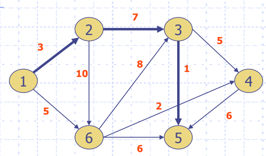
Graphs - Shortest Paths

- In a graph in which edges have costs …
- Find the shortest path from a source to a destination.
- Common algorithm for
  single-source shortest paths
  is due to Edsger Dijkstra
  - While finding the shortest path from a source to one destination,
  - *we can find the shortest paths to all destinations as well!*
- Applications: transportation planning …

Dijkstra's Algorithm - Data Structures

- For a graph,
  $G = (V, E)$
- Dijkstra's algorithm keeps *two* sets of vertices:
  S Vertices whose shortest paths already been determined
  $V\text{-}S$ Remainder
- Also
  d Best estimates of shortest path to each vertex
  π Predecessors for each vertex

The Shortest Path: from vertex 1 to vertex 5



Adjacency matrix

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

Predecessor Sub-graph

- Array of vertex indices, $\pi[j]$, $j = 1 .. |V|$
  - $\pi[j]$ contains the predecessor for node j
  - $\pi[j]$'s predecessor is in $\pi[\pi[j]]$, and so on ….
  - The edges in the predecessor sub-graph are
    $(\pi[j], j)$

Dijkstra's Algorithm - Operation

- Initialize d and π
  - For each vertex, j, in V
    - $d_j = \infty$ — Initial estimates are all ∞
    - π_j = nil — No connections
  - Source distance, $d_s = 0$
- Set S to empty
- While V-S is not empty
  - Sort V-S based on d
  - Add u, the closest vertex in V-S, to S — Add s first!
  - Relax all the vertices still in V-S connected to u

Dijkstra's Algorithm - Operation

- The Relaxation process

Relax the node v attached to node u

Edge cost matrix

```
relax( Node u, Node v, double w[][] )
    if (d[v] > d[u] + w[u][v]){
        d[v] = d[u] + w[u][v];
        pi[v] = u;
}
```

If the current best estimate to v is greater than the path through u ..

Update the estimate to v

Make v's predecessor point to u

Dijkstra's Algorithm - Full

◆ The Shortest Paths algorithm

Given a graph, g, and a source, s

```
shortest_paths( Graph g, Node s ){
    initialise_single_source( g, s );
    S = { 0 };        /* Make S empty */
    Q = Vertices(g); /* Put the vertices in a PQ */
    while (! Empty(Q)){
        u = removeMin( Q );
        AddNode( S, u ); /* Add u to S */
        for each vertex v in Adjacent( u )
            relax( u, v, w )
    }
}
```


Dijkstra's Algorithm - Initialise

◆ The Shortest Paths algorithm

Given a graph, g, and a source, s

Initialize d, π, S, vertex Q

```
shortest_paths( Graph g, Node s ){
    initialise_single_source( g, s );
    S = { 0 };        /* Make S empty */
    Q = Vertices(g) /* Put the vertices in a PQ */
    while (! Empty(Q)){
        u = removeMin( Q );
        AddNode( S, u ); /* Add u to S */
        for each vertex v in Adjacent( u )
            relax( u, v, w );
    }
}
```


Dijkstra's Algorithm - Loop

◆ The Shortest Paths algorithm

Given a graph, g, and a source, s

While there are still nodes in Q

Greedy!

```
shortest_paths                s ){
    initialise                g, s );
    S = { 0 };           Make S empty */
    Q = Vertices( g ); /* Put the vertices in a PQ */
    while (! Empty(Q)){
        u = removeMin( Q );
        AddNode( S, u ); /* Add u to S */
        for each vertex v in Adjacent( u )
            relax( u, v, w );
    }
}
```


