lecture 7

single source shortest paths

Dijkstra's algorithm

[graph can be directed or undirected]

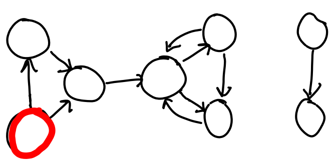
Resources for this lecture

- Sedgewick Algorithms 2
  - https://class.coursera.org/algs4partII-002/lecture/20
- Roughgarden's Algorithms 1
  https://class.coursera.org/algo-004/lecture/57

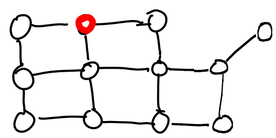
Warmup Problem

Given a vertex s ("source / start")
in a graph (directed or not),
find the "shortest" path (fewest edges)
to all reachable vertices.

directed graph

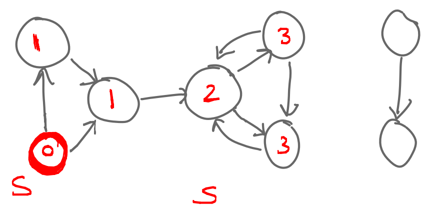


undirected graph

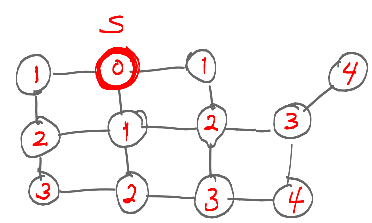


Solution: use breadth first search

directed graph



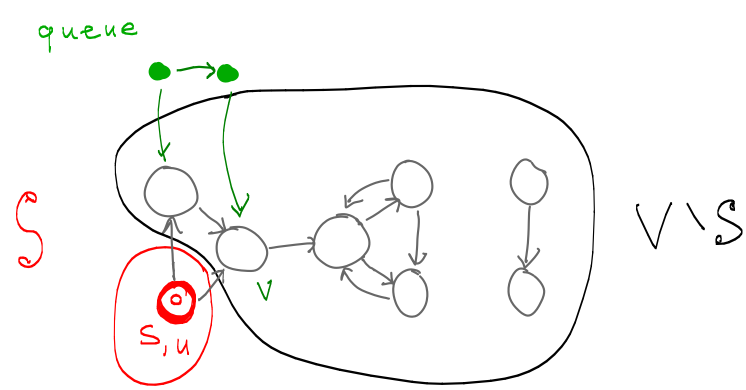
undirected graph

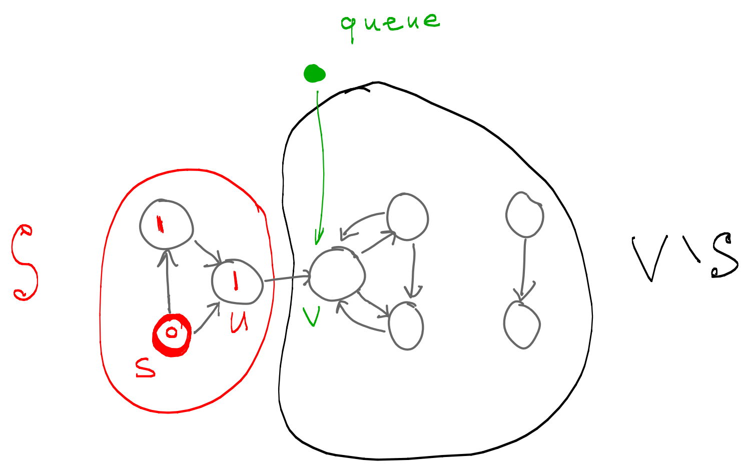


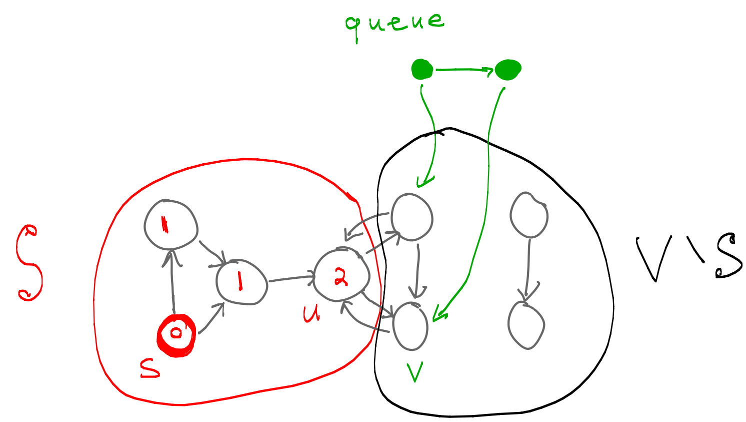
Finds vertices reachable by paths of length ℓ
ℓ = 0, 1, 2, ...

