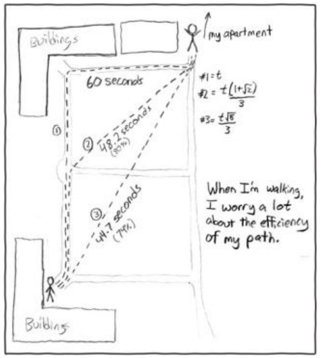
CSE 421: Algorithms

Winter 2014

Lecture 9: MSTs and shortest paths

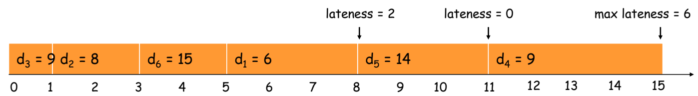
Reading: Sections 4.1-4.5



review: scheduling to minimize lateness

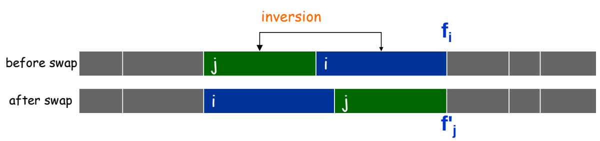
Example:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



minimizing lateness: inversions

- **Definition.** An **inversion** in schedule **S** is a pair of jobs **i** and **j** such that $d_i < d_j$ but **j** scheduled before **i**.



- **Claim.** Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

optimal schedules and inversions

- **Claim:** There is an optimal schedule with no idle time and no inversions
- Proof:
  - By previous argument there is an optimal schedule **O** with no idle time
  - If **O** has an inversion then it has a **consecutive** pair of requests in its schedule that are inverted and can be swapped without increasing lateness

optimal schedules and inversions

Eventually these swaps will produce an optimal schedule with no inversions

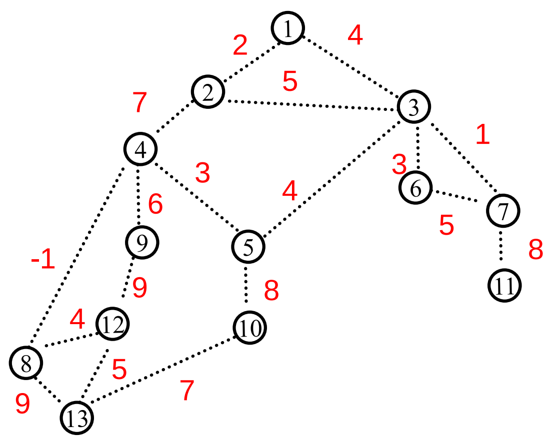
- Each swap decreases the number of inversions by 1
- There are a bounded number of (at most $n(n-1)/2$) inversions (we only care that this is finite.)

QED

minimum spanning trees (or forests)

- Given an undirected graph $G=(V,E)$ with each edge e having a weight $w(e)$
- Find a subgraph T of G of minimum total weight s.t. every pair of vertices connected in G are also connected in T
  - if G is connected then T is a tree otherwise it is a forest

weighted undirected graph



greedy algorithm

Prim's Algorithm:

- start at a vertex s
- add the cheapest edge adjacent to s
- repeatedly add the cheapest edge that joins the vertices explored so far to the rest of the graph

prim's algorithm

Prim(**G**,**w**,**s**)
S ← {**s**}

while **S**≠**V** do
of all edges **e**=(**u**,**v**) s.t. **v**∉**S** and **u**∈**S** select* one with the minimum value of **w**(**e**)
S←**S**∪ {**v**}
pred[**v**]←**u**

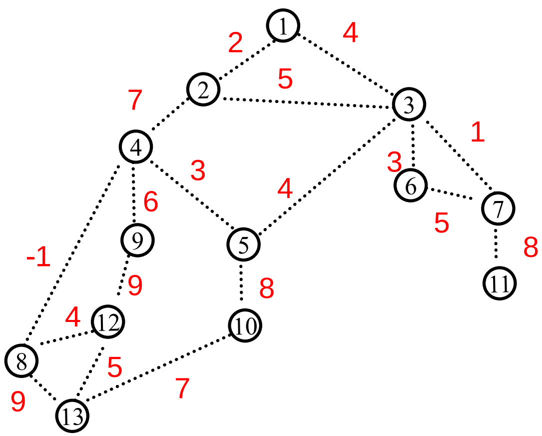
*For each v∉S maintain small[v]=minimum value of w(e) over all vertices u∈S s.t. e=(u,v) is in of G

second greedy algorithm

Kruskal's Algorithm

– Start with the vertices and no edges
– Repeatedly add the cheapest edge that joins two different components, i.e. that doesn't create a cycle