Dijkstra's Algorithm - Relax neighbours

◆ The Shortest Paths algorithm

Given a graph, g, and a source, s

Update the estimate of the shortest paths to all nodes attached to u

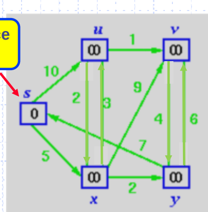
Greedy!

```
shortest_paths(                   )
    initialise_s                  s )
    S = { 0 }                    mpty */
    Q = Vertices( g ) /     Put the vertices in a PQ */
    while (! Empty(Q)){
        u = removeMin(             ;
        AddNode( S, u );  /* Add u to S */
        for each vertex v in Adjacent( u )
            relax( u, v, w );
    }
}
```


Dijkstra's Algorithm - Operation

◆ Initial Graph

Source Mark 0

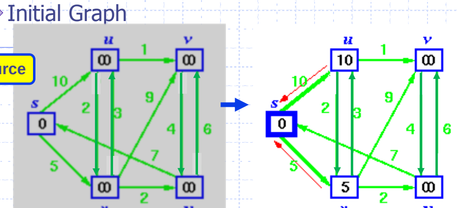


Distance to all nodes marked ∞

Dijkstra's Algorithm - Operation

◆ Initial Graph

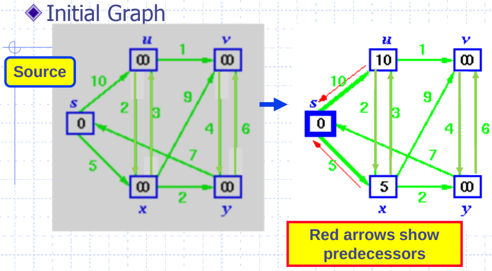
Source



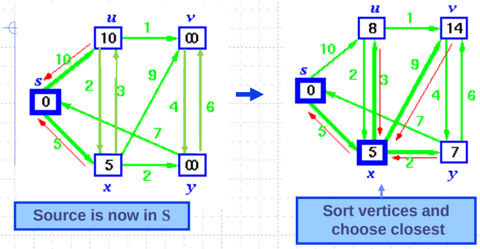
Relax vertices adjacent to source

Dijkstra's Algorithm - Operation

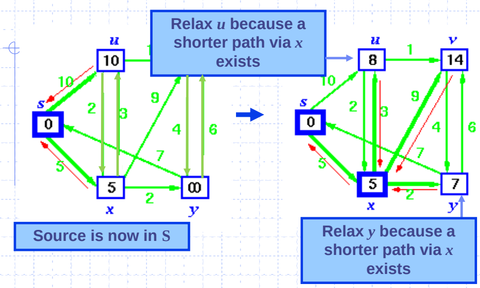
◆ Initial Graph



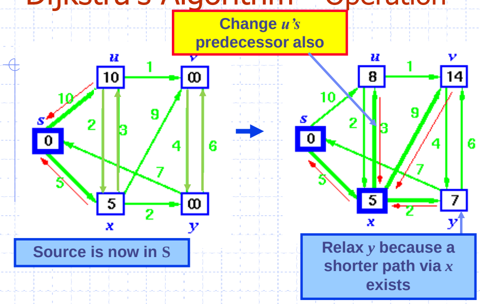
Dijkstra's Algorithm - Operation



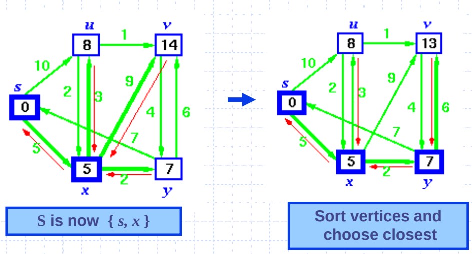
Dijkstra's Algorithm - Operation



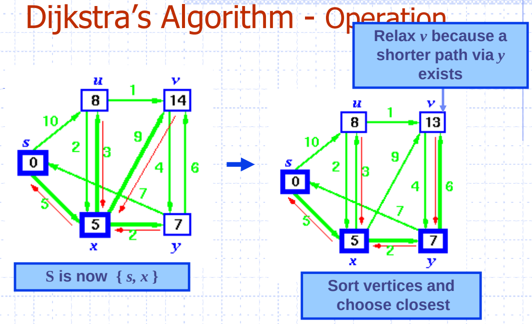
Dijkstra's Algorithm - Operation



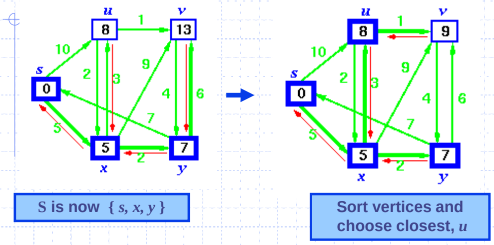
Dijkstra's Algorithm - Operation



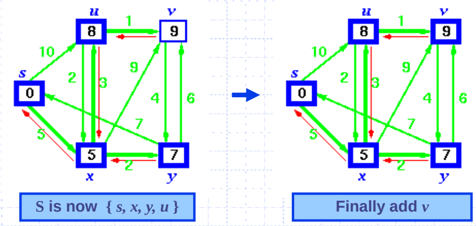
Dijkstra's Algorithm - Operation



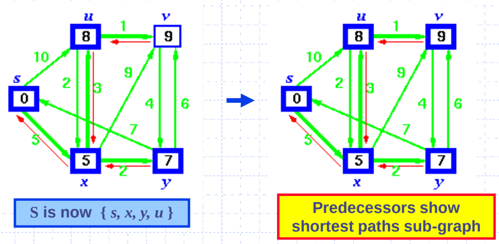
Dijkstra's Algorithm - Operation



Dijkstra's Algorithm - Operation



Dijkstra's Algorithm - Operation



Dijkstra's Algorithm - Time Complexity

- Dijkstra's Algorithm
  - A key step is sorting the remaining vertices after each vertex joins *S*. This can be done by creating a heap as a priority queue.
