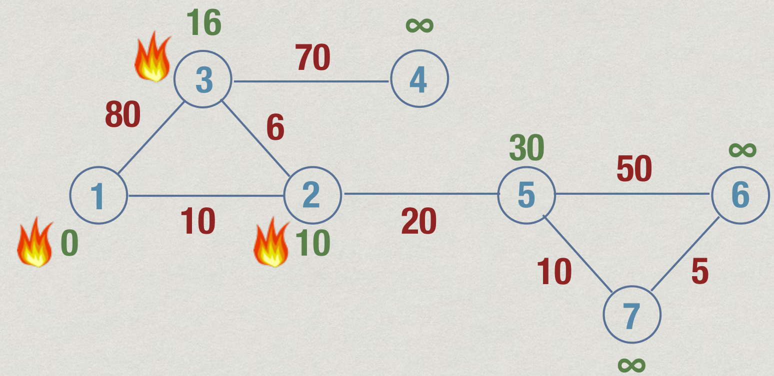
Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



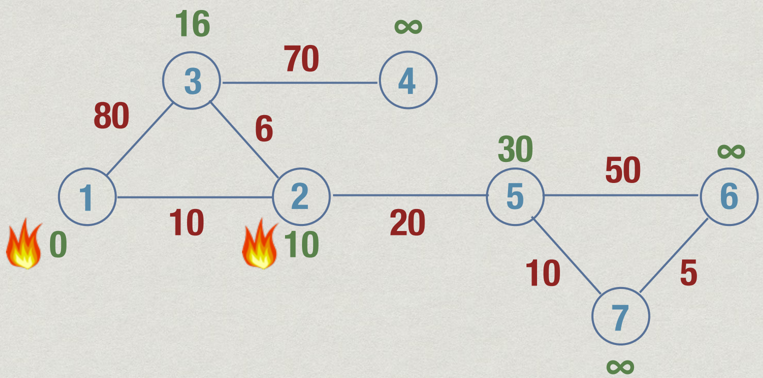
Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



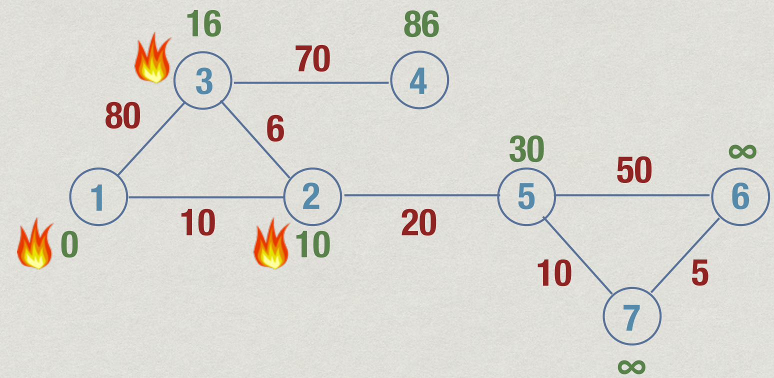
Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



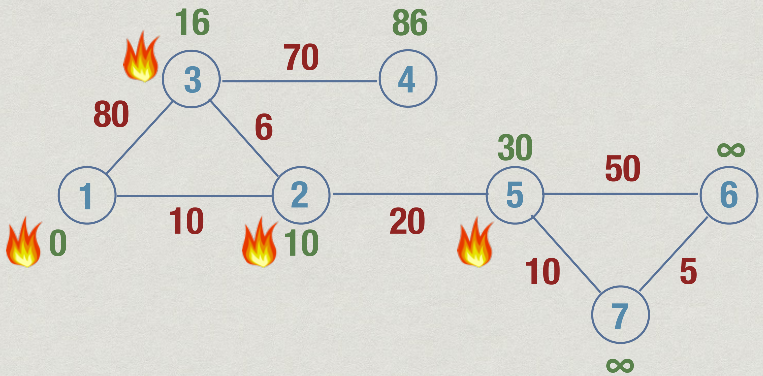
Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



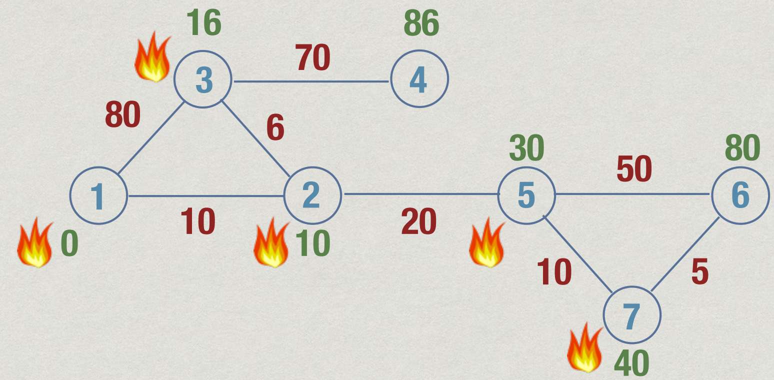
Single source shortest paths

* Compute expected time to burn of each vertex
* Update this each time a new vertex burns



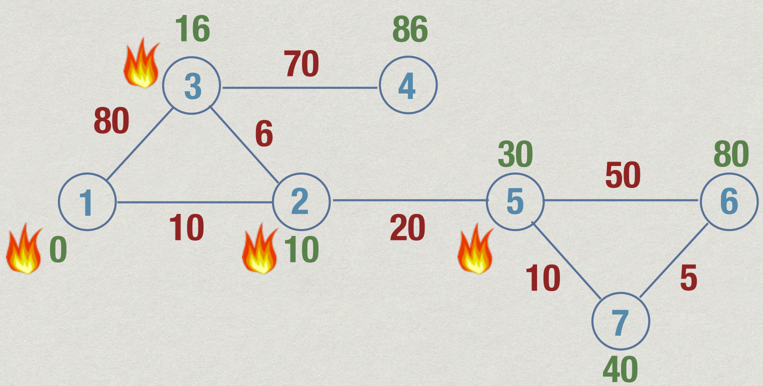
Single source shortest paths

* Compute expected time to burn of each vertex
* Update this each time a new vertex burns



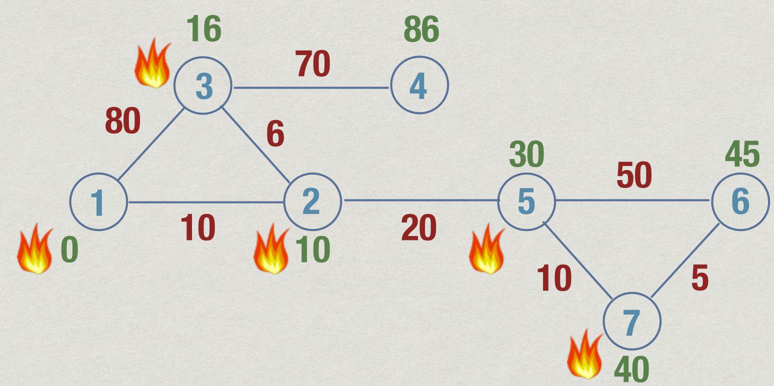
Single source shortest paths

* Compute expected time to burn of each vertex
* Update this each time a new vertex burns



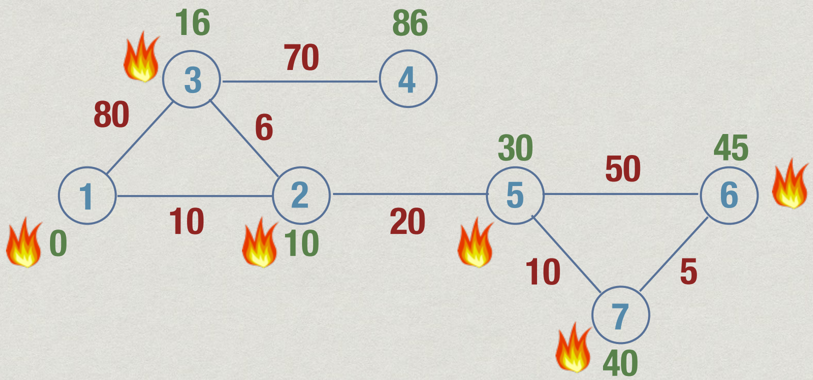
Single source shortest paths

* Compute expected time to burn of each vertex
* Update this each time a new vertex burns



Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns

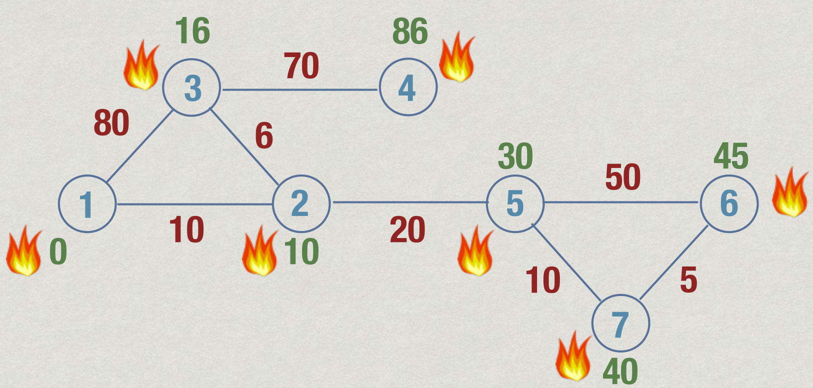


Algorithmically

- Maintain two arrays
  - `BurntVertices[ ]`, initially `False` for all i
  - `ExpectedBurnTime[ ]`, initially ∞ for all i
    - For ∞, use sum of all edge weights + 1
- Set `ExpectedBurnTime[1] = 0`
- Repeat, until all vertices are burnt
  - Find `j` with minimum `ExpectedBurnTime`
  - Set `BurntVertices[j] = True`
  - Recompute `ExpectedBurnTime[k]` for each neighbour `k` of `j`

Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



Dijkstra's algorithm

```
function ShortestPaths(s){ // assume source is s
  for i = 1 to n
    BV[i] = False; EBT[i] = infinity

  EBT[s] = 0

  for i = 1 to n
    Choose u such that BV[u] == False
                         and EBT[u] is minimum
    BV[u] = True
    for each edge (u,v) with BV[v] == False
      if EBT[v] > EBT[u] + weight(u,v)
        EBT[v] = EBT[u] + weight(u,v)
```