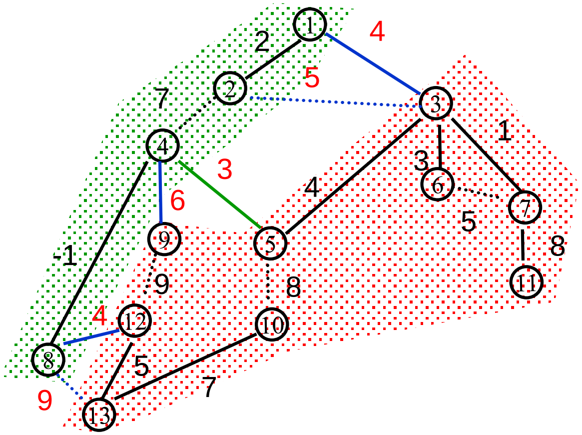
weighted undirected graph



why greed is good

- **Definition:** Given a graph G=(V,E), a **cut** of G is a partition of V into two non-empty pieces, S and V-S
- **Lemma:** For every cut (S,V-S) of G, there is a minimum spanning tree (or forest) containing any **cheapest edge crossing the cut**, i.e. connecting some node in S with some node in V-S.
  – call such an edge **safe**

cuts and spanning trees



the greedy algorithms always choose safe edges

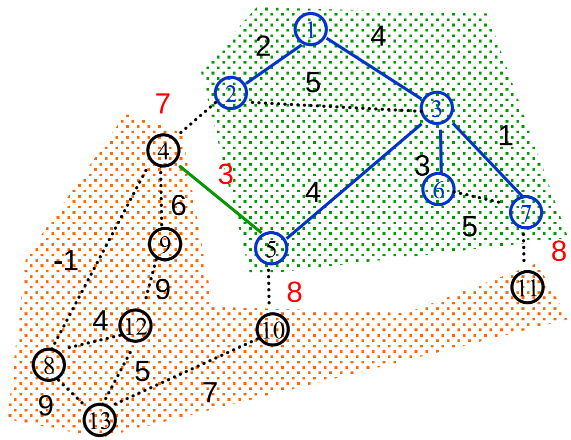
Prim's Algorithm

the greedy algorithms always choose safe edges

Prim's Algorithm

- Always chooses cheapest edge from current tree to rest of the graph
- This is cheapest edge across a cut which has the vertices of that tree on one side.

prim's algorithm



the greedy algorithms always choose safe edges

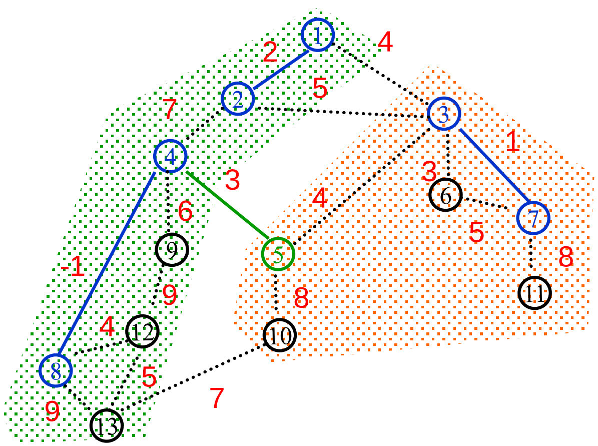
Kruskal's Algorithm

the greedy algorithms always choose safe edges

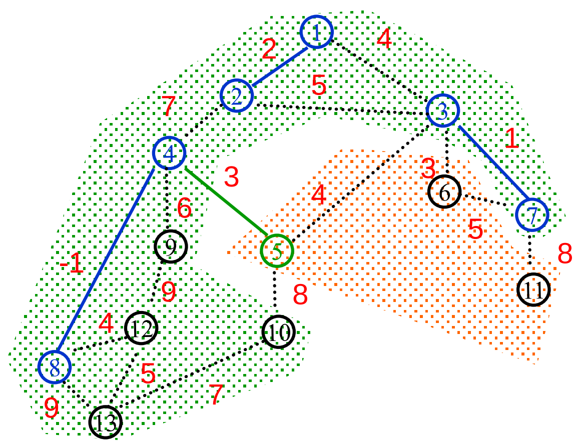
Kruskal's Algorithm

– Always chooses cheapest edge connecting two pieces of the graph that aren't yet connected
– This is the cheapest edge across any cut which has those two pieces on different sides and doesn't split any current pieces.

kruskal's algorithm

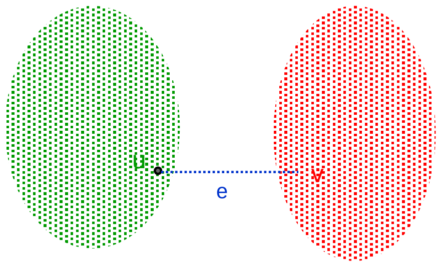


kruskal's algorithm



proof of lemma: exchange argument

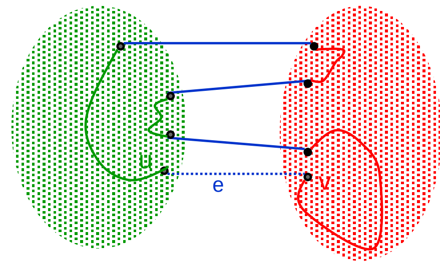
Suppose you have an MST not using cheapest edge e