BFS (review COMP 250)

```
S = { }                      //  set of vertices for which we know shortest path
for each u in V,
    d[ u ] = infinity        //  length of shortest path
    parent[ u ] = null       //  previous node in shortest path
    visited[ u ] = false
initialize empty queue

queue.add( startingVertex )
dist[ startingVertex ]      = 0
visited[ starting Vertex ] = true

while queue is not empty
    u = queue.remove()
    add u to S
    for each edge (u,v)              //   for each v in u's adjList
        if  !(visited[ v ])
            queue.add(v)
            visited[v] = true
            parent[ v ] = u          // shortest known path to v is via u
            dist[ v ] = dist[ u ] + 1
```

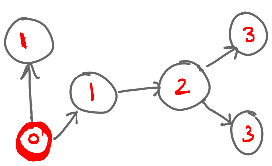




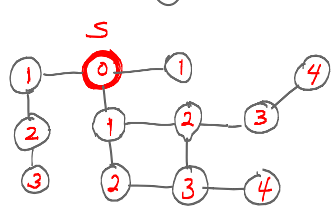


BFS defines a rooted tree which is typically different than the DFS tree. The edges of the tree are defined by the 'parent'.

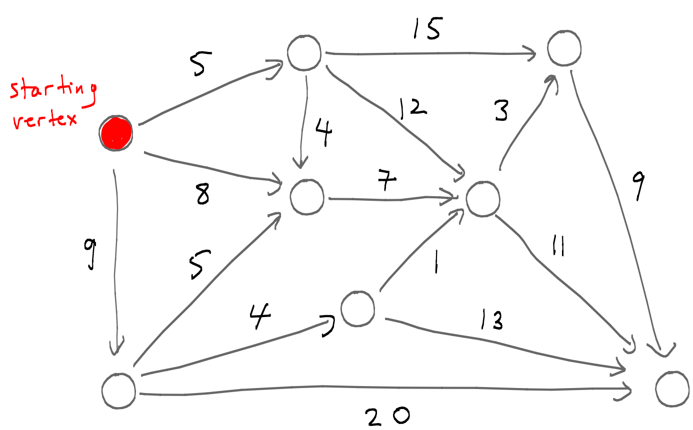
directed graph



undirected graph



Shortest paths in weighted graphs?

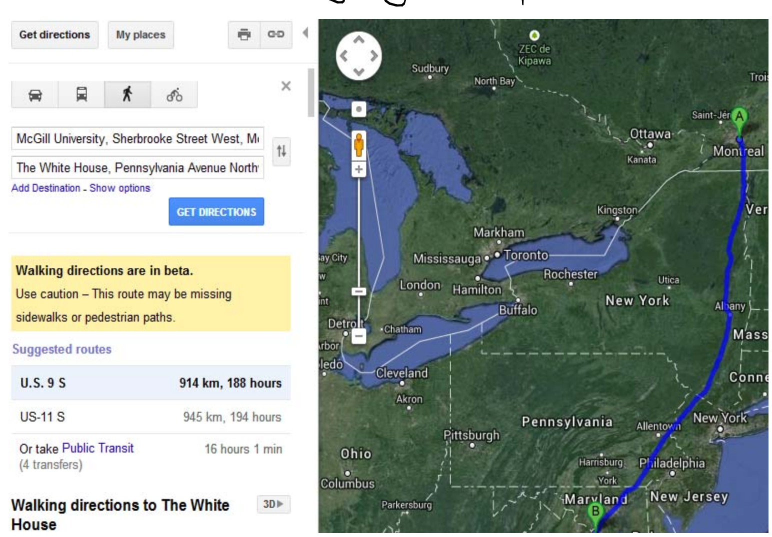


Given a weighted graph and a starting vertex s, what is the shortest cost path (sum of weights) from s to each vertex v.

