Data Structures – Week #10

Graphs & Graph Algorithms

Outline

- Motivation for Graphs
- Definitions
- Representation of Graphs
- Topological Sort
- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Single-Source Shortest Path Problem (SSSP)
  - Dijkstra's Algorithm
- Minimum Spanning Trees
  - Prim's Algorithm
  - Kruskal's Algorithm

Graphs & Graph Algorithms

Motivation

- Graphs are *useful structures for solving many problems* computer science is interested in including but not limited to
  - *Computer and telephony networks*
  - *Game theory*
  - *Implementation of automata*

Graph Definitions

- A *graph* $G=(V,E)$ consists a set of *vertices* V and a set of *edges* E.
- An *edge* $(v,w) \in E$ has a starting vertex v and an ending vertex w. An edge sometimes is called an *arc*.
- If the pair is ordered, then the graph is *directed*. Directed graphs are also called *digraphs*.
- Graphs which have a third component called a *weight* or *cost* associated with each edge are called *weighted graphs*.

Adjacency Set and Being *Adjacent*

- Vertex v is *adjacent* to u iff $(u,v) \in E$. In an undirected graph with $e=(u,v)$, u and v are adjacent to each other.
- In Fig. 6.1, the vertices v, w and x form the *adjacency set* of u or

  $Adj(u)=\{v,w,x\}$.

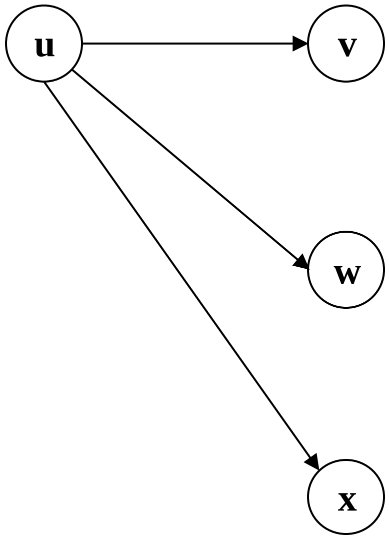


Figure 6.1. Adjacency set of *u*

Path Definitions

- A *path* in a graph is a sequence of vertices w_1, w_2, …, w_n where each edge $(w_i, w_{i+1}) \in E$ for $1 \leq i < n$.
- The *length of a path* is the number of edges on the path, (i.e., n-1 for the above path). A path from a vertex to itself, containing no edges has a length 0.
- An edge (v,v) is called a *loop*.
- A *simple* path is one in which all vertices, except possibly the first and the last, are distinct.

More Definitions

- A *cycle* is a path such that the vertex at the destination of the last edge is the source of the first edge.
  - A digraph is *acyclic* iff it has no cycles in it.
- *In-degree* of a vertex is the *number of edges arriving* at that vertex.
- *Out-degree* of a vertex is the *number of edges leaving* that vertex.

Connectedness

An undirected graph is *connected* if there exists a path from every vertex to every other vertex.

- A digraph with the same property is called *strongly connected*.
- If a digraph is not strongly connected, but the underlying graph (i.e., the undirected graph with the same topology) is connected, then the digraph is said to be *weakly connected*.
- A graph is *complete* or *fully connected* if there is an edge between every pair of vertices.

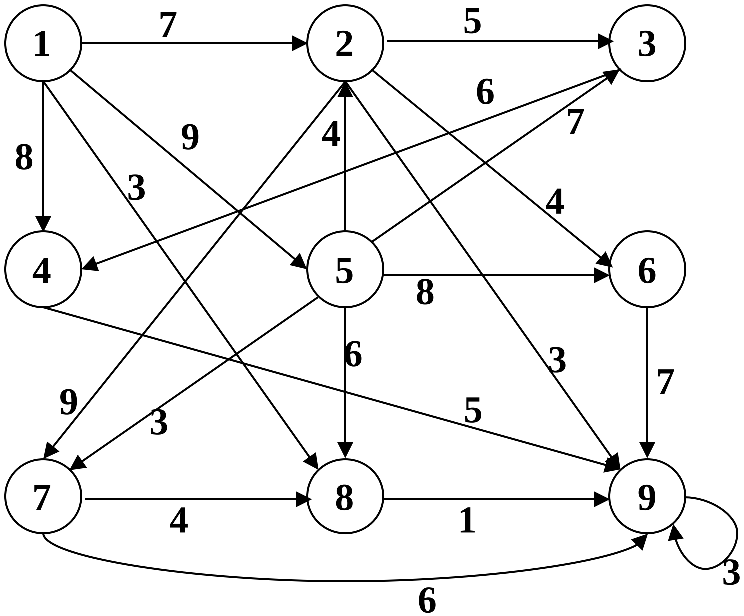
Representation of Graphs

- Two ways to represent graphs:
  - *Adjacency **matrix** representation*
  - *Adjacency **list** representation*

Adjacency Matrix Representation

- Assume you have *n* vertices.
- In a boolean array with n^2 elements, where each element represents the connection of a pair of vertices, you assign *true* to those elements that are connected by an edge and *false* to others.
- *Good for dense graphs!*
- Not very efficient for sparse (i.e., not dense) graphs.
- Space requirement: $O(|V|^2)$.

Adjacency matrix representation (AMR)



	1	2	3	4	5	6	7	8	9
1	∞	7	∞	8	9	∞	∞	3	∞
2	∞	∞	5	∞	∞	4	9	∞	3
3	∞	∞	∞	6	∞	∞	∞	∞	∞
4	∞	∞	∞	∞	∞	∞	∞	∞	5
5	∞	4	7	∞	∞	8	3	6	∞
6	∞	∞	∞	∞	∞	∞	∞	∞	7
7	∞	∞	∞	∞	∞	∞	∞	4	6
8	∞	∞	∞	∞	∞	∞	∞	∞	1
9	∞	∞	∞	∞	∞	∞	∞	∞	3