Endpoints of e, u and v must be connected in T

proof of lemma

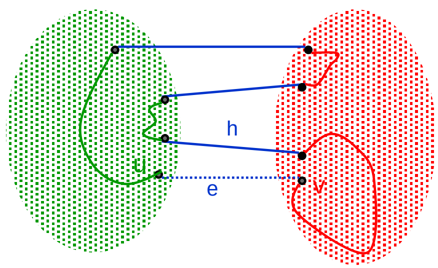
Suppose you have an MST T not using cheapest edge e



Endpoints of e, u and v must be connected in T

proof of lemma

Suppose you have an MST T not using cheapest edge e

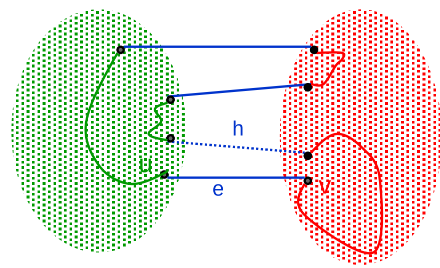


Endpoints of e, u and v must be connected in T

$$w(e) \le w(h)$$

proof of lemma

Suppose you have an MST T not using cheapest edge e



Endpoints of e, u and v must be connected in T

$$w(e) \le w(h)$$

implementation and analysis (kruskal)

- First sort the edges by weight $O(m \log m)$
- Go through edges from smallest to largest
  - if endpoints of edge e are currently in different components

    then add to the graph

    else skip
- Union-find data structure handles last part
- Total cost of last part: $O(m\ \alpha(n))$ where $\alpha(n) \ll \log m$
- Overall $O(m \log n)$

union-find disjoint sets data structure

- Maintaining components
  - start with n different components

    one per vertex
  - find components of the two endpoints of e

    $2m$ finds
  - union two components when edge connecting them is added

    $n-1$ unions

prim's algorithm with priority queues

- For each vertex u not in tree maintain current cheapest edge from tree to u
  - Store u in priority queue with key = weight of this edge
- Operations:
  - $n-1$ insertions (each vertex added once)
  - $n-1$ delete-mins (each vertex deleted once)

    pick the vertex of smallest key, remove it from the p.q. and add its edge to the graph
  - $< m$ decrease-keys (each edge updates one vertex)

prim's algorithm with priority queues

- Priority queue implementations
  - Array

    insert $O(1)$, delete-min $O(n)$, decrease-key $O(1)$

    total $O(n+n^2+m)=O(n^2)$
  - Heap

    insert, delete-min, decrease-key all $O(\log n)$

    total $O(m \log n)$
  - d-Heap ($d=m/n$)

    insert, decrease-key $O(\log_{m/n} n)$

    delete-min $O((m/n) \log_{m/n} n)$

    total $O(m \log_{m/n} n)$

an application

Minimum cost network design:

- Build a network to connect all locations $\{v_1,\ldots,v_n\}$
- Cost of connecting v_i to v_j is $w(v_i,v_j)>0$
- Choose a collection of links to create that will be as cheap as possible
- Any minimum cost solution is an MST
  - If there is a solution containing a cycle then we can remove any edge and get a cheaper solution

application #2

Maximum Spacing Clustering

- Given
  - a collection U of n objects $\{p_1,\ldots,p_n\}$
  - Distance measure $d(p_i,p_j)$ satisfying
    - $d(p_i,p_i)=0$
    - $d(p_i,p_j)>0$ for $i\neq j$
    - $d(p_i,p_j)=d(p_j,p_i)$
  - Positive integer $k\leq n$
- Find a k-clustering, i.e. partition of U into k clusters $C_1,\ldots,C_k$, such that the spacing between the clusters is as large possible where
  - spacing = $\min\{d(p_i,p_j)$: p_i and p_j in different clusters$\}$

greedy algorithm

- Start with n clusters each consisting of a single point
- Repeatedly find the closest pair of points in different clusters under distance d and merge their clusters until only k clusters remain
- Gets the same components as Kruskal's Algorithm does!
  - The sequence of closest pairs is exactly the MST
- Alternatively we could run Kruskal's algorithm once and for any k we could get the maximum spacing k-clustering by deleting the k-1 most expensive edges

proof

- Removing the k-1 most expensive edges from an MST yields k components $C_1,\ldots,C_k$ and the spacing for them is precisely the cost d^* of the k-1[st] most expensive edge in the tree
- Consider any other k-clustering $C'_1,\ldots,C'_k$
  - Since they are different and cover the same set of points there is some pair of points p_i,p_j such that p_i,p_j are in some cluster C_r but p_i, p_j are in different clusters C'_s and C'_t
    - Since $p_i,p_j \in C_r$, p_i and p_j have a path between them all of whose edges have distance at most d^*
    - This path must cross between clusters in the C' clustering so the spacing in C' is at most d^*

single-source shortest paths

- Given an (un)directed graph G=(V,E) with each edge e having a non-negative weight w(e) and a vertex v
- Find length of shortest paths from v to each vertex in G

a greedy algorithm

Dijkstra's Algorithm:

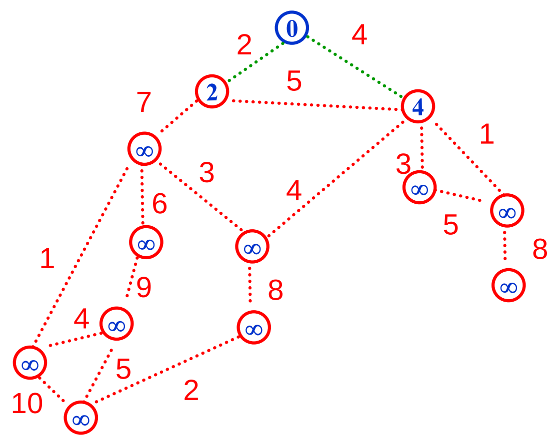
- Maintain a set S of vertices whose shortest paths are known
  - initially S={s}
- Maintaining current best lengths of paths that only go through S to each of the vertices in G
  - path-lengths to elements of S will be right, to V-S they might not be right
- Repeatedly add vertex v to S that has the shortest tentative distance of any vertex in V-S
  - update path lengths based on new paths through v

Dijkstra's algorithm

Dijkstra(**G**,**w**,**s**)
    S←{**s**}
    d[**s**]←**0**
    while **S**≠**V** do
        of all edges **e**=(**u**,**v**) s.t. **v**∉**S** and **u**∈**S** select* one with the minimum value of **d**[**u**]+**w**(**e**)
            S←S∪ {v}
            d[v]←d[u]+w(e)
            pred[**v**]←**u**

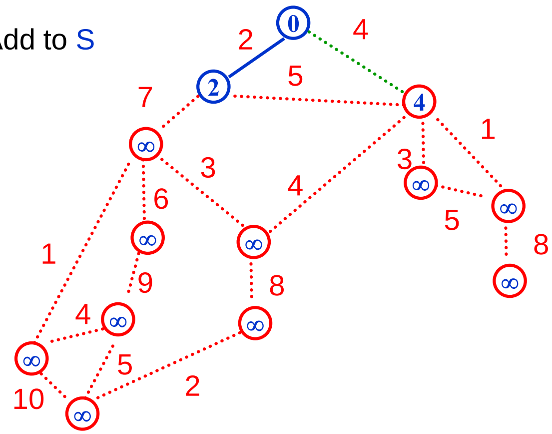
*For each v∉S maintain d'[v]=minimum value of d[u]+w(e) over all vertices u∈S s.t. e=(u,v) is in of G