Assume:

- edge weights (costs) are ≥ 0
- all vertices are reachable from s

Applications: google maps, TomTom



Solution (Dijkstra, 1959)

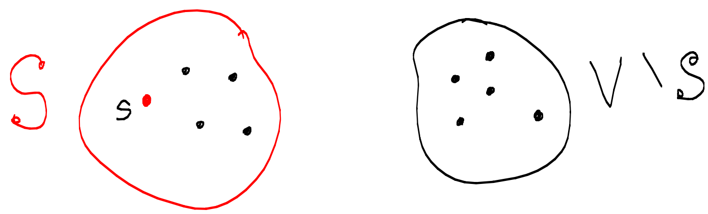
Similar to BFS.

First I'll explain the main idea.

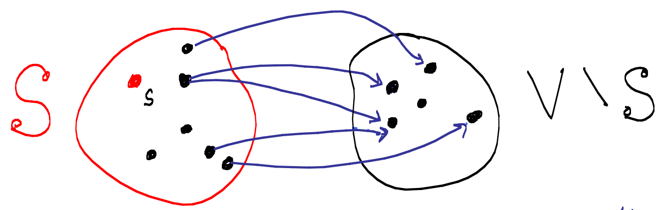
Then I'll prove it is correct.

Then I'll show how to speed it up using an indexed priority queue.

Main idea: at any stage of the algorithm, the vertices are partitioned into two sets (similar to BFS).



Vertices for which we know the shortest cost path from s. (includes vertex s)



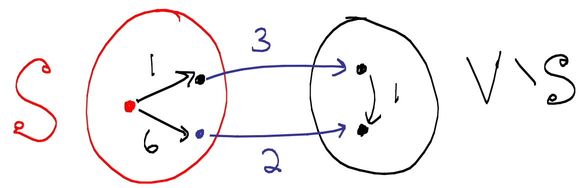
Consider the "crossing edges" from S to V\S

$\{ (u,v) : u \in S, \ v \in V \setminus S \}$.

Dijkstra: Choose the crossing edge (u,v) that gives the shortest cost path of the form $(\underbrace{s, \dots u}_{\text{in } S}, v)$.

The chosen edge (u,v) might not be the shortest crossing edge from S to V \ S.

e.g.



Dijkstra would choose the 3 edge next instead of the 2 edge.

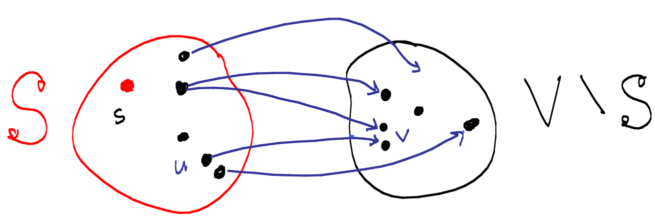
Notation / Definitions

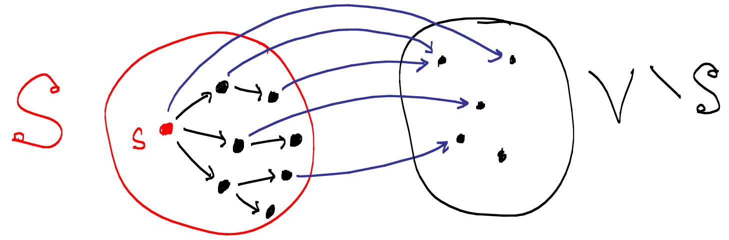
- (u,v) is an edge with weight/cost $cost(u,v) \geq 0$
- "path length" = sum of costs of edges on a path
- dist[v] is the smallest "path length" (length of the shortest path) from s to v.

Dijkstra's Algorithm: Given a graph G = (V,E) and vertex s in V.

```
for all u in V,  initialize dist[ u ] = infinity
S =  {s}
dist[ s ] = 0

while | S | < | V | {

    choose "best crossing edge"  (u,v)  as defined earlier
    add v to S
    dist[ v ] =  dist[ u ] + cost(u, v)
}
```

how ?