Disadvantage: Waste of space for sparse graphs

Advantage: Fast access

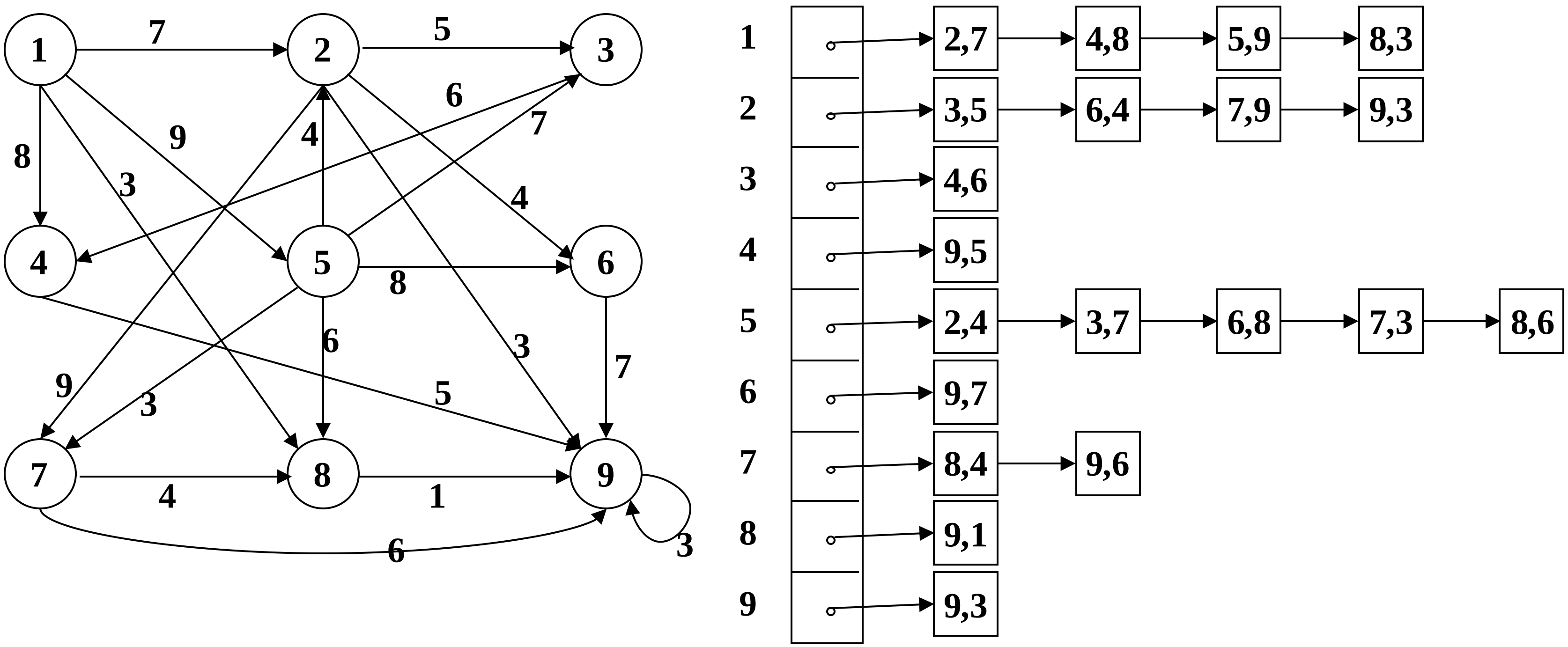
Adjacency List Representation

Assume you have n vertices.

- We employ an array with n elements, where i^{th} element represents vertex i in the graph. Hence, element i is a header to a list of vertices adjacent to the vertex i.
- Good for sparse graphs
- Space requirement: $O(|E|+|V|)$.

Adjacency list representation (ALR)

array index: source vertex;
first number: destination vertex;
second number: cost of the corresponding edge



Disadvantage: Sequential search among edges of a node
Advantage: Minimum space requirement

Topological Sort

- *Topological sort* is an ordering of vertices in an *acyclic digraph* such that if there is a path from v_i to v_j, then v_j appears after v_i in the ordering.

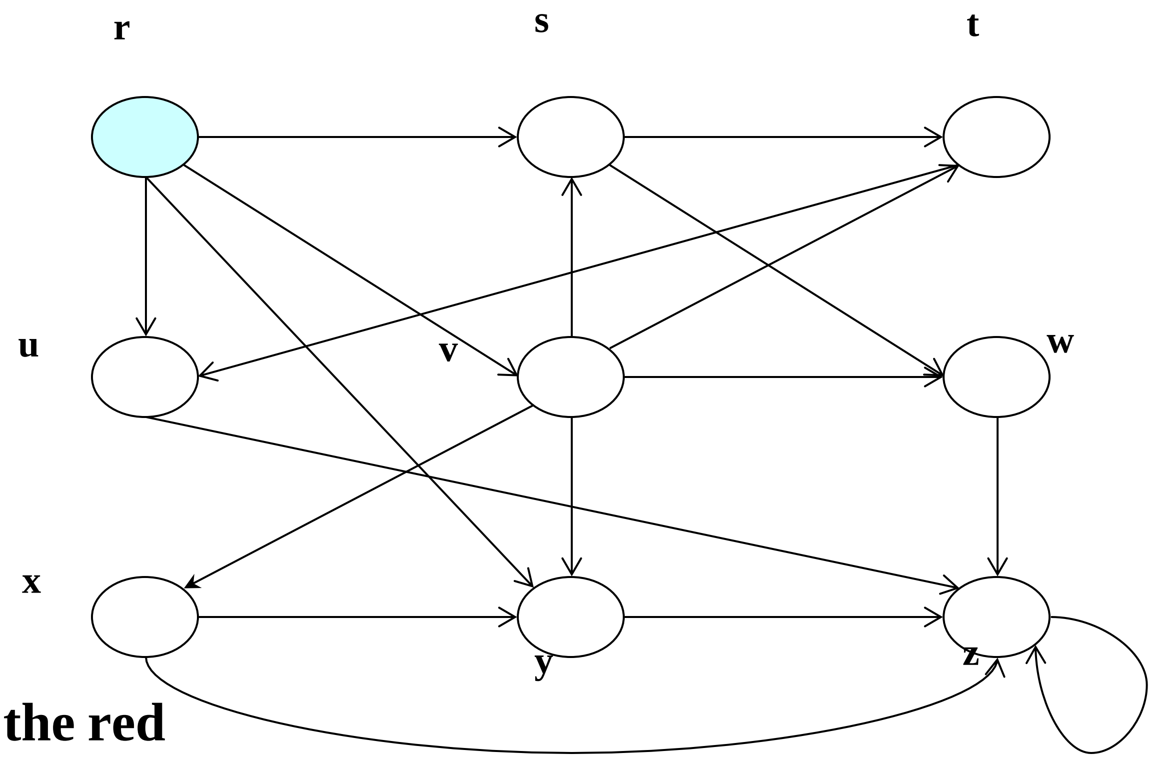
- Example: course prerequisite requirements.