Dijkstra's algorithm

```
function ShortestPaths(s){ // assume source is s
 for i = 1 to n
  Visited[i] = False; Distance[i] = infinity

 Distance[s] = 0

 for i = 1 to n
  Choose u such that Visited[u] == False
                 and Distance[u] is minimum
  Visited[u] = True
  for each edge (u,v) with Visited[v] == False
   if Distance[v] > Distance[u] + weight(u,v)
    Distance[v] = Distance[u] + weight(u,v)
```

Greedy algorithms

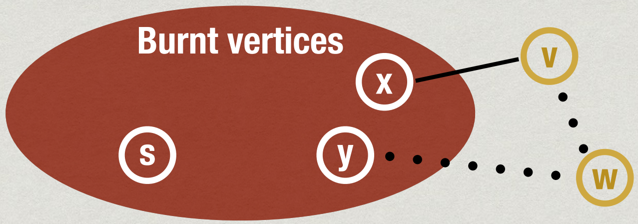
- Algorithm makes a sequence of choices
- Next choice is based on "current best value"
  - Never go back and change a choice
- Dijkstra's algorithm is greedy
  - Select vertex with minimum expected burn time
- Need to prove that greedy strategy is optimal
- Most times, greedy approach fails
  - Current best choice may not be globally optimal

Dijkstra's algorithm

- Maintain two arrays
  - `Visited[ ]`, initially `False` for all i
  - `Distance[ ]`, initially ∞ for all i
    - For ∞, use sum of all edge weights + 1
- Set `Distance[1] = 0`
- Repeat, until all vertices are burnt
  - Find `j` with minimum `Distance`
  - Set `Visited[j] = True`
  - Recompute `Distance[k]` for each neighbour `k` of `j`

Correctness

- Each new shortest path we discover extends an earlier one
- By induction, assume we have identified shortest paths to all vertices already burnt



- Next vertex to burn is v, via x
- Cannot later find a shorter path from y to w to v

Dijkstra's algorithm

```
function ShortestPaths(s){ // assume source is s
 for i = 1 to n
  Visited[i] = False; Distance[i] = infinity

 Distance[s] = 0

 for i = 1 to n
  Choose u such that Visited[u] == False
                  and Distance[u] is minimum
  Visited[u] = True
  for each edge (u,v) with Visited[v] == False
   if Distance[v] > Distance[u] + weight(u,v)
    Distance[v] = Distance[u] + weight(u,v)
```

Complexity

- Does adjacency list help?
  - Scan neighbours to update burn times
  - O(m) across all iterations
- However, identifying minimum burn time vertex still takes O(n) in each iteration
- Still $O(n^2)$

Complexity

- Outer loop runs n times
  - In each iteration, we burn one vertex
  - O(n) scan to find minimum burn time vertex
- Each time we burn a vertex v, we have to scan all its neighbours to update burn times
  - O(n) scan of adjacency matrix to find all neighbours
- Overall $O(n^2)$

Complexity

- Can maintain `ExpectedBurnTime` in a more sophisticated data structure
  - Different types of trees (heaps, red-black trees) allow both of the following in O(log n) time
    - find and delete minimum
    - insert or update a value

Complexity

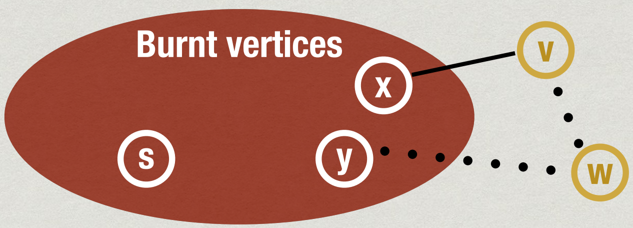
- With such a tree
  - Finding minimum burn time vertex takes O(log n)
  - With adjacency list, updating burn times take O(log n) each, total O(m) edges
- Overall O(n log n + m log n) = O((n+m) log n)

Why negative weights?

- Weights represent money
  - Taxi driver earns money from airport to city, travels empty to next pick-up point
  - Some segments earn money, some lose money
- Chemistry
  - Nodes are compounds, edges are reactions
  - Weights are energy absorbed/released by reaction

Limitations

- What if edge weights can be negative?
- Our correctness argument is no longer valid



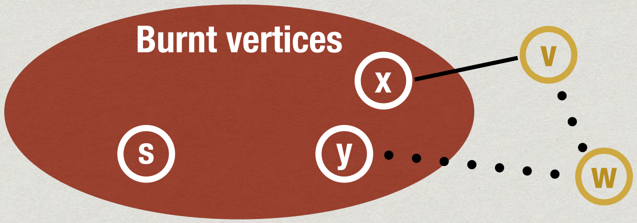
- Next vertex to burn is v, via x
- Might find a shorter path later with negative weights from y to w to v

Handling negative edges

- **Negative cycle**: loop with a negative total weight
  - Problem is not well defined with negative cycles
  - Repeatedly traversing cycle pushes down cost without a bound
- With negative edges, but no negative cycles, other algorithms exist (will see later)
  - Bellman-Ford
  - Floyd-Warshall all pairs shortest path

Correctness for Dijsktra's algorithm

- By induction, assume we have identified shortest paths to all vertices already burnt



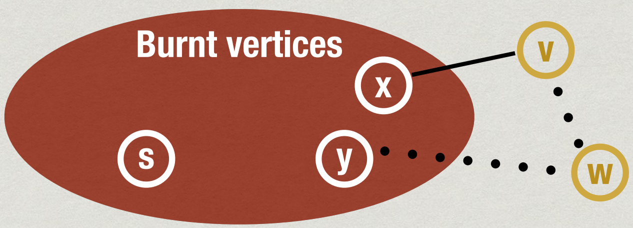
- Next vertex to burn is v, via x
- Cannot later find a shorter path from y to w to v

Negative weights …

- **Negative cycle**: loop with a negative total weight
  - Problem is not well defined with negative cycles
  - Repeatedly traversing cycle pushes down cost without a bound
- With negative edges, but no negative cycles, shortest paths do exist

Negative weights

- Our correctness argument is no longer valid



- Next vertex to burn is v, via x
- Might find a shorter path later with negative weights from y to w to v

About shortest paths

- Shortest paths will never loop
  - Never visit the same vertex twice
  - At most length n-1
- Every prefix of a shortest path is itself a shortest path
  - Suppose the shortest path from s to t is

$$s \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \ldots \rightarrow v_m \rightarrow t$$

  - Every prefix $s \rightarrow v_1 \rightarrow \ldots \rightarrow v_r$ is a shortest path to v_r

Updating Distance( )

- When vertex j is "burnt", for each edge (j,k) update

  Distance(k) = min(Distance(k), Distance(j)+weight(j,k))

- Refer to this as update(j,k)
- Dijkstra's algorithm
  - When we compute update(j,k), Distance(j) is always guaranteed to be correct distance to j
- What can we say in general?

Updating Distance( ) …

update(j,k):
  Distance(k) = min(Distance(k), Distance(j)+weight(j,k))

- Dijkstra's algorithm performs a particular "greedy" sequence of updates
  - Computes shortest paths without negative weights
- With negative edges, this sequence does not work
- Is there some sequence that does work?

Properties of update(j,k)

update(j,k):
  Distance(k) = min(Distance(k), Distance(j)+weight(j,k))

- Distance(k) is no more than Distance(j)+weight(j,k)
- If Distance(j) is correct and j is the second-last node on shortest path to k, Distance(k) is correct
- Update is safe
  - Distance(k) never becomes "too small"
  - Redundant updates cannot hurt

Updating distance( ) …

- Suppose the shortest path from s to t is

$$s \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \ldots \rightarrow v_m \rightarrow t$$

- If our update sequence includes …,update(s,v_1), …,update(v_1,v_2),…,update(v_2,v_3),…,update(v_m,t),…, in that order, Distance(t) will be computed correctly
  - If Distance(j) is correct and j is the second-last node on shortest path to k, Distance(k) is correct after update(j,k)

Bellman-Ford algorithm

- Initialize Distance(s) = 0, Distance(u) = ∞ for all other vertices
- Update all edges n-1 times!

Bellman-Ford algorithm

- Initialize Distance(s) = 0, Distance(u) = ∞ for all other vertices
- Update all edges n-1 times!

Iteration 1	Iteration 2
…	…
update(s,v_1)	update(s,v_1)
…	…
update(v_1,v_2)	update(v_1,v_2)
…	…
update(v_2,v_3)	update(v_2,v_3)
…	…
update(v_m,t)	update(v_m,t)
…	…

Bellman-Ford algorithm

- Initialize Distance(s) = 0, Distance(u) = ∞ for all other vertices
- Update all edges n-1 times!

Iteration 1
…
update(s,v_1)
…
update(v_1,v_2)
…
update(v_2,v_3)
…
update(v_m,t)
…

Bellman-Ford algorithm

- Initialize Distance(s) = 0, Distance(u) = ∞ for all other vertices
- Update all edges n-1 times!

Iteration 1	Iteration 2	…
…	…	…
update(s,v_1)	update(s,v_1)	…
…	…	…
update(v_1,v_2)	update(v_1,v_2)	…
…	…	…
update(v_2,v_3)	update(v_2,v_3)	…
…	…	…
update(v_m,t)	update(v_m,t)	…
…	…	…

Bellman-Ford algorithm

- Initialize Distance(s) = 0, Distance(u) = ∞ for all other vertices
- Update all edges n-1 times!

Iteration 1	Iteration 2	…	Iteration n-1
…	…	…	…
update(s,v_1)	update(s,v_1)	…	update(s,v_1)
…	…	…	…
update(v_1,v_2)	update(v_1,v_2)	…	update(v_1,v_2)
…	…	…	…
update(v_2,v_3)	update(v_2,v_3)	…	update(v_2,v_3)
…	…	…	…
update(v_m,t)	update(v_m,t)	…	update(v_m,t)
…	…	…	…

Bellman-Ford algorithm

- Initialize Distance(s) = 0, Distance(u) = ∞ for all other vertices
- Update all edges n-1 times!

Iteration 1	Iteration 2	…	Iteration n-1
…	…	…	…
update(s,v_1)	update(s,v_1)	…	update(s,v_1)
…	…	…	…
update(v_1,v_2)	**update(v_1,v_2)**	…	update(v_1,v_2)
…	…	…	…
update(v_2,v_3)	update(v_2,v_3)	…	update(v_2,v_3)
…	…	…	…
update(v_m,t)	update(v_m,t)	…	update(v_m,t)
…	…	…	…

Bellman-Ford algorithm

- Initialize Distance(s) = 0, Distance(u) = ∞ for all other vertices
- Update all edges n-1 times!