Dijkstra's Algorithm



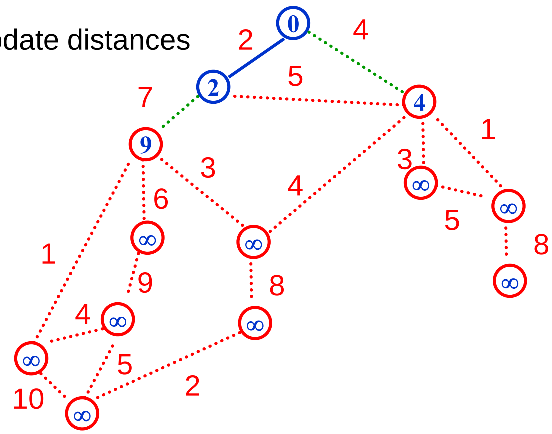
Dijkstra’s Algorithm

Add to S



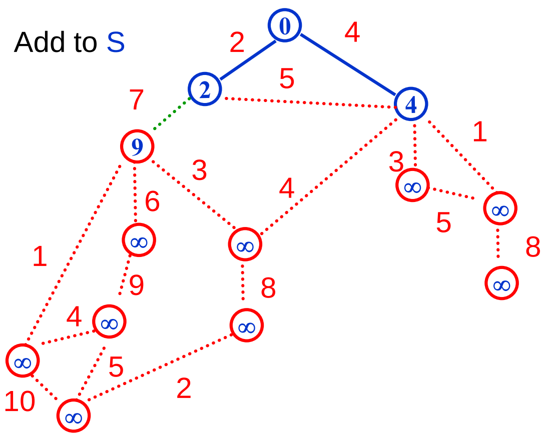
Dijkstra’s Algorithm

Update distances



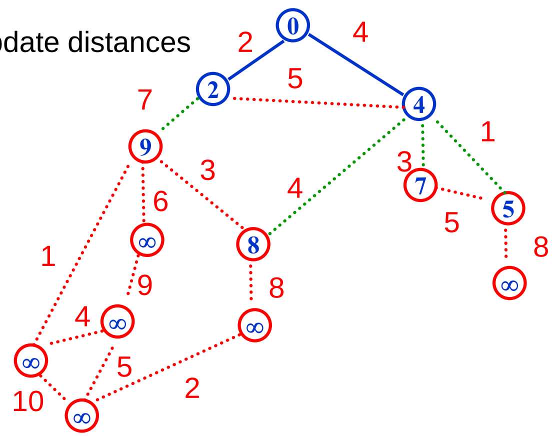
Dijkstra’s Algorithm

Add to S



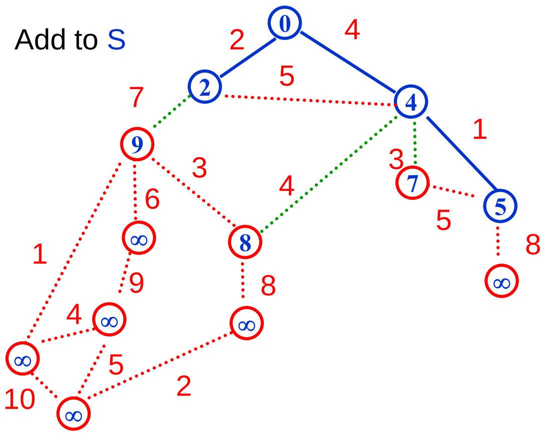
Dijkstra’s Algorithm

Update distances



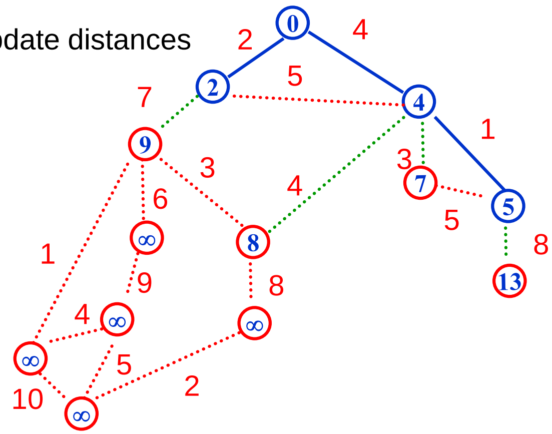
Dijkstra's Algorithm

Add to S



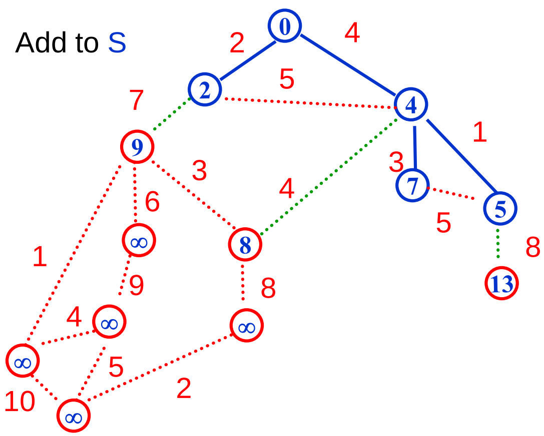
Dijkstra's Algorithm

Update distances



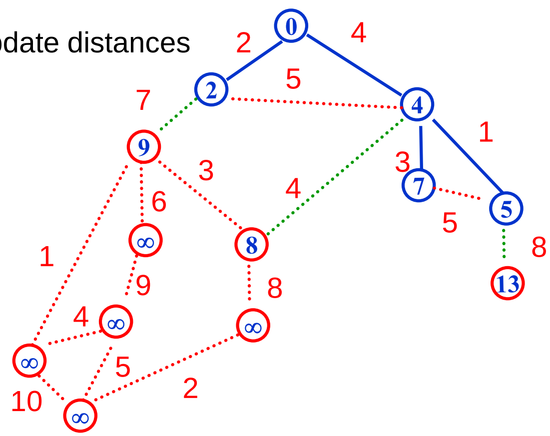
Dijkstra's Algorithm

Add to S



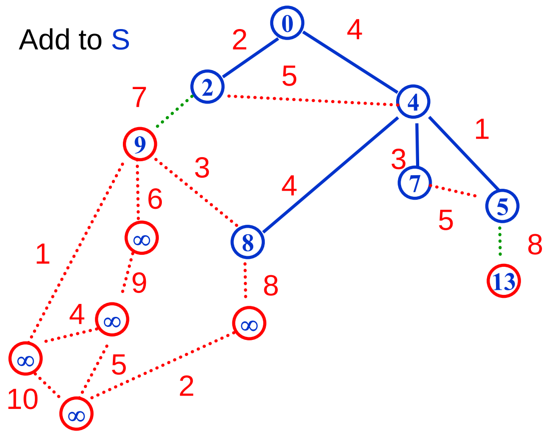
Dijkstra's Algorithm

Update distances



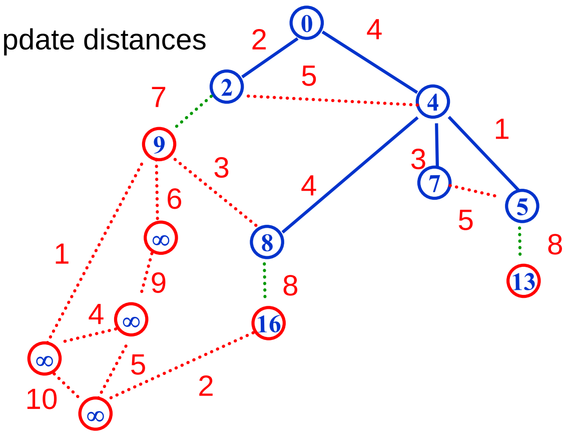
Dijkstra's Algorithm

Add to S



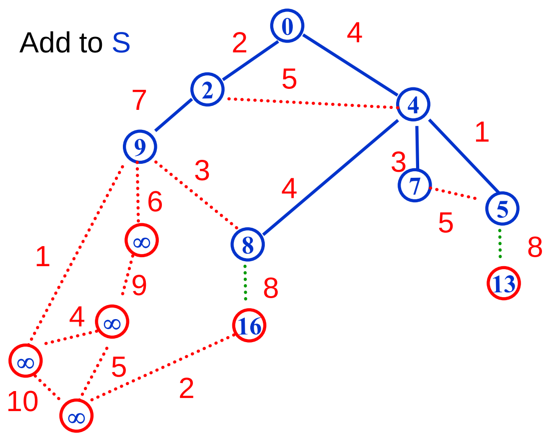
Dijkstra's Algorithm

Update distances



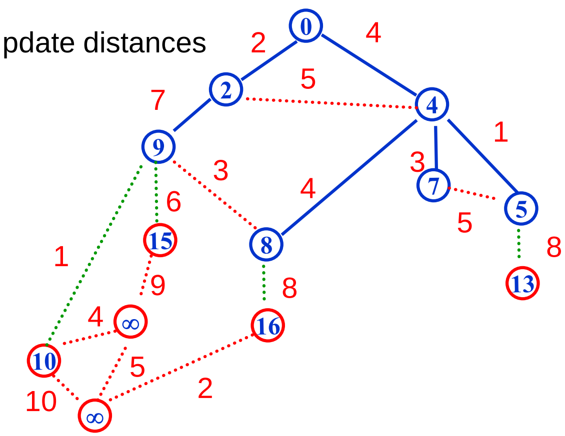
Dijkstra's Algorithm

Add to S



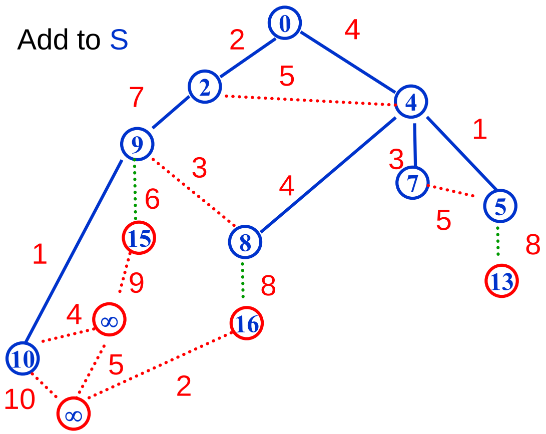
Dijkstra's Algorithm

Update distances



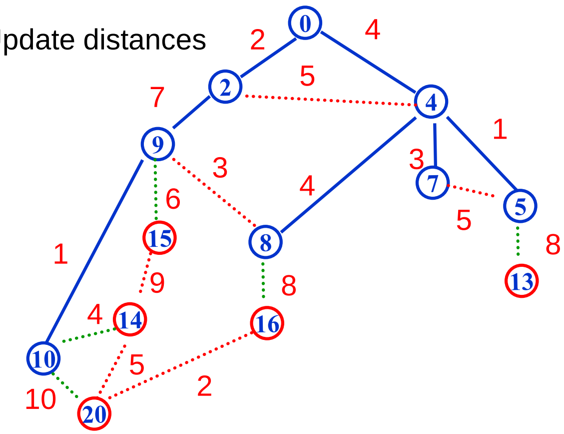
Dijkstra’s Algorithm

Add to S