Algorithm for Topological Sort*

```
Void Toposort ()
{
  Queue Q; int ctr=0; Vertex v,w;
  Q=createQueue(NumVertex);
  for each vertex v
      if (indegree[v] == 0) enqueue(v,Q);
  while (!IsEmpty(Q)) {
     v=dequeue(Q); topnum[v]=++ctr;
     for each w adjacent to v
       if (--indegree[w] == 0) enqueue(w,Q);
  }
  if (ctr != NumVertex) report error ('graph cyclic!')
  free queue;
}
```

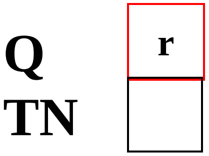
***From [2]**

An Example to Topological Sort

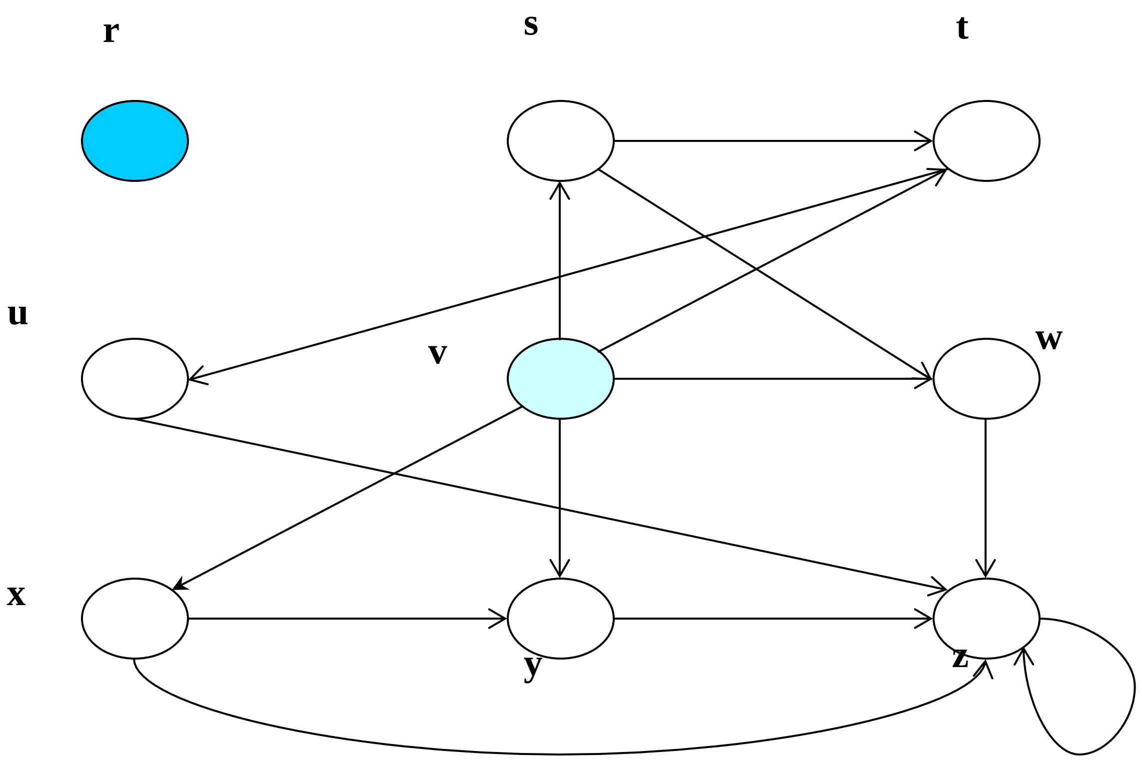


Q is indicated by the red squares.

TN keeps track of the order in which the vertices are processed.