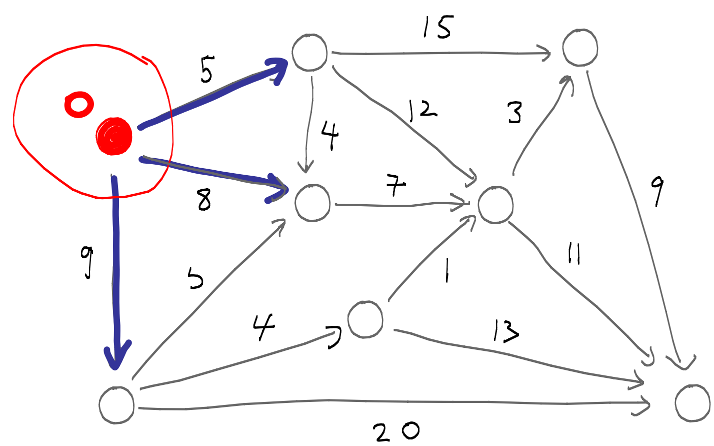


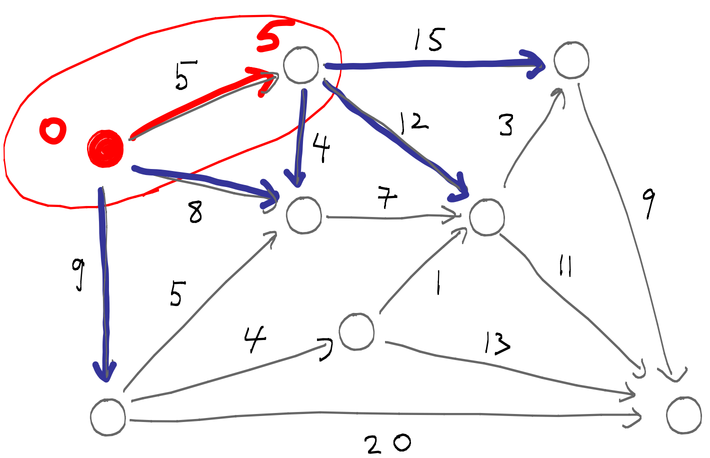


Like BFS, Dijkstra grows a rooted tree.

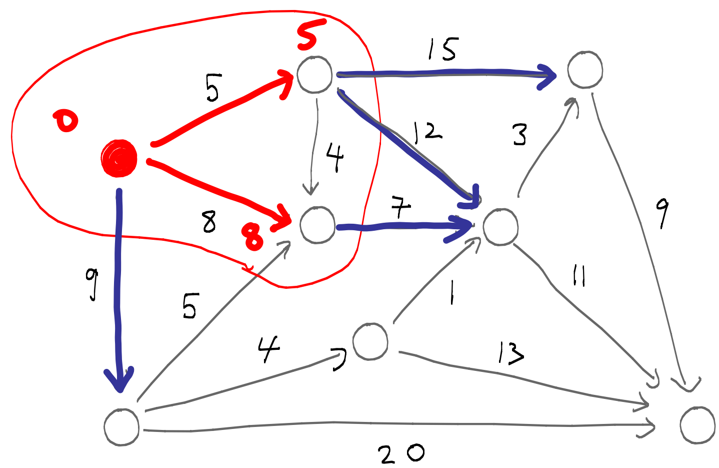
Unlike BFS which increments path lengths by 1, Dijkstra increments path lengths by an edge cost (possibly different than 1).