  - Complexity is
    - $O(|E|) \quad (\sum_{v \ni V}(\boldsymbol{deg}(v) + \log|\boldsymbol{V}|))$
    - ***or*** $O(n^2) \quad (n(n + \log n) = n^2 + n \log n)$

      for a dense graph with $n = |V|$ **and** $|E| \approx |V|^2$

Graphs: Minimum Spanning Trees

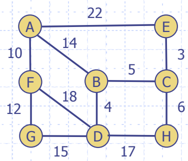
- A spanning tree of an undirected connected graph is a tree that contains all the vertices of the graph.
- If the graph has weights with the edges, a minimum spanning tree of the graph is a spanning tree of the smallest weight.
- The weight of a tree is the sum of the weights on all the edges in the tree.
- Applications: cable network design, etc.

Prim's Algorithm

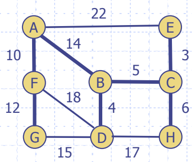
- The initial subtree contains a single vertex.
- In each iteration, the algorithm expands the subtree by adding the nearest vertex that is not in the tree.
- After adding a vertex to the subtree, the algorithm re-calculates the distances of the remaining vertices to the subtree.
- The algorithm terminates when all the vertices are contained in the subtree.

Example

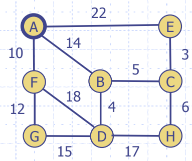
Graph



Minimum spanning tree

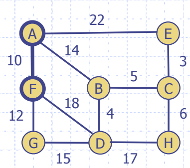


Example



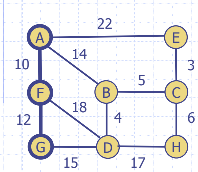
Tree vertices	Remaining vertices	
A(-,-)	B(A,14)	C(-,∞)
	D(-,∞)	E(A,22)
	F(A,10)	G(-,∞)
	H(-,∞)	

Example



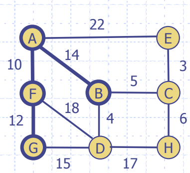
Tree vertices	Remaining vertices	
A(-,-)	B(A,14)	C(-,∞)
F(A,10)	D(F,18)	E(A,22)
		G(F,12)
	H(-,∞)	

Example



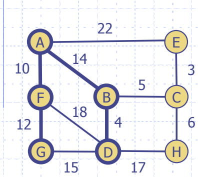
Tree vertices	Remaining vertices	
A(-,-)	B(A,14)	C(-,∞)
F(A,10)	D(G,15)	E(A,22)
G(F,12)		
	H(-,∞)	