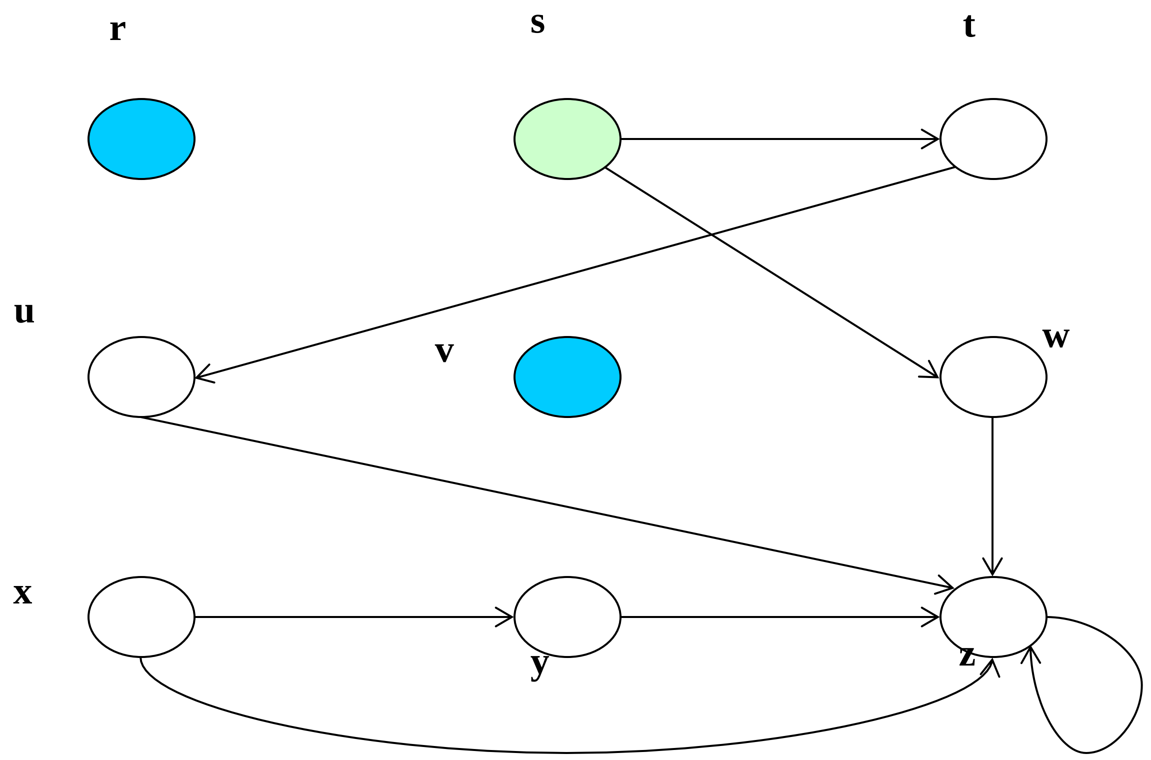


An Example to Topological Sort



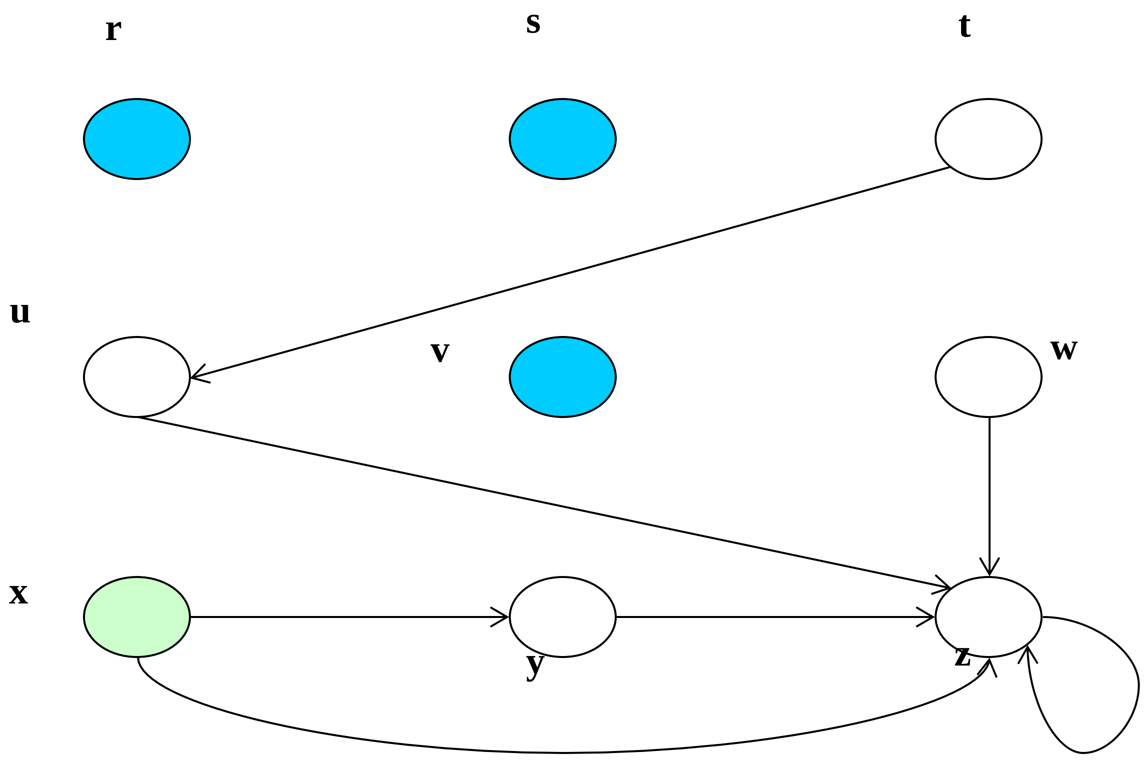
Q	r	v
TN	0	

An Example to Topological Sort



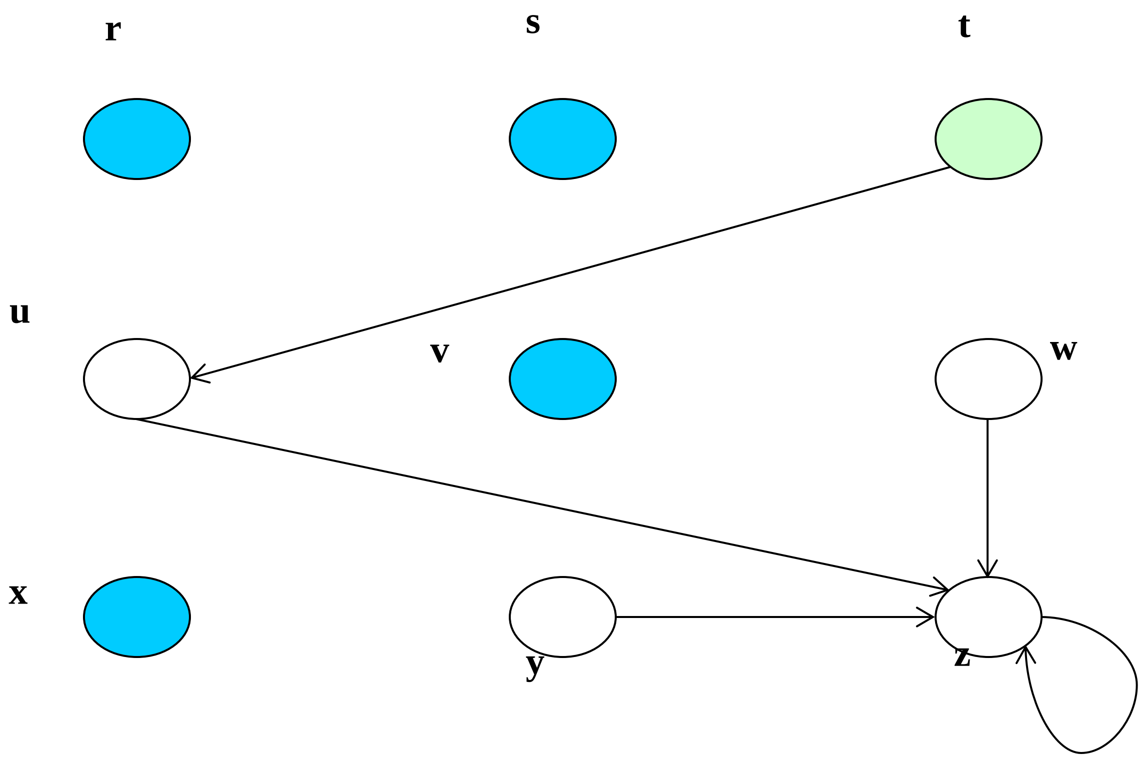
Q	**r**	**v**	**s**	**x**
TN	**0**	**1**		