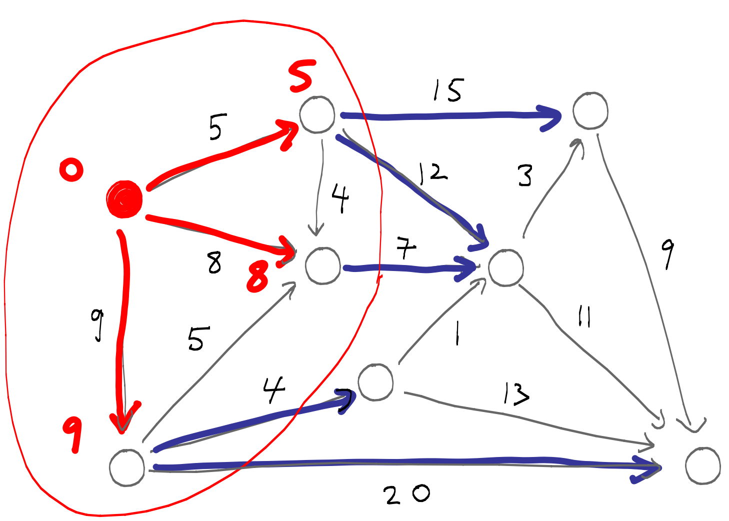
Example

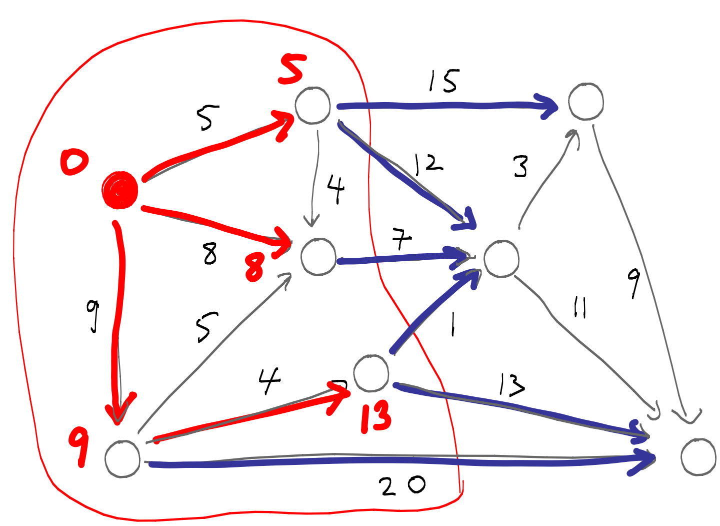


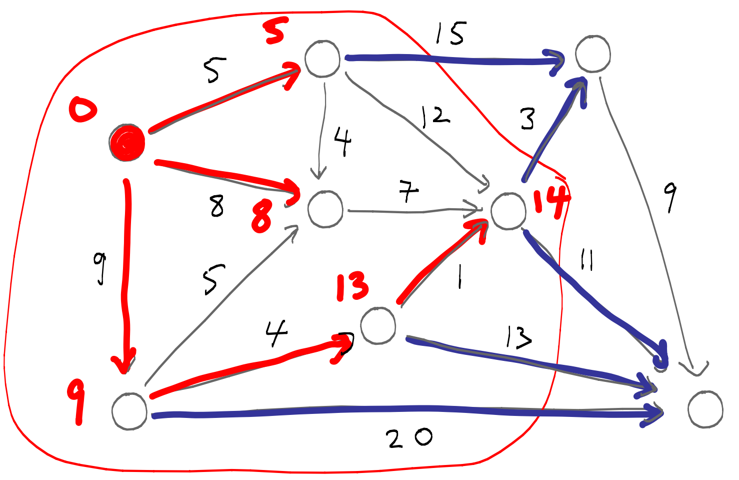


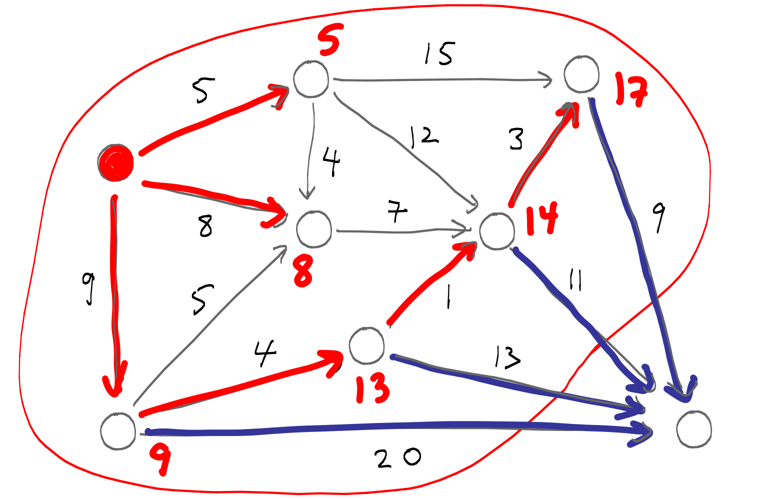
Example

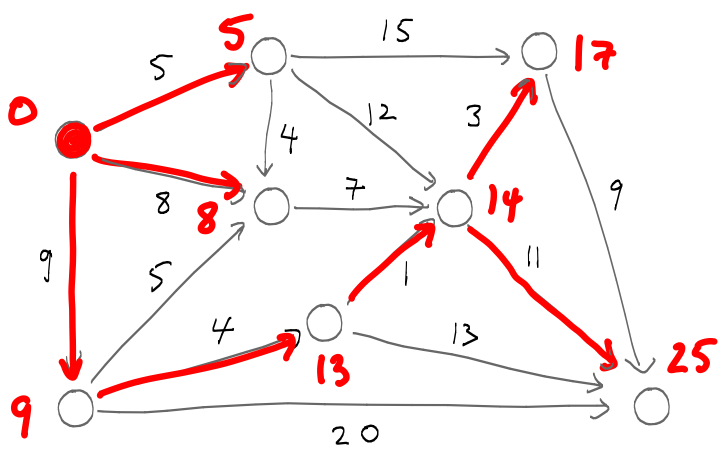












Dijkstra's algorithm seems simple enough but there are a few subtleties:

1) proving it is correct

2) using a priority queue to efficiently find the best crossing edge

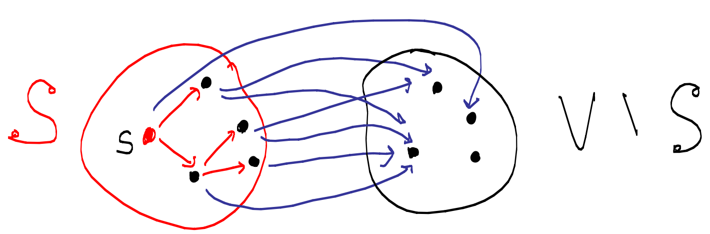
Claim: during the execution of Dijkstra's algorithm, for each u in S, the algorithm has found the shortest path from s to u.

Proof (by induction on the size of S):

Base case: if |S| = 1, then dist(s) = 0. (trivial)

Induction hypothesis: the claim is true when |S| = k.

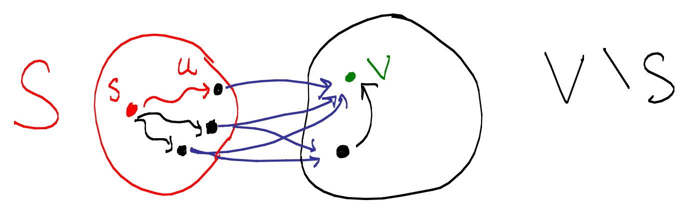
Induction step: if claim is true for |S| = k, then claim is true for |S| = k+1.



Let v be the k+1-st node added to S by Dijkstra.

Let (u,v) be the crossing edge chosen by Dijkstra, where u in S.

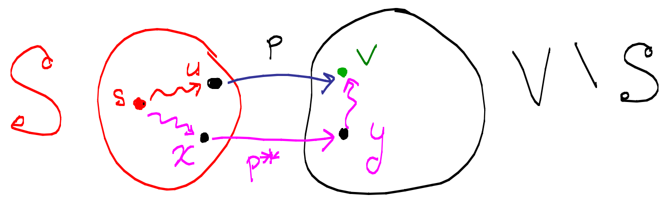
Let **P** be the path from s to v found by Dijkstra, namely (s, .. u, v).



By induction hypothesis, (s, .. ,u) the shortest path from s to u.

Could there be a shorter path from s to v than the one found by Dijkstra ?

Take **any other path P*** from s to v. Let **x** be the last vertex in S along this path P*, and let **y** be the vertex that follows **x**. By definition, **y** is in V \ S.



cost(**P***) >= dist[**x**] + cost(**x**,**y**) since edge costs are non-negative and y might be different than v.

>= dist[u] + cost(u, v) since Dijkstra chose v rather than y at step k+1

= cost(P)

[Proof still works if x is u and/or y is v.]