Iteration 1	Iteration 2	…	Iteration n-1
…	…	…	…
update(s,v_1)	update(s,v_1)	…	update(s,v_1)
…	…	…	…
update(v_1,v_2)	update(v_1,v_2)	…	update(v_1,v_2)
…	…	…	…
update(v_2,v_3)	update(v_2,v_3)	…	update(v_2,v_3)
…	…	…	…
update(v_m,t)	update(v_m,t)	…	update(v_m,t)
…	…	…	…

Bellman-Ford algorithm

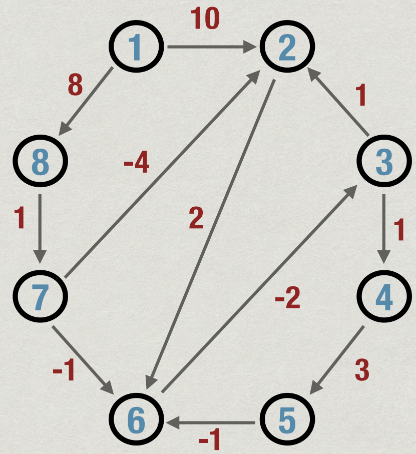
- Initialize Distance(s) = 0, Distance(u) = ∞ for all other vertices
- Update all edges n-1 times!

Iteration 1	Iteration 2	…	Iteration n-1
…	…	…	…
update(s,v_1)	update(s,v_1)	…	update(s,v_1)
…	…	…	…
update(v_1,v_2)	**update(v_1,v_2)**	…	update(v_1,v_2)
…	…	…	…
update(v_2,v_3)	update(v_2,v_3)	…	update(v_2,v_3)
…	…	…	…
update(v_m,t)	update(v_m,t)	…	**update(v_m,t)**
…	…	…	…

Bellman-Ford algorithm

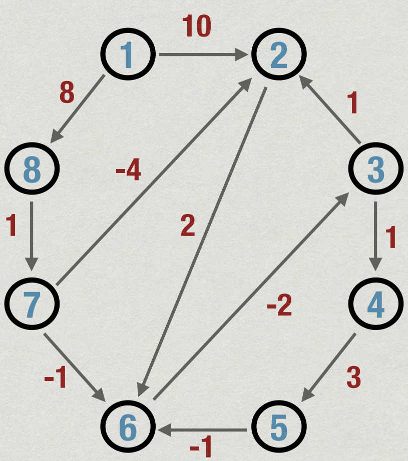
```
function BellmanFord(s)//source s, with -ve weights

for i = 1 to n
 Distance[i] = infinity

Distance[s] = 0

for i = 1 to n-1 //repeat n-1 times
 for each edge(j,k) in E
  Distance(k) = min(Distance(k),
                    Distance(j) + weight(j,k))
```

Example

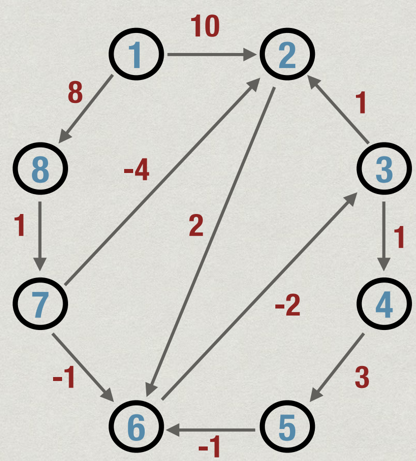


Node
1
2
3
4
5
6
7
8

Example



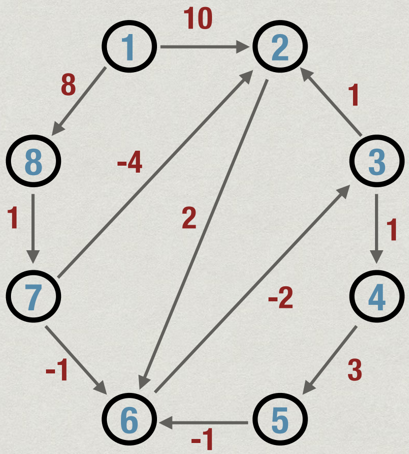
Example



Iteration

Node	0
1	0
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞
8	∞

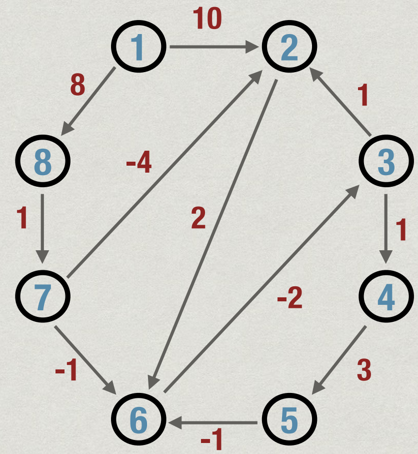
Example



Iteration

Node	0	1
1	0	0
2	∞	10
3	∞	∞
4	∞	∞
5	∞	∞
6	∞	∞
7	∞	∞
8	∞	8

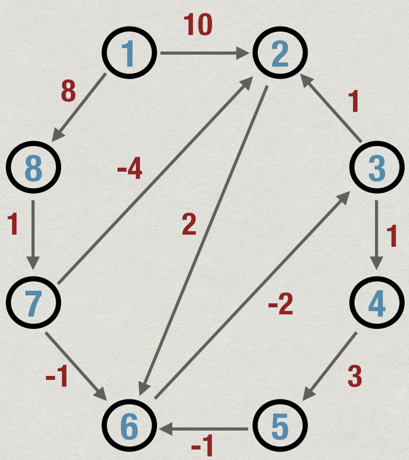
Example



Iteration

Node	0	1	2	3
1	0	0	0	0
2	∞	10	10	5
3	∞	∞	∞	10
4	∞	∞	∞	∞
5	∞	∞	∞	∞
6	∞	∞	12	8
7	∞	∞	9	9
8	∞	8	8	8

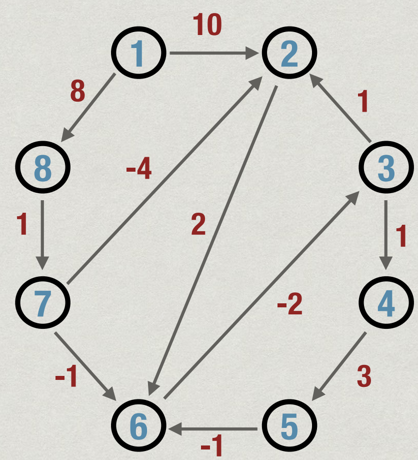
Example



Iteration

Node	0	1	2
1	0	0	0
2	∞	10	10
3	∞	∞	∞
4	∞	∞	∞
5	∞	∞	∞
6	∞	∞	12
7	∞	∞	9
8	∞	8	8

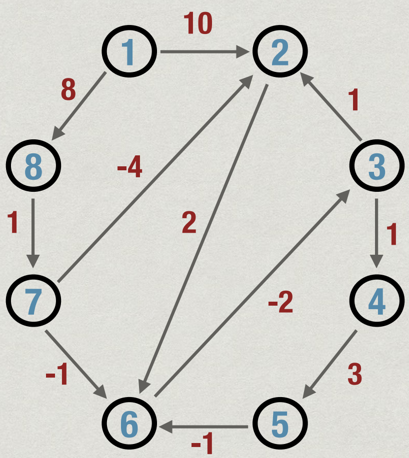
Example



Iteration

Node	0	1	2	3	4
1	0	0	0	0	0
2	∞	10	10	5	5
3	∞	∞	∞	10	6
4	∞	∞	∞	∞	11
5	∞	∞	∞	∞	∞
6	∞	∞	12	8	7
7	∞	∞	9	9	9
8	∞	8	8	8	8

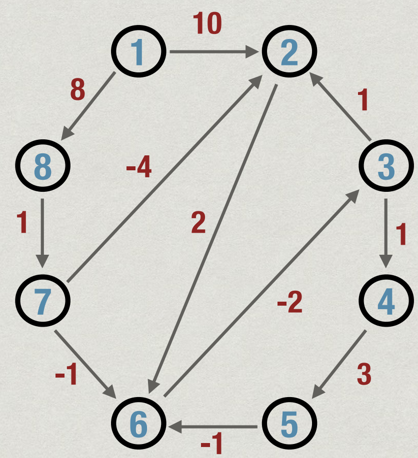
Example



Iteration

Node	0	1	2	3	4	5
1	0	0	0	0	0	0
2	∞	10	10	5	5	5
3	∞	∞	∞	10	6	5
4	∞	∞	∞	∞	11	7
5	∞	∞	∞	∞	∞	14
6	∞	∞	12	8	7	7
7	∞	∞	9	9	9	9
8	∞	8	8	8	8	8

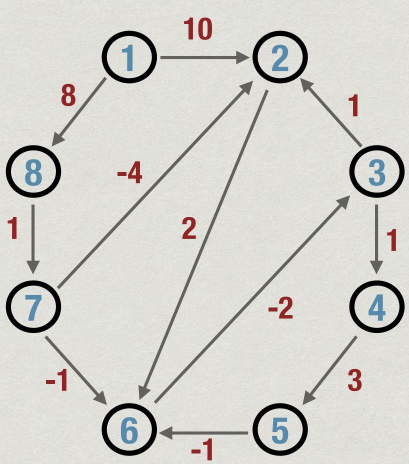
Example



Iteration

Node	0	1	2	3	4	5	6	7
1	0	0	0	0	0	0	0	0
2	∞	10	10	5	5	5	5	5
3	∞	∞	∞	10	6	5	5	5
4	∞	∞	∞	∞	11	7	6	6
5	∞	∞	∞	∞	∞	14	10	9
6	∞	∞	12	8	7	7	7	7
7	∞	∞	9	9	9	9	9	9
8	∞	8	8	8	8	8	8	8

Example



Iteration

Node	0	1	2	3	4	5	6
1	0	0	0	0	0	0	0
2	∞	10	10	5	5	5	5
3	∞	∞	∞	10	6	5	5
4	∞	∞	∞	∞	11	7	6
5	∞	∞	∞	∞	∞	14	10
6	∞	∞	12	8	7	7	7
7	∞	∞	9	9	9	9	9
8	∞	8	8	8	8	8	8

Complexity

- Outer loop runs n times
- In each loop, for each edge (j,k), we run update(j,k)
  - Adjacency matrix — $O(n^2)$ to identify all edges
  - Adjacency list — $O(m)$
- Overall
  - Adjacency matrix — $O(n^3)$
  - Adjacency list — $O(mn)$

Weighted graphs

- Negative weights are allowed, but not negative cycles
- Shortest paths are still well defined
- Bellman-Ford algorithm computes single-source shortest paths
- Can we compute shortest paths between all pairs of vertices?