An Example to Topological Sort



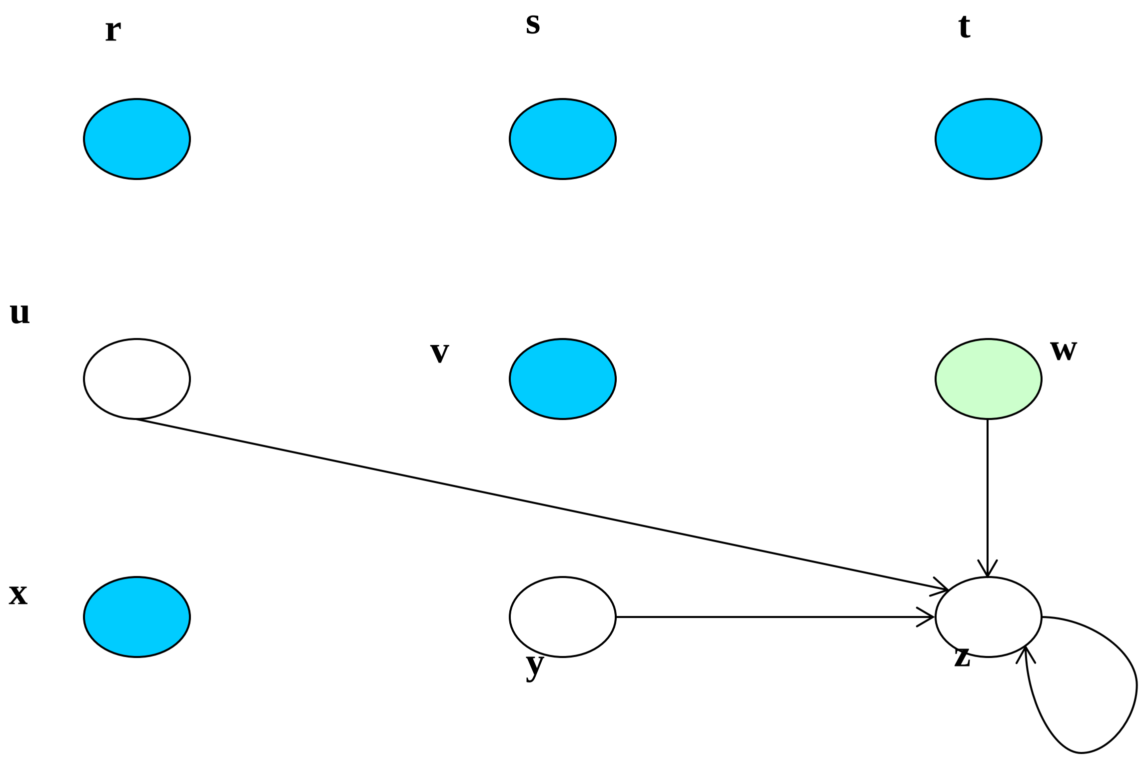
Q	r	v	s	x	t	w
TN	0	1	2			

An Example to Topological Sort



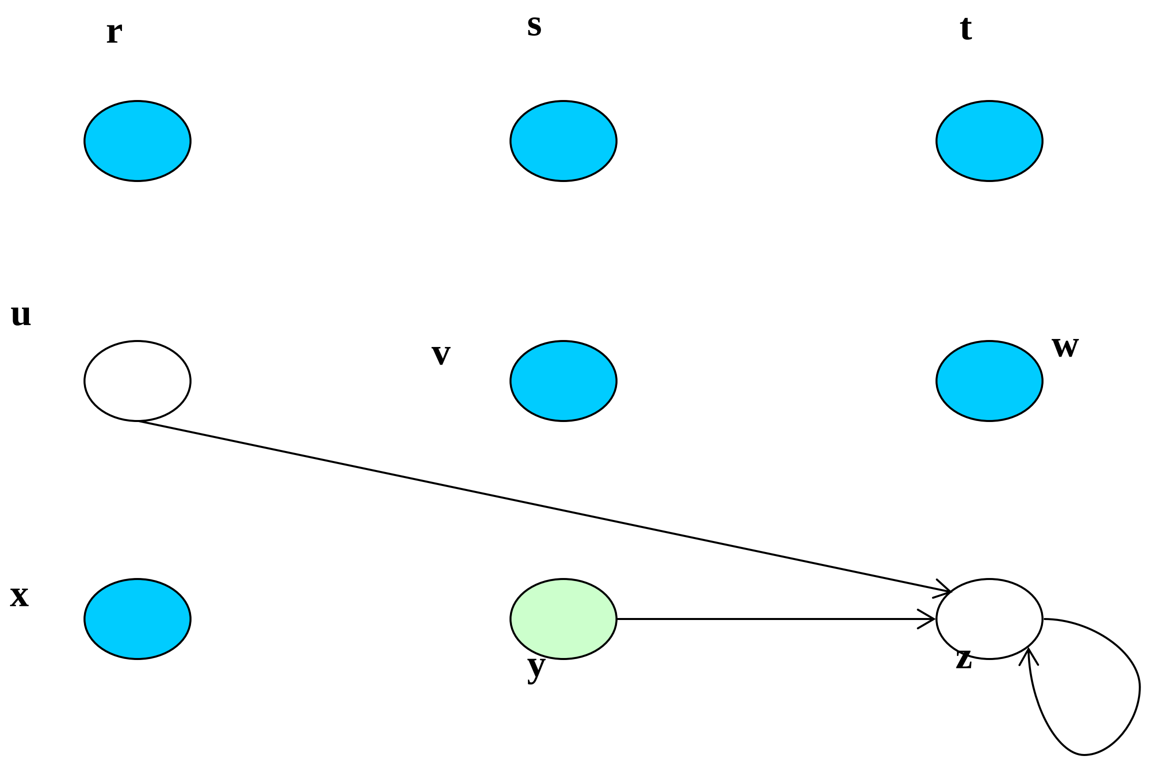
Q	**r**	**v**	**s**	**x**	**t**	**w**	**y**
TN	**0**	**1**	**2**	**3**			