Example



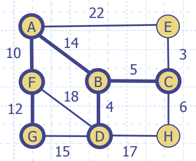
Tree vertices	Remaining vertices	
A(-,-)		C(B,5)
F(A,10)	D(B,4)	E(A,22)
G(F,12)		
B(A,14)	H(-,∞)	

Example



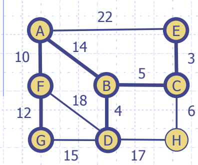
Tree vertices	Remaining vertices	
A(-,-)		C(B,5)
F(A,10)		E(A,22)
G(F,12)		
B(A,14)	H(D,17)	
D(B,4)		

Example



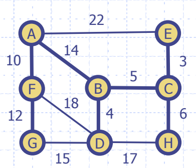
Tree vertices	Remaining vertices
A(-,-)	
F(A,10)	E(C,3)
G(F,12)	
B(A,14)	H(C,6)
D(B,4)	
C(B,5)	

Example



Tree vertices	Remaining vertices
A(-,-)	
F(A,10)	
G(F,12)	
B(A,14)	H(C,6)
D(B,4)	
C(B,5)	
E(C,3)	

Example



Tree vertices	Remaining vertices
A(-,-)	
F(A,10)	
G(F,12)	
B(A,14)	
D(B,4)	
C(B,5)	
E(C,3)	
H(C,6)	

Prim's Algorithm - Time Complexity

- Prim's Algorithm
  - Prim's algorithm has the same time complexity.
  - Complexity is
    - $O(|E|)$ ***or***
    - $O(n^2)$

      for a dense graph with $n = |V|$ **and** $|E| \approx |V|^2$

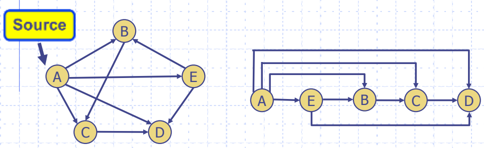
Graphs - Topological Ordering

- Topological ordering is an operation on directed acyclic graphs (DAGs).
- A topological ordering for the vertices in graph *G* is a sequential list *L* of the vertices, such that if there is a directed edge from vertex *A* to vertex *B* in *G*, then *A* comes before *B* in *L*.
- Applications: scheduling of tasks from the given dependencies among tasks ...

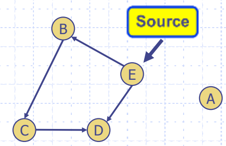
Topological Ordering

- An algorithm for topological ordering is *source removal*.
- In each step of the algorithm, a *source* is identified. A source is a vertex with no incoming edges. The source is removed from the graph along with all its outgoing edges. The vertex is then added at the end of the list.
- The process continues till all vertices are removed from the graph.

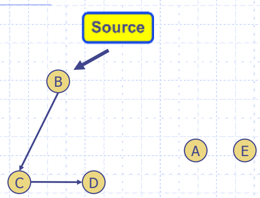
Example



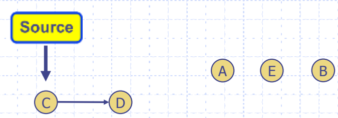
Example



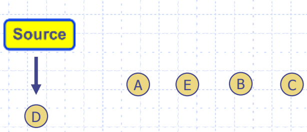
Example



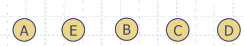
Example



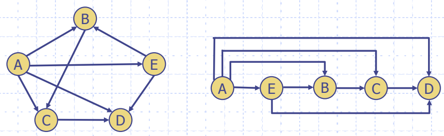
Example



Example



Example



Source Removal - Time Complexity

- Source removal
  - After removing a vertex (and its outgoing edges), the algorithm examines the remaining vertices for a source.
  - Complexity is
    - $O(|E|)$ ***or***
    - $O(n^2)$

    for a dense graph with $n = |V|$ **and** $|E| \approx |V|^2$