Inductively exploring shortest paths

- Simplest shortest path from i to j is a direct edge (i,j)
- General case:

$$i \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \ldots \rightarrow v_m \rightarrow j$$

  - All of $\{v_1, v_2, v_3 \ldots, v_m\}$ are distinct, and different from i and j
  - Restrict what vertices can appear in this set

About shortest paths

- Shortest paths will never loop
  - Never visit the same vertex twice
  - At most length n-1
- Use this to inductively explore all possible shortest paths efficiently

Inductively exploring shortest paths …

- Recall that $V = \{1,2,\ldots,n\}$
- $W^k(i,j)$: weight of shortest path from i to j among paths that only go via $\{1,2,\ldots,k\}$
  - $\{k+1,\ldots,n\}$ cannot appear on the path
  - i, j themselves need not be in $\{1,2,\ldots,k\}$
- $W^0(i,j)$: direct edges
  - $\{1,2,\ldots,n\}$ cannot appear between i and j

Inductively exploring shortest paths …

- From $W^{k-1}(i,j)$ to $W^{k}(i,j)$
  - Case 1: Shortest path via {1,2,…,k} does not use vertex k
    - $W^{k}(i,j) = W^{k-1}(i,j)$
  - Case 2: Shortest path via {1,2,…,k} does go via k
    - k can appear only once along this path
    - Break up as paths i to k and k to j, each via {1,2,…,k-1}
    - $W^{k}(i,j) = W^{k-1}(i,k) + W^{k-1}(k,j)$
- Conclusion: $W^{k}(i,j) = \min(W^{k-1}(i,j)\ ,\ W^{k-1}(i,k) + W^{k-1}(k,j))$

Floyd-Warshall algorithm

- W^0 is adjacency matrix with edge weights
  - $W^0[i][j]$ = weight(i,j) if there is an edge (i,j),
    ∞, otherwise
- For k in 1,2,…,n
  - Compute $W^k(i,j)$ from $W^{k-1}(i,j)$ using
    $W^{k}(i,j) = \min(W^{k-1}(i,j)\ ,\ W^{k-1}(i,k) + W^{k-1}(k,j))$
- W^n contains weights of shortest paths for all pairs

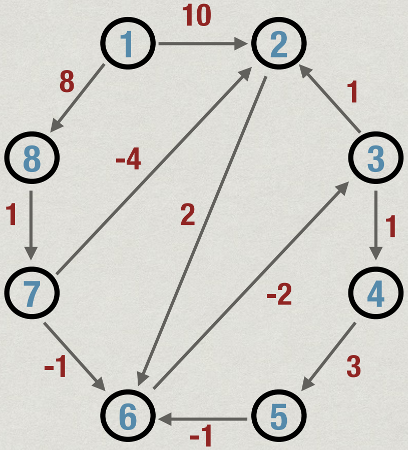
Floyd-Warshall algorithm

- W^0 is adjacency matrix with edge weights
  - $W^0[i][j]$ = weight(i,j) if there is an edge (i,j),
    ∞, otherwise
- For k in 1,2,…,n
  - Compute $W^k(i,j)$ from $W^{k-1}(i,j)$ using
    $W^{k}(i,j) = \min(W^{k-1}(i,j)\ ,\ W^{k-1}(i,k) + W^{k-1}(k,j))$
- W^n contains weights of shortest paths for all pairs

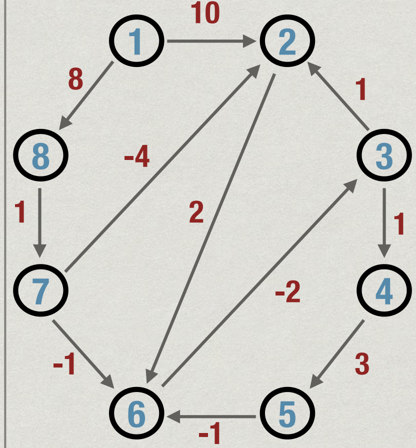
Floyd-Warshall algorithm

```
function FloydWarshall

for i = 1 to n
 for j = 1 to n
   W[i][j][0] = infinity

for each edge (i,j) in E
  W[i][j][0] = weight(i,j)

for k = 1 to n
  for i = 1 to n
    for j = 1 to n
      W[i][j][k] = min(W[i][j][k-1],
                       W[i][k][k-1] + W[k][j][k-1])
```

Example



Example



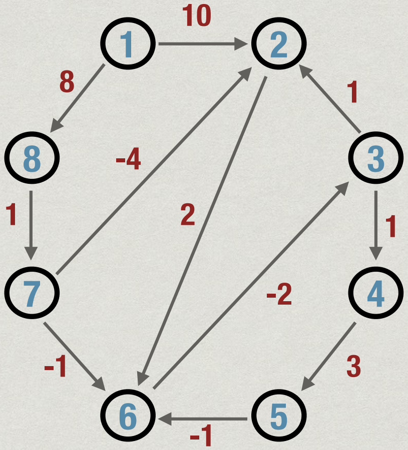
W^0

	1	2	3	4	5	6	7	8
1	∞	10	∞	∞	∞	∞	∞	8
2	∞	∞	∞	∞	∞	2	∞	∞
3	∞	1	∞	1	∞	∞	∞	∞
4	∞	∞	∞	∞	3	∞	∞	∞
5	∞	∞	∞	∞	∞	-1	∞	∞
6	∞	∞	-2	∞	∞	∞	∞	∞
7	∞	-4	∞	∞	∞	-1	∞	∞
8	∞	∞	∞	∞	∞	∞	1	∞

W^1

	1	2	3	4	5	6	7	8
1	∞	10	∞	∞	∞	∞	∞	8
2	∞	∞	∞	∞	∞	2	∞	∞
3	∞	1	∞	1	∞	∞	∞	∞
4	∞	∞	∞	∞	3	∞	∞	∞
5	∞	∞	∞	∞	∞	-1	∞	∞
6	∞	∞	-2	∞	∞	∞	∞	∞
7	∞	-4	∞	∞	∞	-1	∞	∞
8	∞	∞	∞	∞	∞	∞	1	∞

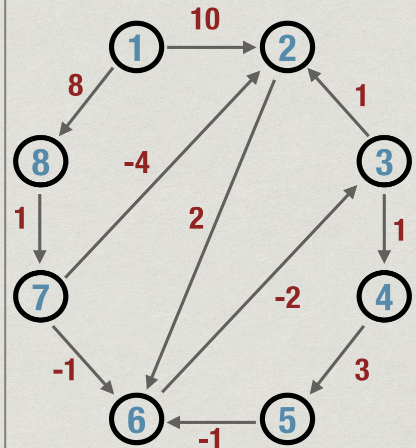
Example



W^0

	1	2	3	4	5	6	7	8
1	∞	10	∞	∞	∞	∞	∞	8
2	∞	∞	∞	∞	∞	2	∞	∞
3	∞	1	∞	1	∞	∞	∞	∞
4	∞	∞	∞	∞	3	∞	∞	∞
5	∞	∞	∞	∞	∞	-1	∞	∞
6	∞	∞	-2	∞	∞	∞	∞	∞
7	∞	-4	∞	∞	∞	-1	∞	∞
8	∞	∞	∞	∞	∞	∞	1	∞

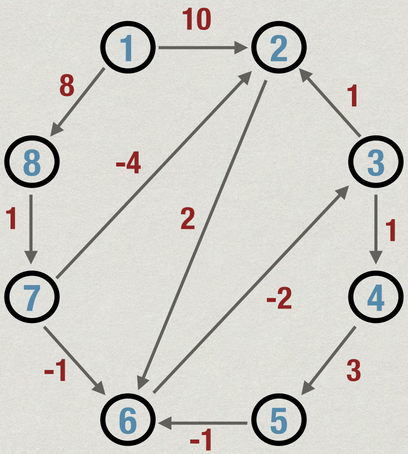
Example



W^1

	1	2	3	4	5	6	7	8
1	∞	10	∞	∞	∞	∞	∞	8
2	∞	∞	∞	∞	∞	2	∞	∞
3	∞	1	∞	1	∞	∞	∞	∞
4	∞	∞	∞	∞	3	∞	∞	∞
5	∞	∞	∞	∞	∞	-1	∞	∞
6	∞	∞	-2	∞	∞	∞	∞	∞
7	∞	-4	∞	∞	∞	-1	∞	∞
8	∞	∞	∞	∞	∞	∞	1	∞

Example



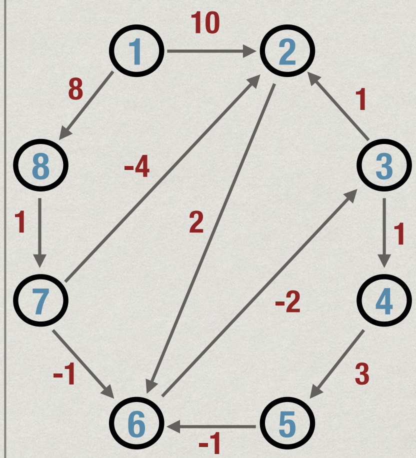
W^2

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

W^1

	1	2	3	4	5	6	7	8
1	∞	10	∞	∞	∞	∞	∞	8
2	∞	∞	∞	∞	∞	2	∞	∞
3	∞	1	∞	1	∞	∞	∞	∞
4	∞	∞	∞	∞	3	∞	∞	∞
5	∞	∞	∞	∞	∞	-1	∞	∞
6	∞	∞	-2	∞	∞	∞	∞	∞
7	∞	-4	∞	∞	∞	-1	∞	∞
8	∞	∞	∞	∞	∞	∞	1	∞

Example



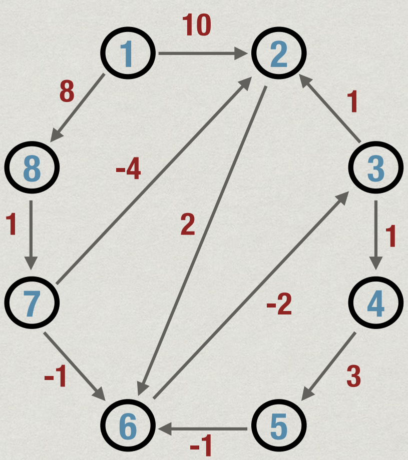
W^2

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

W^3

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

Example



W^2

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

Complexity

- Easy to see that the complexity is $O(n^3)$
  - n iterations
  - In each iteration, we update n^2 entries
- A word about space complexity
  - Naive implementation is $O(n^3)$—W[i][j][k]
  - Only need two “slices” at a time, W[i][j][k-1] and W[i][j][k]
  - Space requirement reduces to $O(n^2)$