An Example to Topological Sort



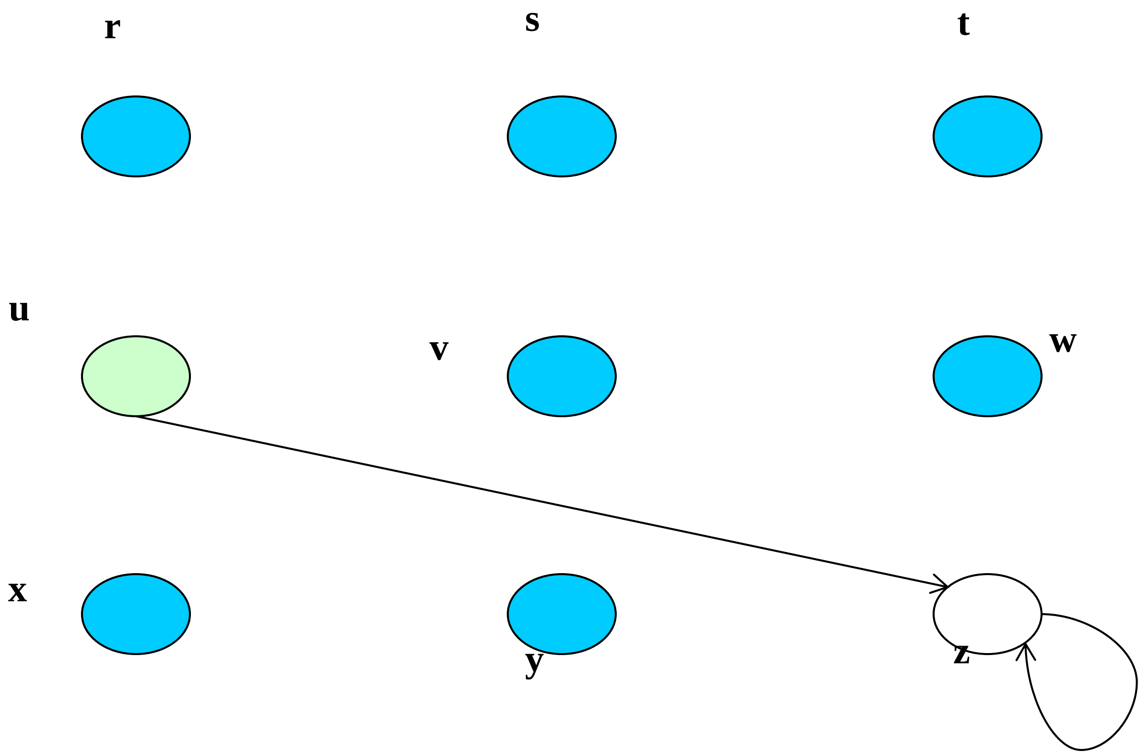
Q	**r**	**v**	**s**	**x**	**t**	**w**	**y**	**u**
TN	**0**	**1**	**2**	**3**	**4**			