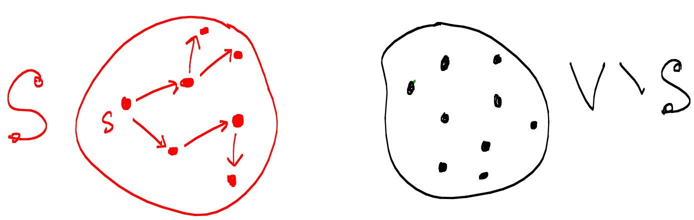
Dijkstra seems simple enough but there are a few subtleties

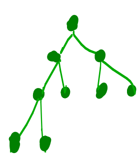
1) proving it is correct

2) using a priority queue to implement it efficiently

Q: How to decide which is the next vertex to be added to S ?

A: Use a priority queue (pq), whose priorities are distances of the shortest <u>known</u> path from s to v, where v in V \ S.

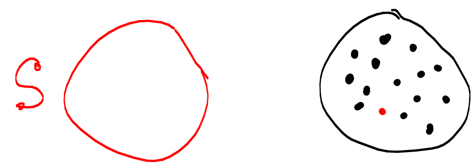




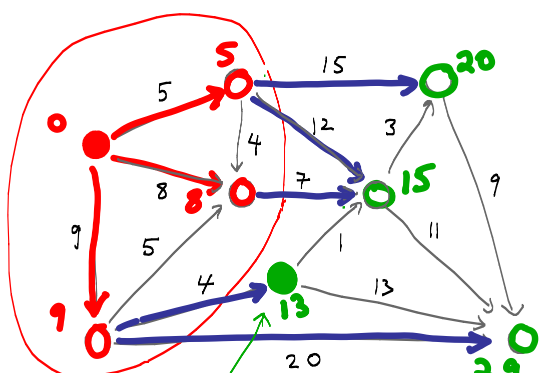
priority queue

<u>Dijkstra's algorithm (find shortest path from a given vertex s)</u>

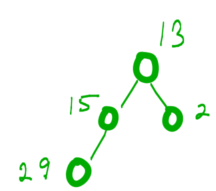
```
// initialization

S = { }
initialize empty priority queue pq
for each u in V
    parent[ u ] = null          //  u.parent = null
    pq.add( u,  infinity)       //  add all vertices to pq

pq.changePriority( s, 0)

//  see next slide for the main loop
```