Historical remarks

- Floyd-Warshall is a hybrid name
- Warshall originally proposed an algorithm for transitive closure
  - Generating path matrix P[i][j] from adjacency matrix A[i][j]
- Floyd adapted it to compute shortest paths

Inductively computing P[i][j]

- From $P^{k-1}(i,j)$ to $P^k(i,j)$
  - Case 1: There is a path from i to j without using vertex k
    - $P^k(i,j) = P^{k-1}(i,j)$
  - Case 2: Path via {1,2,…,k} does go via k
    - k can appear only once along this path
    - Break up as paths i to k and k to j, each via {1,2,…,k-1}
    - $P^k(i,j) = P^{k-1}(i,k)$ and $P^{k-1}(k,j)$
- Conclusion: $P^k(i,j) = P^{k-1}(i,j)$ or ($P^{k-1}(i,k)$ and $P^{k-1}(k,j)$)

Computing paths

- A(i,j) = 1 iff there is an edge from i to j
- Want P(i,j) = 1 iff there is a path from i to j
- Iteratively compute $P^k(i,j) = 1$ iff there is a path from i to j where all intermediate vertices are in {1,2,…,k}
  - {k+1,…,n} cannot appear on the path
  - i, j themselves need not be in {1,2,…,k}
- $P^0(i,j) = A(i,j)$: direct edges
  - {1,2,…,n} cannot appear between i and j

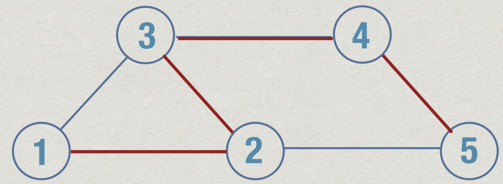
Warshall's algorithm

```
function Warshall

for i = 1 to n
 for j = 1 to n
   P[i][j][0] = False

for each edge (i,j) in E
  P[i][j][0] = True

for k = 1 to n
  for i = 1 to n
    for j = 1 to n
      P[i][j][k] = P[i][j][k-1] or
                   (P[i][k][k-1] and P[k][j][k-1])
```

Example: Road network

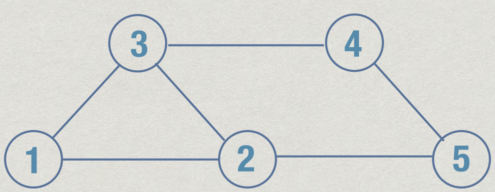
- District hit by a cyclone, damaging the roads
- Government sets to work to restore the roads
- Priority is to ensure that all parts of the district can be reached
- What set of roads should be restored first?

Spanning tree



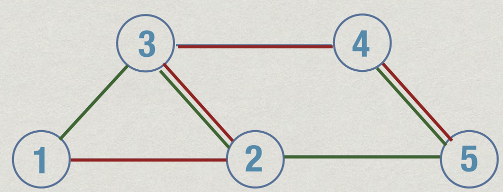
- Minimum connectivity: no loops
  - Removing an edge from a loop cannot disconnect graph
- Connected acyclic graph — tree
- Spanning tree

Spanning tree



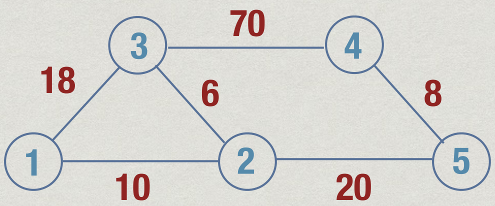
- Minimum connectivity: no loops
  - Removing an edge from a loop cannot disconnect graph
- Connected acyclic graph — tree
- Spanning tree

Spanning tree



- Minimum connectivity: no loops
  - Removing an edge from a loop cannot disconnect graph
- Connected acyclic graph — tree
- Spanning tree

Spanning tree with costs

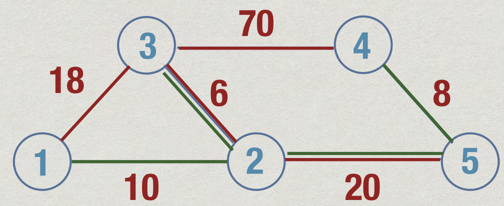


- Restoration of each road has a cost
- Among the different spanning trees, choose the one with minimum cost
- Minimum cost spanning tree

Spanning tree with costs

Cost = 114

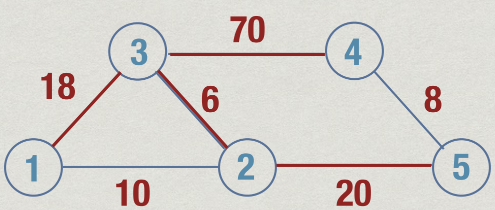
Cost = 44



- Restoration of each road has a cost
- Among the different spanning trees, choose the one with minimum cost
- Minimum cost spanning tree

Spanning tree with costs

Cost = 114



- Restoration of each road has a cost
- Among the different spanning trees, choose the one with minimum cost
- Minimum cost spanning tree

Facts about trees

Definition: A tree is a connected acyclic graph

Fact 1: A tree on n vertices has exactly n-1 edges

- Start with a tree and delete edges
- Initially one single component
- Deleting an edge must split a component into two
- After n-1 edge deletions, n components, each an isolated vertex

Facts about trees

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 p from i to j
- New edge (i,j) plus path p creates a cycle

Facts about trees

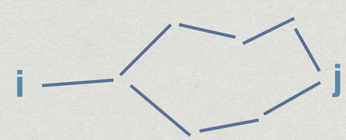
Any two of the following facts about a graph G implies the third

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

Facts about trees

Fact 3: In a tree, every pair of nodes is connected by a unique path

- If there are two paths from i to j, there must be a cycle



Building a minimum cost spanning trees

Two natural strategies

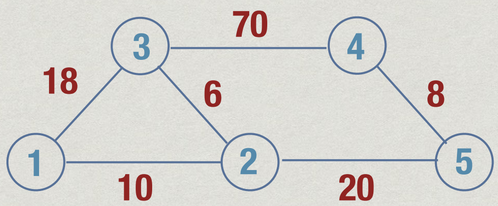
- Start with smallest edge and grow it into a tree

  Prim's Algorithm

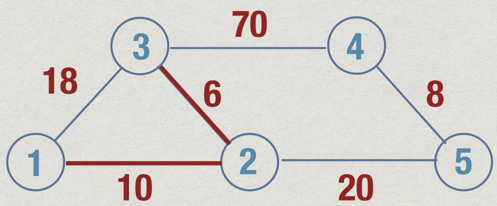
- Scan edges in ascending order of cost and connect components to form a tree

  Kruskal's Algorithm

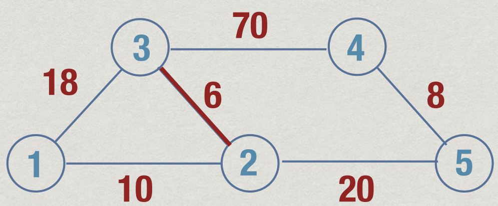
Prim’s algorithm



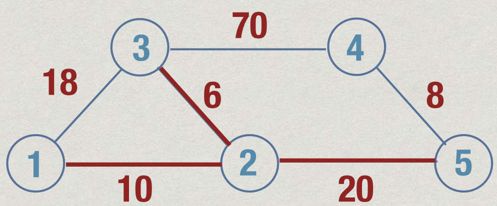
Prim’s algorithm



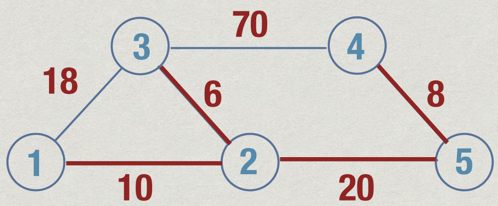
Prim’s algorithm



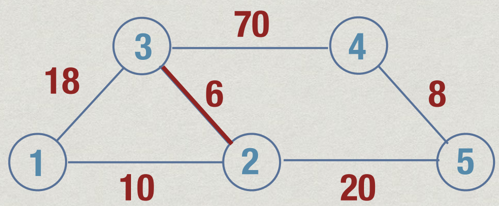
Prim’s algorithm



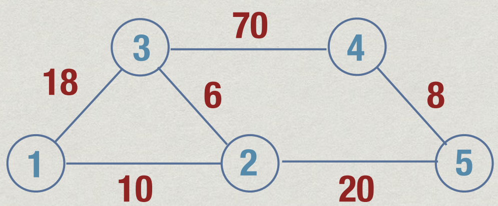
Prim’s algorithm



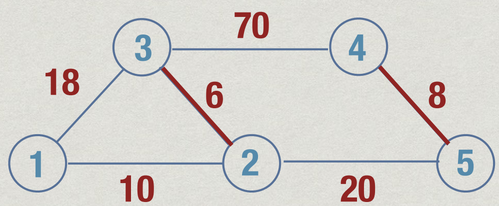
Kruskal’s algorithm



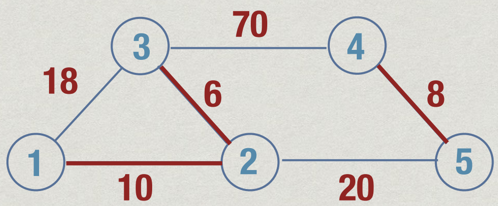
Kruskal’s algorithm



Kruskal’s algorithm



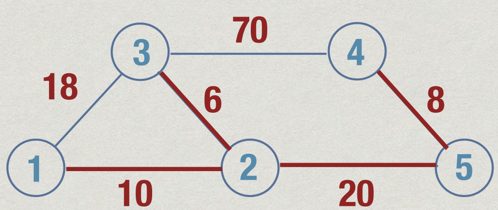
Kruskal's algorithm



Spanning tree

- Weighted undirected graph, G = (V,E,w)
  - Assume G is connected
- Identify a spanning tree with minimum weight
  - Tree connecting all vertices in V
- **Strategy 1:**
  - Start with minimum cost edge
  - Keep extending the tree with smallest edge

Kruskal's algorithm



Prim's algorithm

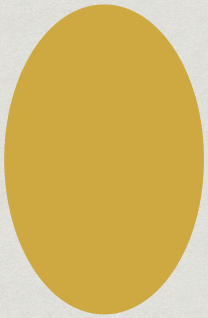
```
algorithm Prim_V1

Let e = (i,j) be minimum cost edge in E

TE = [e]      //List of edges in tree
TV = [i,j]    //List of vertices connected by tree

for i = 3 to n
  choose edge f = (u,v) of minimum cost
    such that u in TV and v not in TV
  TE.append(f)
  TV.append(v)

return(TE)
```

Correctness

- Prim’s algorithm is a greedy algorithm
  - Like Dijkstra’s single source shortest path
- A local heuristic is used to decide which edge to add next to the tree
- Choices made are never reconsidered
- Why does this sequence of local choices achieve a global optimum?