An Example to Topological Sort



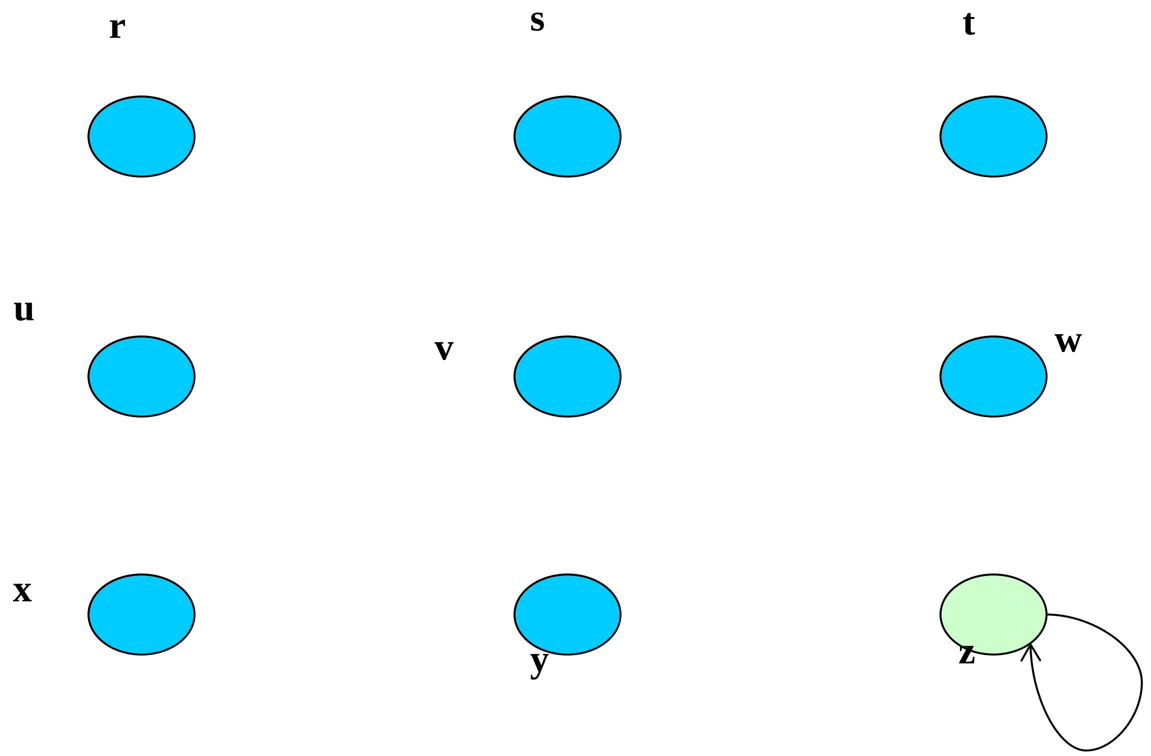
Q	r	v	s	x	t	w	y	u
TN	0	1	2	3	4	5		

An Example to Topological Sort



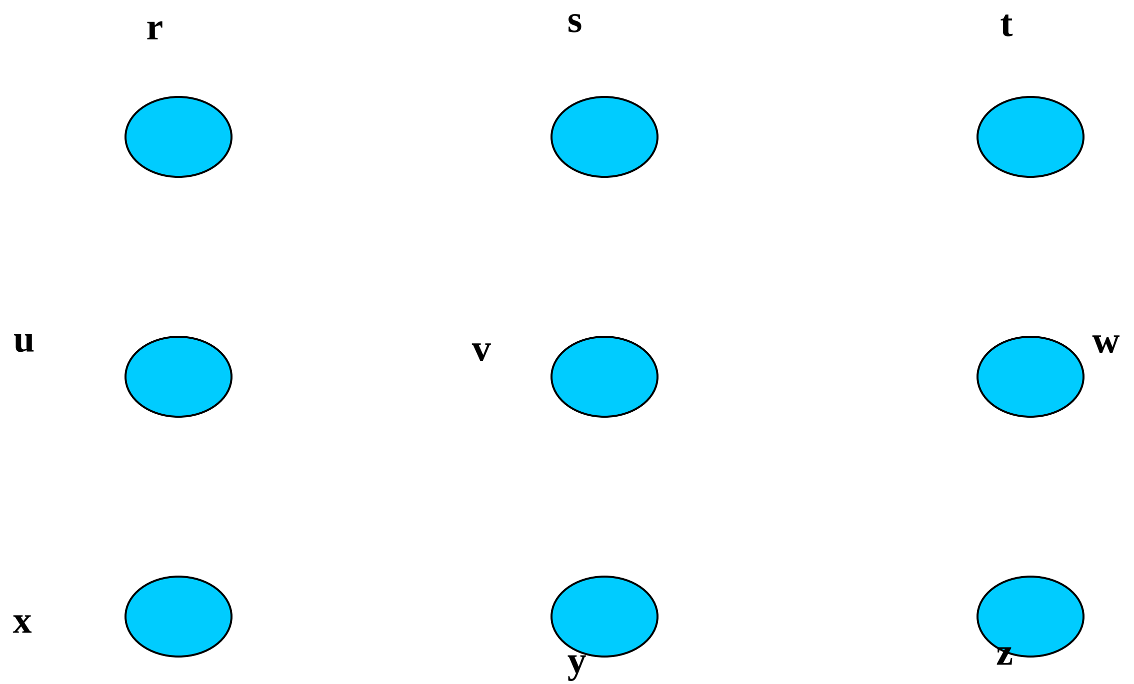
Q	r	v	s	x	t	w	y	u
TN	0	1	2	3	4	5	6	

An Example to Topological Sort



Q	r	v	s	x	t	w	y	u	z
TN	0	1	2	3	4	5	6	7	

An Example to Topological Sort



Q	**r**	**v**	**s**	**x**	**t**	**w**	**y**	**u**	**z**
TN	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Breadth-First Search (BFS)

- Given a graph, *G*, and a source vertex, *s*, breadth-first search (BFS) checks to discover every vertex reachable from *s*.
- BFS discovers vertices reachable from *s* in a breadth-first manner.
- That is, vertices a distance of *k* away from *s* are systematically discovered before vertices reachable from *s* through a path of length *k+1*.

Breadth-First Search (BFS)

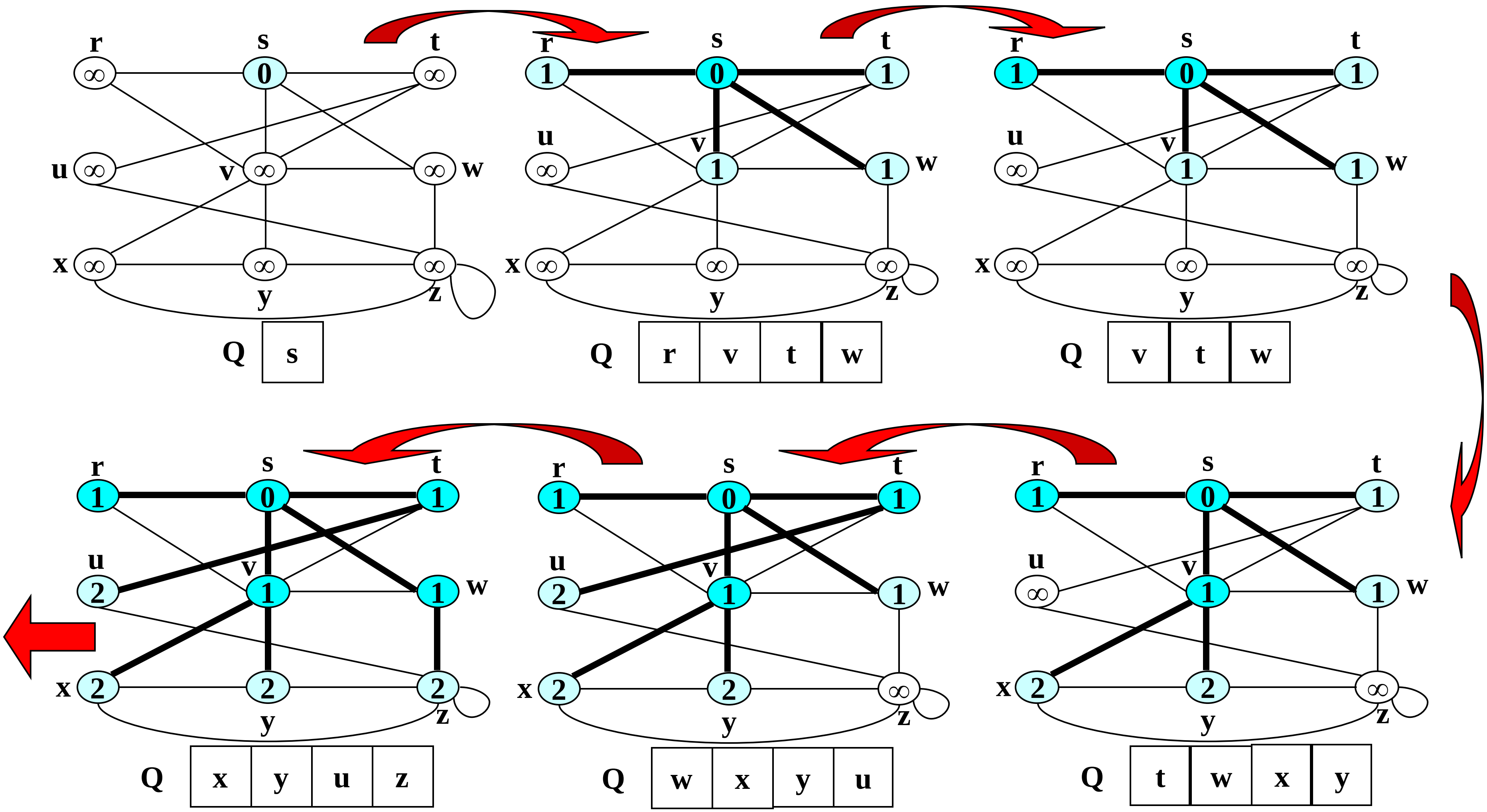
- To follow how the algorithm proceeds, BFS colors each vertex white, gray or black.
- Unprocessed nodes are colored white while vertices discovered (encountered during search) turn to gray. Vertices processed (i.e., vertices with all neighbors discovered) become black.
- Algorithm terminates when all vertices are visited.