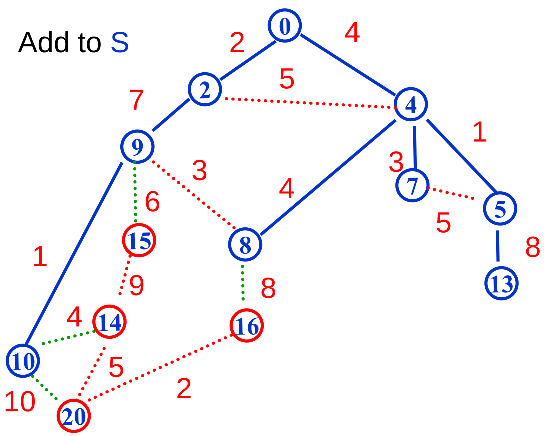
Dijkstra’s Algorithm

Update distances



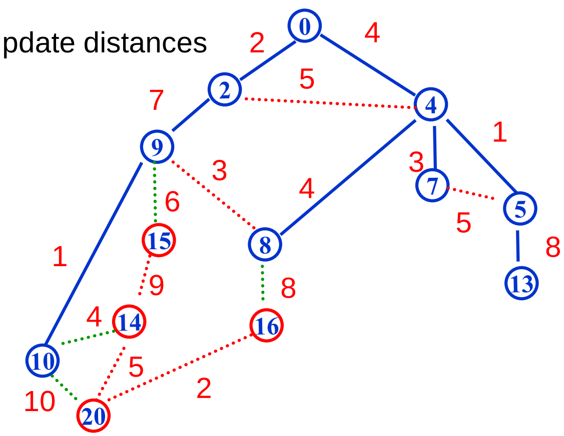
Dijkstra’s Algorithm

Add to S



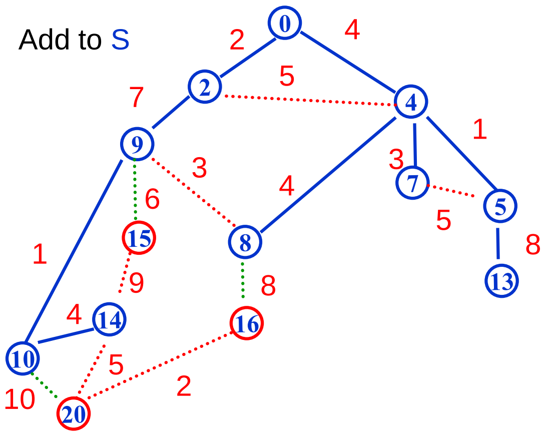
Dijkstra’s Algorithm

Update distances



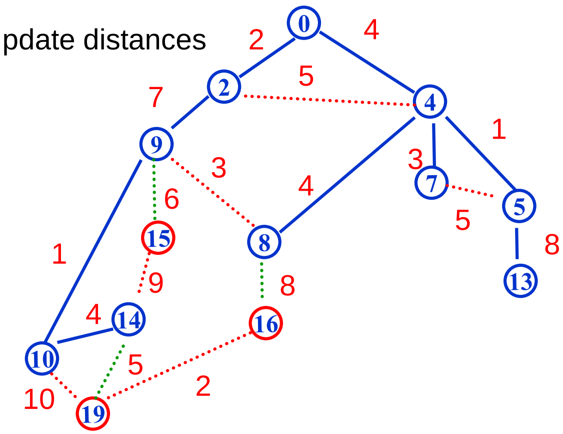
Dijkstra's Algorithm

Add to S



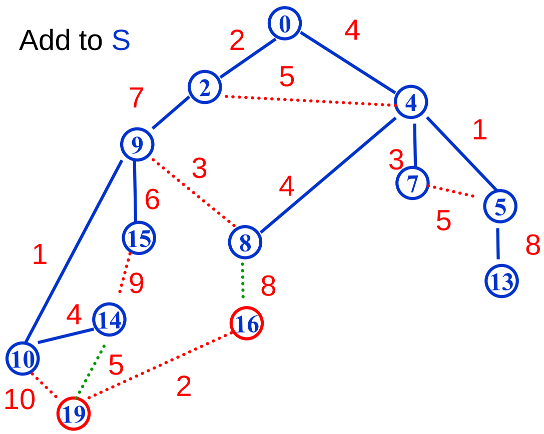
Dijkstra's Algorithm

Update distances



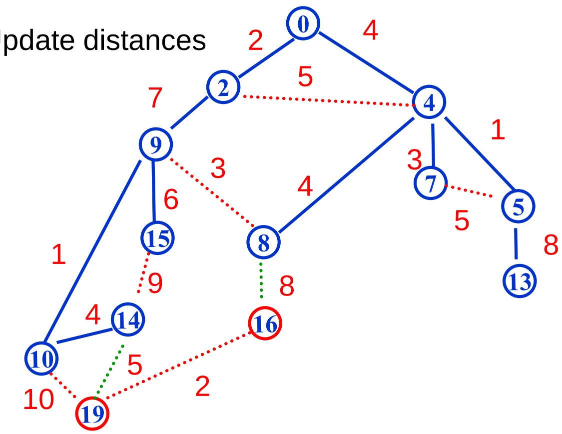
Dijkstra's Algorithm

Add to S



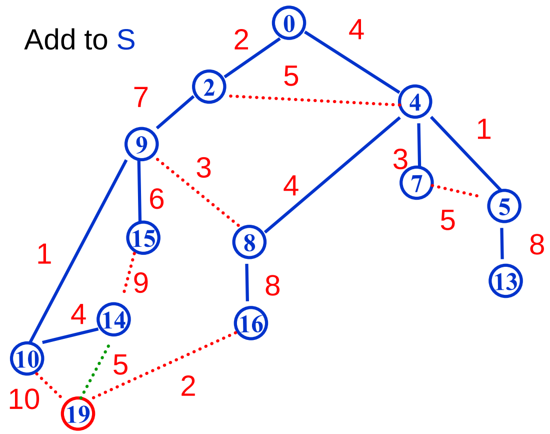
Dijkstra's Algorithm

Update distances



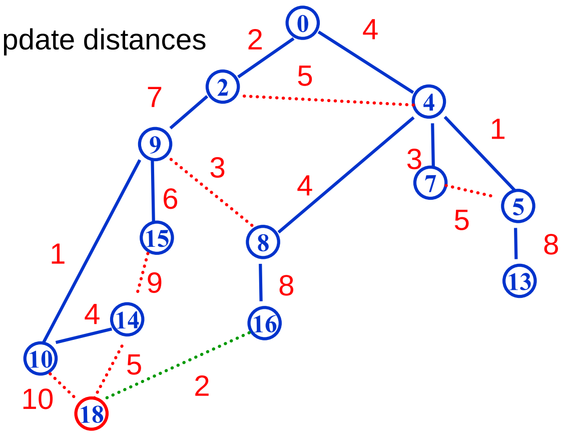
Dijkstra's Algorithm

Add to S



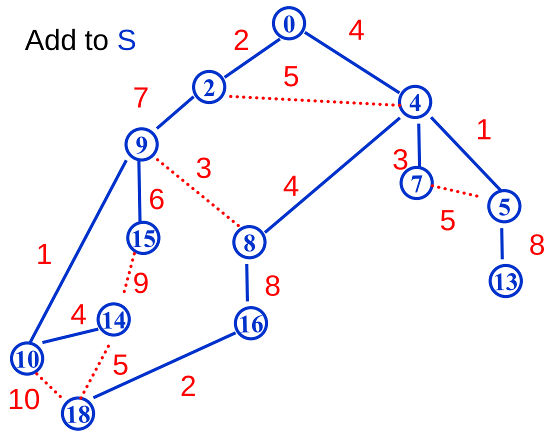
Dijkstra's Algorithm

Update distances



Dijkstra's Algorithm

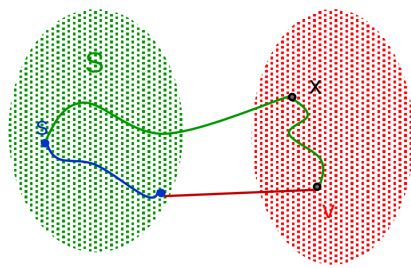
Add to S



Dijkstra's Algorithm Correctness

Suppose all distances to vertices in S are correct
and u has smallest current value in V-S

∴ distance value of vertex in V-S=length of shortest path from s
with only last edge leaving S