Minimum separator lemma

- Let T be a minimum cost spanning tree, e = (u,w) not in T
- u in U and w in W are connected by a path p in T
  - p starts in U and ends in W
  - Let f = (u’,w’) be the first edge on p such that u’ in U and w’ in W
  - Drop f and add e to get a smaller spanning tree

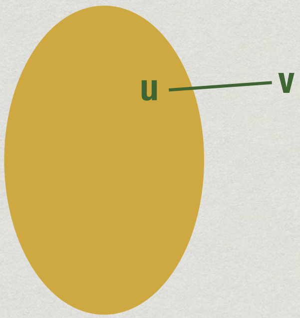


Minimum separator lemma

- Let V be partitioned into two non-empty sets U and W = V - U
- Let e = (u,w) be minimum cost edge with u in U and w in W
  - Assume all edges have different weights (relax this condition later)
- Then every minimum cost spanning tree must include e

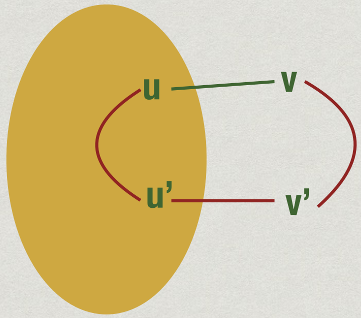
Minimum separator lemma

- Let T be a minimum cost spanning tree, e = (u,w) not in T
- u in U and w in W are connected by a path p in T
  - p starts in U and ends in W
  - Let f = (u’,w’) be the first edge on p such that u’ in U and w’ in W
  - Drop f and add e to get a smaller spanning tree



Minimum separator lemma

- Let T be a minimum cost spanning tree, e = (u,w) not in T
- u in U and w in W are connected by a path p in T
  - p starts in U and ends in W
  - Let f = (u',w') be the first edge on p such that u' in U and w' in W
  - Drop f and add e to get a smaller spanning tree

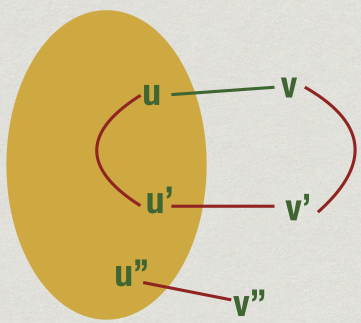


Correctness of Prim's algorithm

- Correctness follows directly from minimum separator lemma
- At each stage, TV and (V-TV) form a non-trivial partition of V
- The smallest edge connecting TV to (V-TV) must belong to every minimum cost spanning tree
  - This is the edge that the algorithm picks

Minimum separator lemma

- Proof of the lemma is slightly subtle
- Not enough to replace any edge from U to W by e = (u,v)
- Need to identify such an edge on the path from u to v



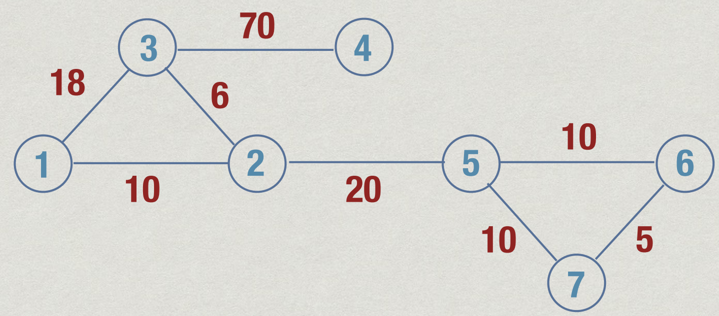
Further observations

- Need not start with smallest edge overall
  - For any vertex v, smallest edge attached to v must be in the minimum cost spanning tree
    - Consider the partition {v}, V-{v}
  - Can start with any such edge

Prim's algorithm revisited

- Start with TV = {s} for any vertex s
- For each vertex v outside TV, maintain
  - Distance_TV(v), smallest edge weight from v to TV
  - Neighbour_TV(v), nearest neighbour of v in TV
- At each stage, add to TV ("burn") vertex u with smallest Distance_TV(u)
  - Update Distance_TV(v), Neighbour_TV(v) for each neighbour of u
- Very similar to Dijkstra's algorithm!

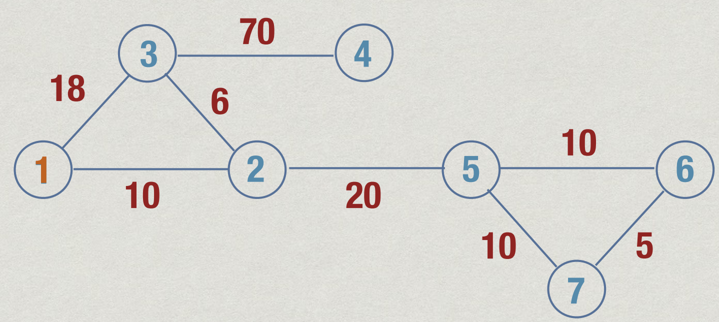
Prim's algorithm



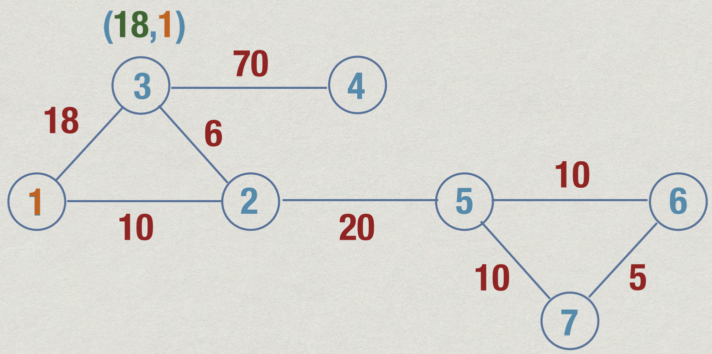
Prim's algorithm, refined

```
function Prim
  for i = 1 to n
    visited[i] = False; Nbr_TV[i] = -1; Dist_TV[i] = infinity

  TE = []   //List of spanning tree edges
  visited[1] = True
  for each edge (1,j)
    Nbr_TV[j] = 1; Dist_TV[j] = weight(1,j)

  for i = 2 to n
    Choose u such that Visited[u] == False
                       and Dist_TV[u] is minimum
    Visited[u] = True
    TE.append{(u,Nbr_TV[u])}
    for each edge (u,v) with Visited[v] == False
      if Dist_TV[v] > weight(u,v)
        Dist_TV[v] = weight(u,v); Nbr_TV[i] = u
```