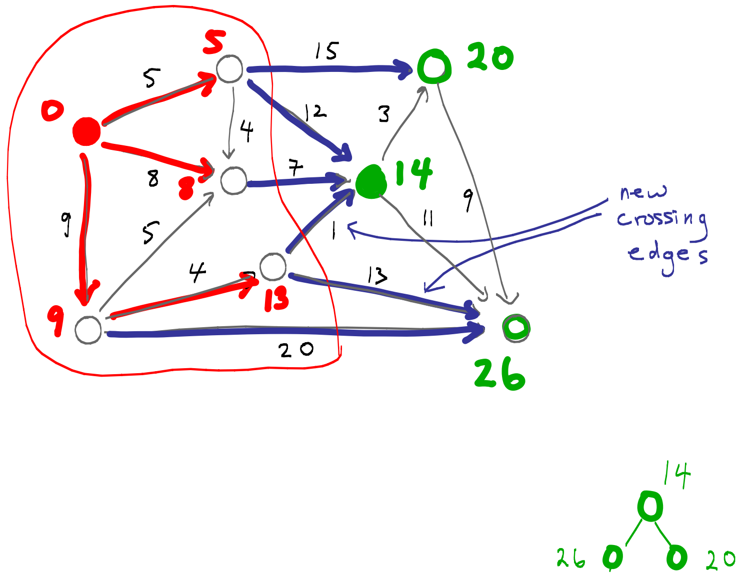


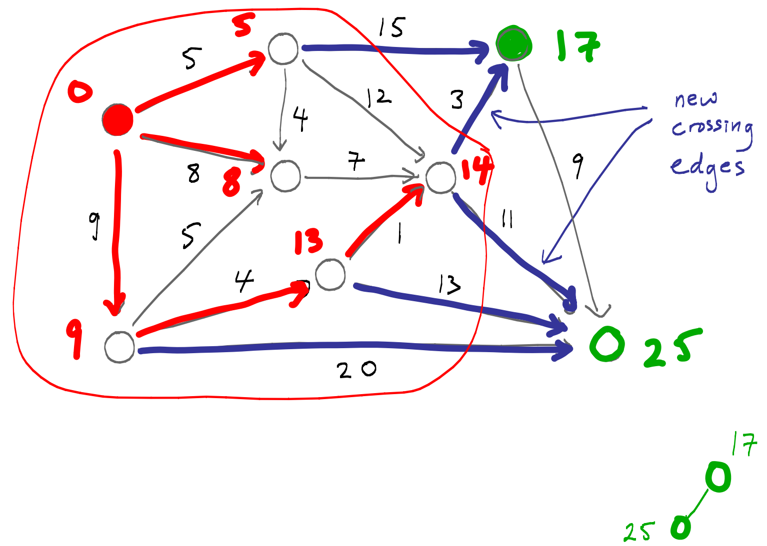

```
while pq is not empty {

   u        =  pq.getMinVertex()       //  slightly different from
   dist[u] =  pq.removeMinDistance()   //  IndexedHeap
   S.add( u )                          //  in assignments

   for each  v in u.adjList
       if v in V \ S {                 //  (u,v) is a crossing edge

           tmpDistToV =  dist[ u ] + cost(u,v)
           if  tmpDistToV  <  pq.getPriority( v )
               pq.setPriority(v, tmpDistV )
               parent[ v ] = u
       }
   }
}
```

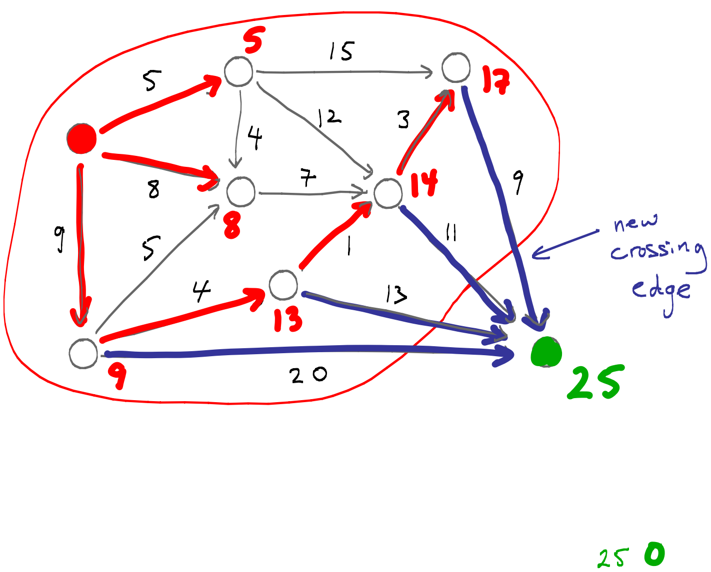


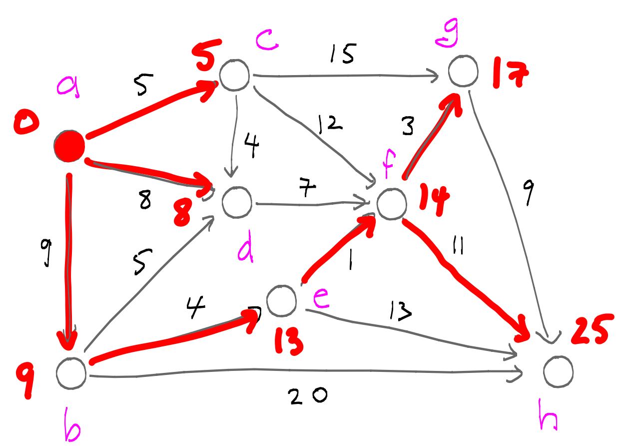
this vertex gets removed from priority queue











Test case for A2.

<u>How much space and time does Dijkstra's algorithm take?</u>

Recall we assumed there is a path from s to all nodes.

<u>Space:</u> Each vertex u in V enters the priority queue once. So the size of the priority queue is at most |V|.

<u>Time:</u> There is at most one pq.changePriority() per edge in the graph. If the priority queue is implemented as a heap, then each update takes O(log |V|) time.

So Dijkstra's algorithm takes O(|E| log |V|) time.

Assignment 2

Implement two versions of Dijkstra

1) $O(E \log |V|)$ version that I just described.

2) $O(E \log |E|)$ version

store edges rather than vertices in the priority queue