Suppose some other path to v and x= first vertex on this path not in S

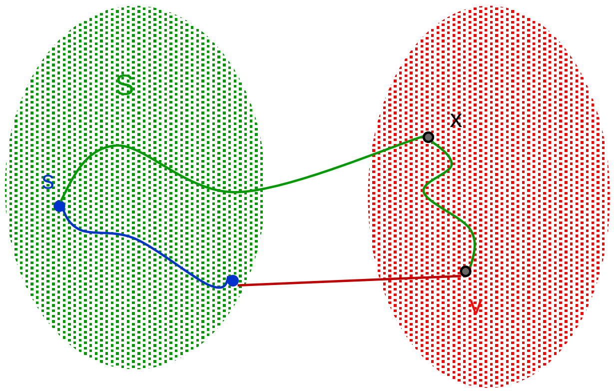
$d'(v) \leq d'(x)$

x-v path length ≥ 0

∴ other path is longer

Therefore adding v to S keeps correct distances

Dijkstra's Algorithm Correctness



Dijkstra's Algorithm

- Algorithm also produces a tree of shortest paths to v following pred links
  - From w follow its ancestors in the tree back to v
- If all you care about is the shortest path from v to w simply stop the algorithm when w is added to S

Implementing Dijkstra's Algorithm

- Need to
  - keep current distance values for nodes in V-S
  - find minimum current distance value
  - reduce distances when vertex moved to S

data structure review

- **Priority Queue:**
  - Elements each with an associated **key**
  - Operations
    - **Insert**
    - **Find-min**
      - Return the element with the smallest key
    - **Delete-min**
      - Return the element with the smallest key and delete it from the data structure
    - **Decrease-key**
      - Decrease the key value of some element
- Implementations
  - Arrays: O(n) time find/delete-min, O(1) time insert/decrease-key
  - Heaps: O(log n) time insert/decrease-key/delete-min, O(1) time find-min

Dijkstra's algorithm with priority queues

- For each vertex **u** not in tree maintain cost of current cheapest path through tree to **u**
  - Store **u** in priority queue with key = length of this path
- Operations:
  - n-1 insertions (each vertex added once)
  - n-1 delete-mins (each vertex deleted once)

    pick the vertex of smallest key, remove it from the priority queue and add its edge to the graph
  - <m decrease-keys (each edge updates one vertex)

Dijkstra's algorithm with priority queues

Priority queue implementations

- **Array**

  insert $O(1)$, delete-min $O(n)$, decrease-key $O(1)$

  total $O(n+n^2+m)=O(n^2)$
- **Heap**

  insert, delete-min, decrease-key all $O(\log n)$

  total $O(m \log n)$
- **d-Heap** ($d=m/n$)

  insert, decrease-key $O(\log_{m/n} n)$

  delete-min $O((m/n) \log_{m/n} n)$

  total $O(m \log_{m/n} n)$

Dijskstra's algorithm with priority queues