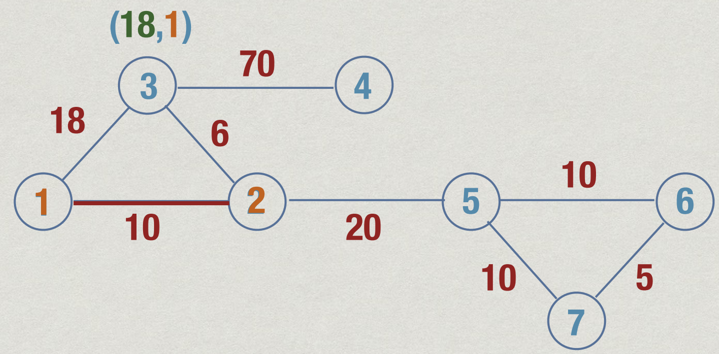
Prim's algorithm



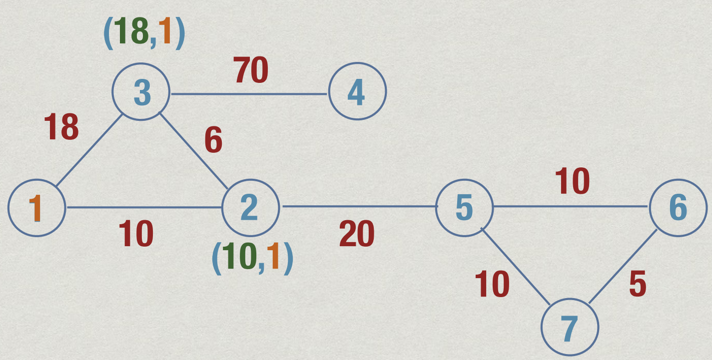
Prim's algorithm



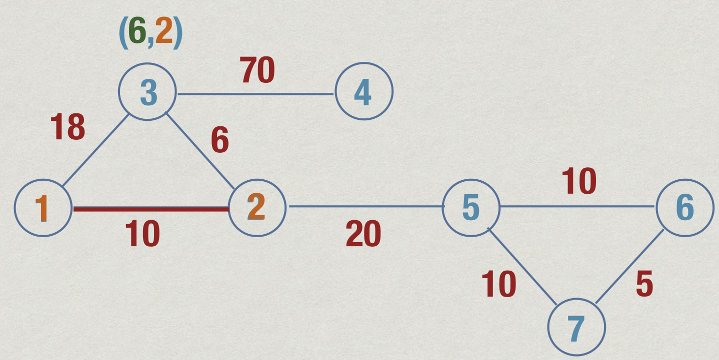
Prim's algorithm



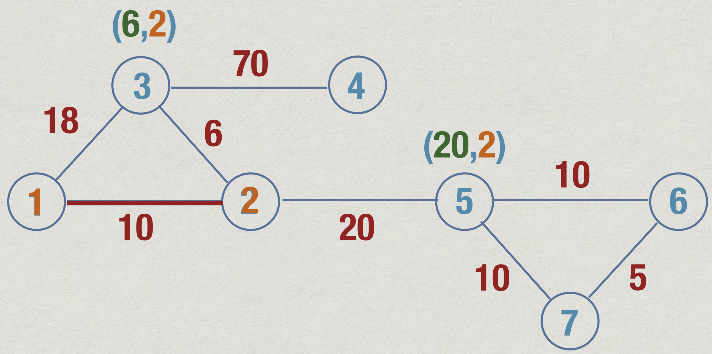
Prim's algorithm



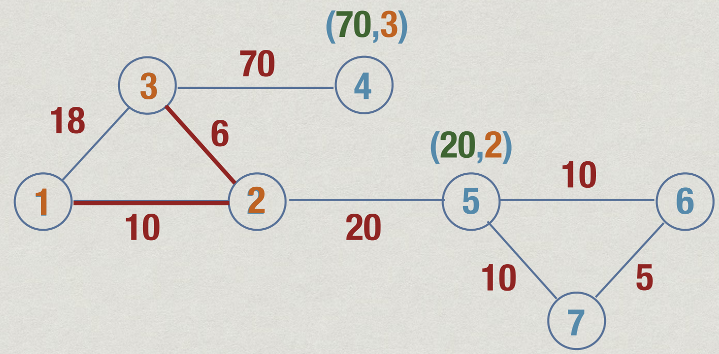
Prim's algorithm



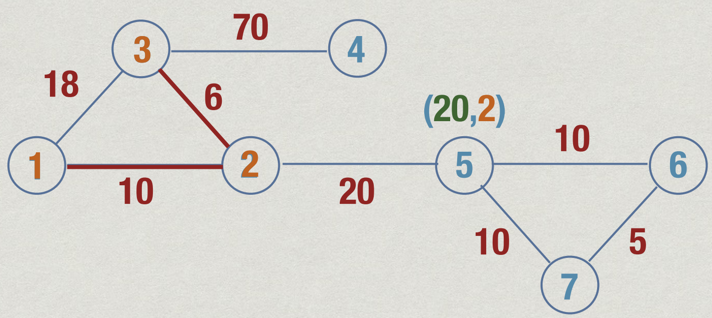
Prim's algorithm



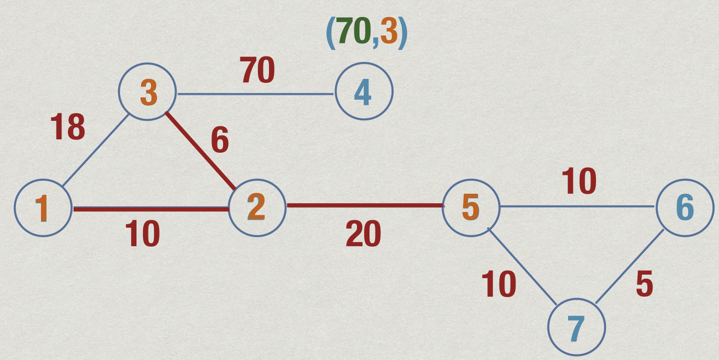
Prim's algorithm



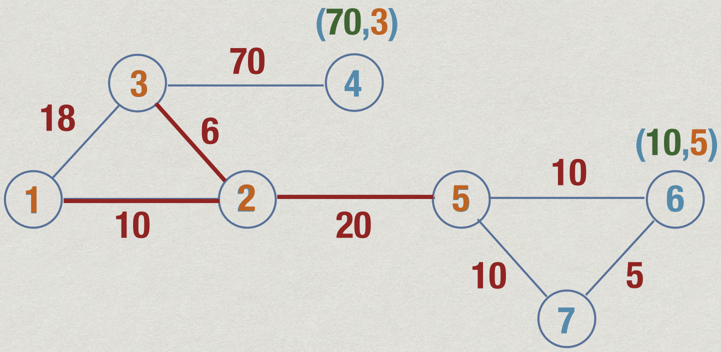
Prim's algorithm



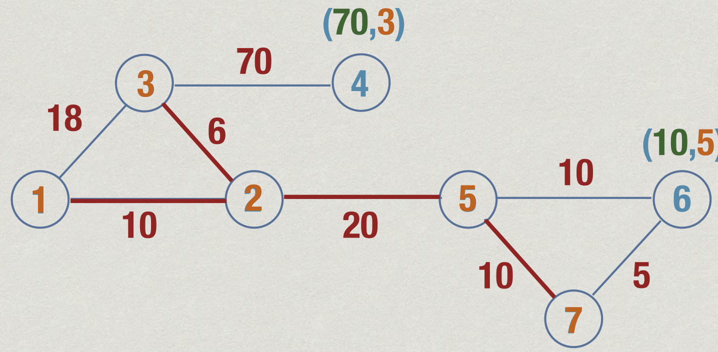
Prim's algorithm



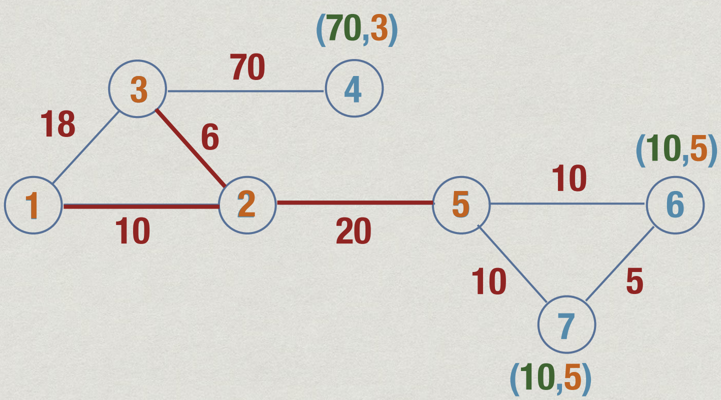
Prim’s algorithm



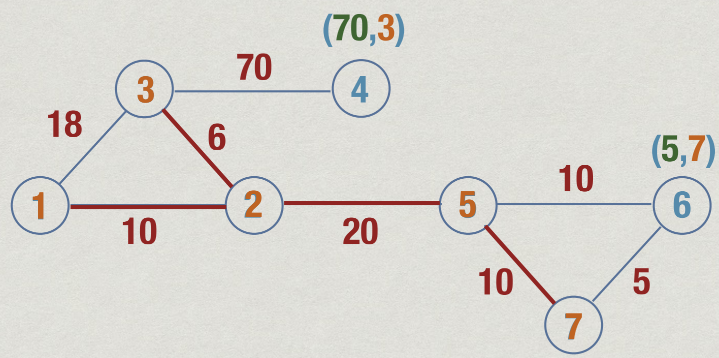
Prim’s algorithm



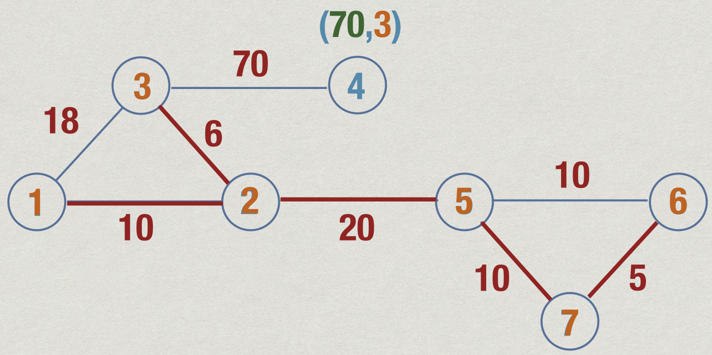
Prim’s algorithm



Prim’s algorithm



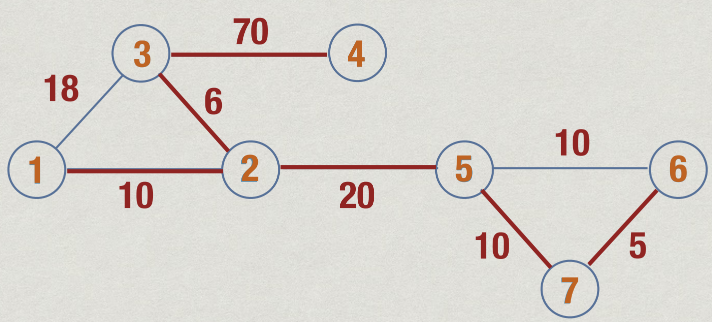
Prim's algorithm



Complexity

- Similar to Dijkstra's algorithm
- Outer loop runs n times
  - In each iteration, we add one vertex to the tree
  - O(n) scan to find nearest vertex to add
- Each time we add a vertex v, we have to scan all its neighbours to update distances
  - O(n) scan of adjacency matrix to find all neighbours
- Overall $O(n^2)$

Prim's algorithm



Complexity

- Moving from adjacency matrix to adjacency list
  - Across n iterations, O(m) to update neighbours
- Maintain distance information in a heap
  - Finding minimum and updating is O(log n)
- Overall O(n log n + m log n) = O((m+n) log n)

Minimum separator lemma

- We assumed edge weights are distinct
- Duplicate edge weights?
  - Fix an overall ordering {1,2,…,m} of edges
  - Edge e = ((u,v),i) is smaller than f = ((u',v'),j) if
    - weight(e) < weight(f)
    - weight(e) = weight(f) and i < j

Spanning tree

- Weighted undirected graph, G = (V,E,w)
  - Assume G is connected
- Identify a spanning tree with minimum weight
  - Tree connecting all vertices in V
- **Strategy 2:**
  - Order edges in ascending order by weight
  - Keep adding edges to combine components

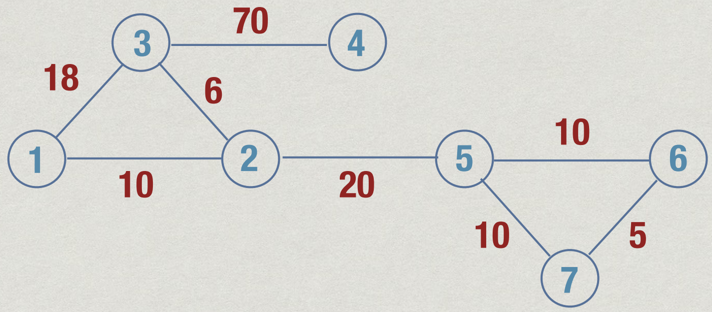
Multiple spanning trees

- If edge weights repeat, the minimum cost spanning tree is not unique
  - "Choose u such that Dist_TV(u) is minimum"
- Different choices generate different trees
  - Different ways of ordering edges {1,2,…,m}
- In general, number of possible minimum cost spanning trees is exponential
  - Greedy algorithm efficiently picks out one of them

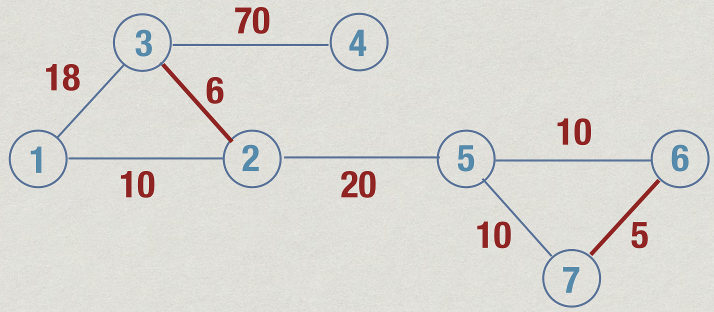
Kruskal's algorithm

```
algorithm Kruskal_V1

Let E = [e1,e2,…,em] be edges sorted by weight

TE = [] // List of edges added so far
i = 1   // Index of edge to try next

while TE.length() < n-1 //n-1 edges form a tree
  if adding E[i] to TE does not form a cycle
    TE.append(E[i])
    i = i+1
```

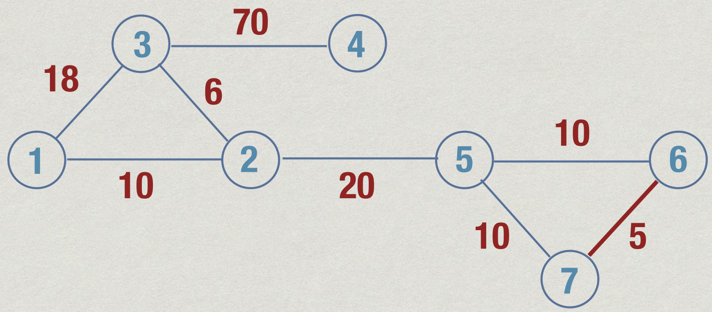
Kruskal's algorithm



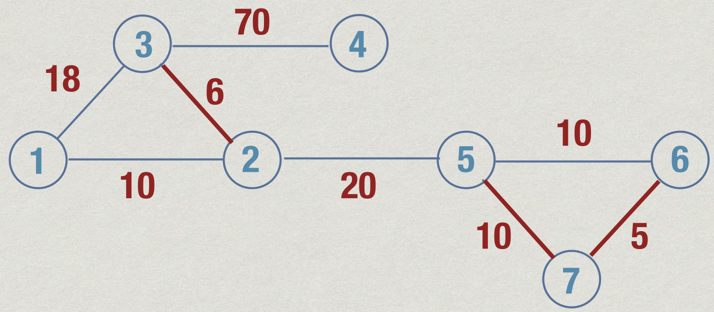
Kruskal's algorithm



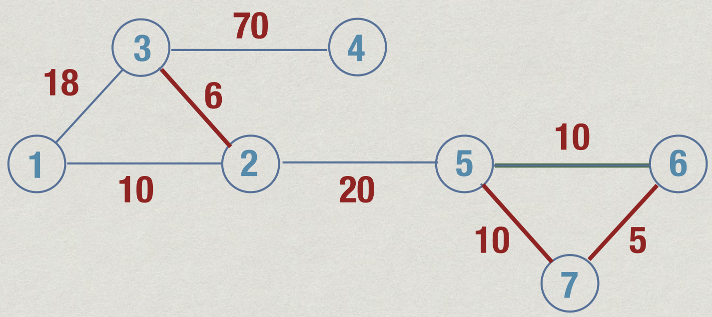
Kruskal's algorithm



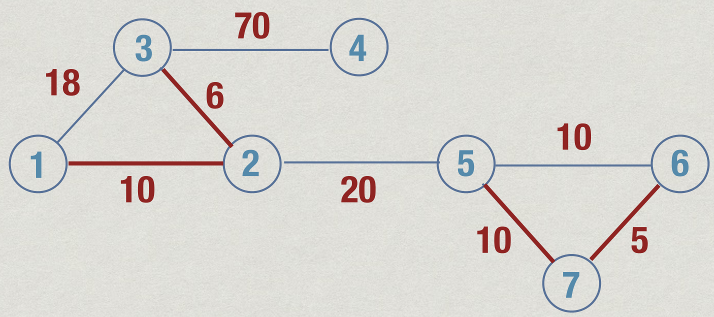
Kruskal's algorithm



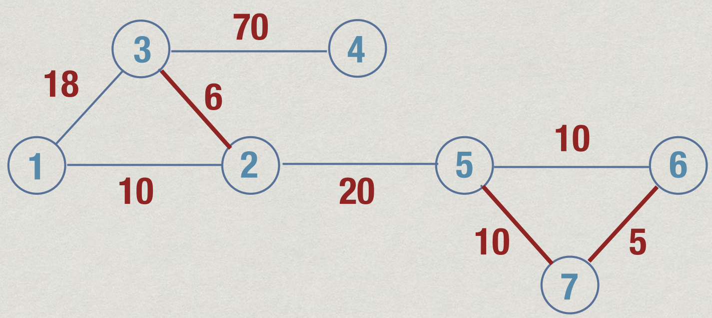
Kruskal's algorithm



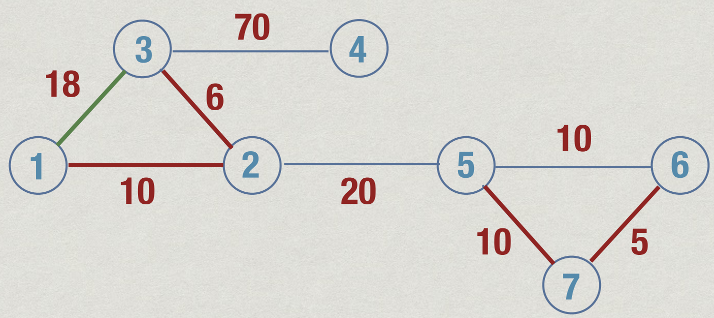
Kruskal's algorithm



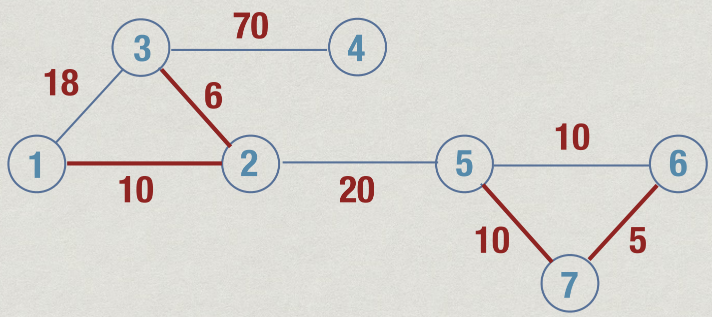
Kruskal's algorithm



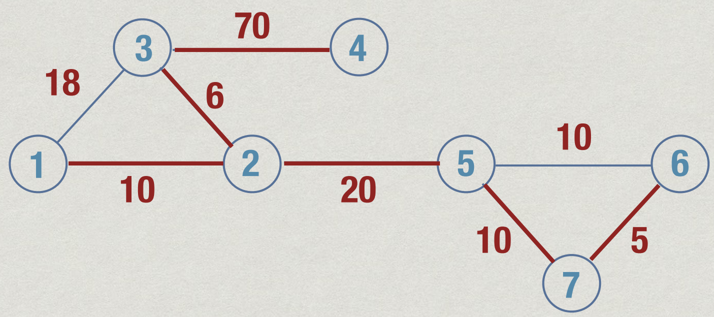
Kruskal's algorithm



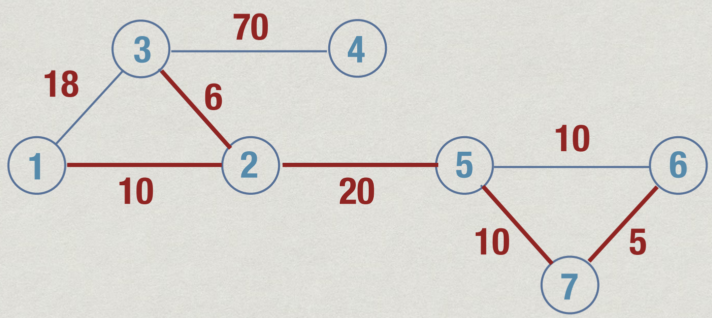
Kruskal's algorithm



Kruskal's algorithm



Kruskal's algorithm



Correctness

- Kruskal's algorithm is also a greedy algorithm
- We fix in advance that edges will be added in ascending order of weight
- Why does this achieve a global optimum?

Minimum separator lemma

- Let V be partitioned into two non-empty sets U and W = V - U
- Let e = (u,w) be minimum cost edge with u in U and w in W
  - Assume all edges have different weights
- Then every minimum cost spanning tree must include e

Correctness of Kruskal's algorithm …

- Suppose e_j = (u,v) with u and v in disjoint components
  - Let U = Component(u), W = V - Component(u)
  - No smaller weight edge in $[e_1, e_2, \ldots, e_{j-1}]$ connects U and W
  - By minimum separator lemma, e_j must be in the minimum cost spanning tree

Correctness of Kruskal's algorithm …

- Unlike Prim's algorithm, at intermediate stages TE is not a tree
- Edges in TE partition vertices into connected components
  - Initially, each vertex is a separate component
  - Adding e = (u,v) merges components of u and v
    - If u and v are already in same component, e forms a cycle, hence discarded

Kruskal's algorithm revisited

- To check if e = (u,v) forms a cycle, keep track of components
- Initially, Component[i] = i for each vertex i
- e = (u,v) can be added if Component[u] is different from Component[v]
  - Merge the two components

Kruskal's algorithm, refined

```
algorithm Kruskal

Let E = [e1,e2,…,em] be edges sorted by weight

for j in 1 to n      //Initially, each vertex is isolated
  Component[j] = j  //Component names are 1..n

TE = []              //List of edges added so far
i = 1                //Index of edge to try next

while TE.length() < n-1  //n-1 edges form a tree
  Let E[i] = (u,v)
  if Component[u] != Component[v] //E[i] does not form cycle
    TE.append(E[i])
    for j in 1 to n //Merge Component[v] into Component[u]
      if Component[j] == Component[v]
        Component[j] = Component[u]
```

Bottleneck

- Naive strategy for labelling and merging components is inefficient
- Components form a partition of the vertex set V
- Union-find data structure implements the following operations efficiently
  - find(v)—find the component containing v
  - union(u,v) —merge the components of u and v
- This will bring down the complexity to O(m log n)

Complexity

- Initially, sort edges, O(m log m)
  - m is at most n^2, so this is also O(m log n)
- Outer loop runs upto m times
  - In each iteration, we examine one edge
  - If we add the edge, we have to merge components
    - O(n) scan to update components
    - This is done once for each tree edge—O(n) times
- Overall $O(n^2)$