Algorithm for Breadth-First Search*

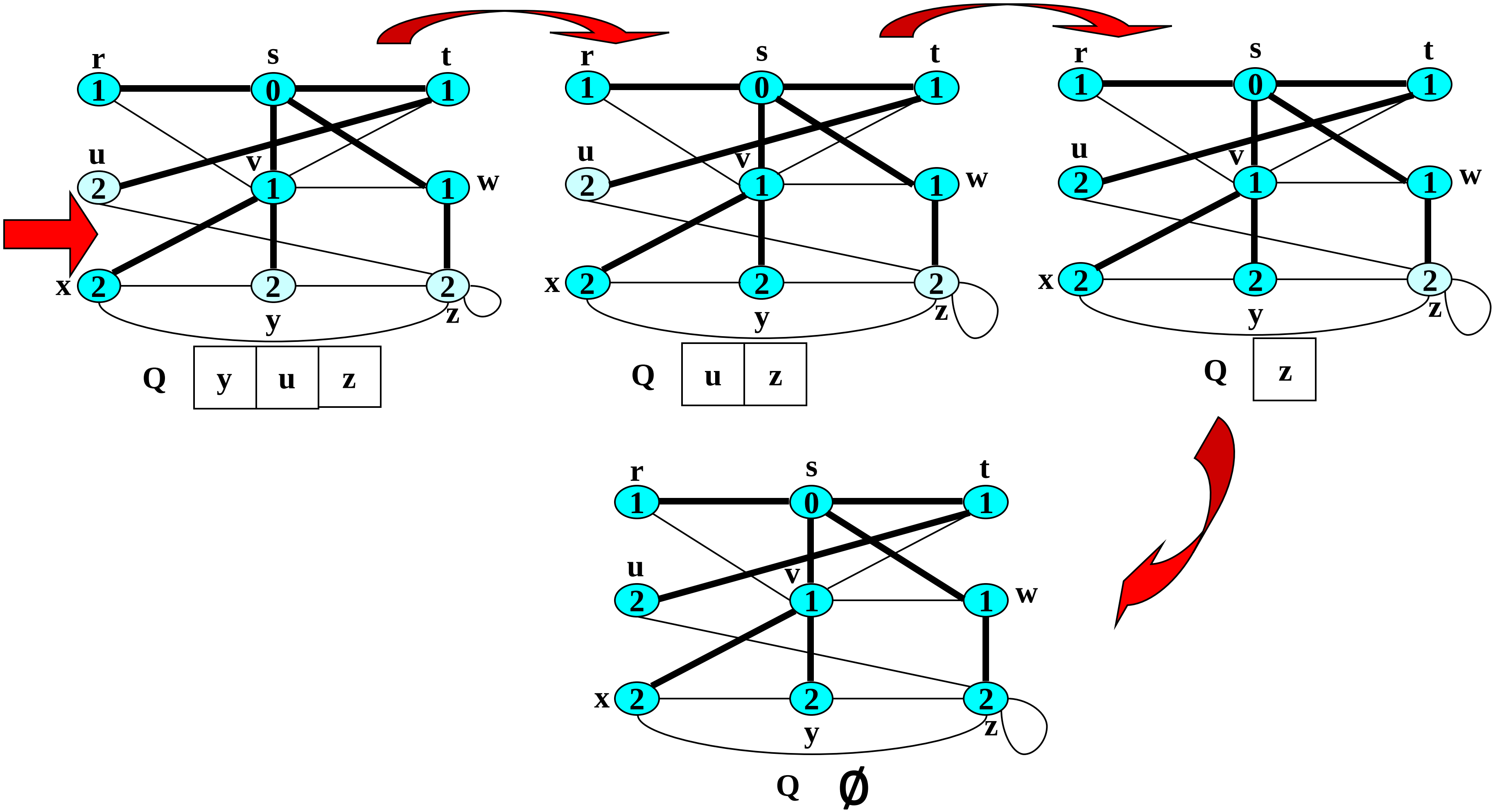
```
BFS(Graph G, Vertex s)
{
// initialize all vertices
    for each vertex u∈V[G]-{s} {
        color [u]=white;
        dist[u]=∞;
        from[u]=NULL;
    }
    color[s]=gray;
    dist[s]=0;
    from[s]=NULL;
    Q={}; enqueue(Q,s);
    while (!isEmpty(Q)) {
        u=dequeue(Q);
        for each v ∈Adj[u]
            if (color [v]==white) {
                color[v]=gray;
                dist[v]=dist[u]+1;
                from[v]=u;
                enqueue(Q,v);
            }
        color[u]=black;
    }
}
```

***From [1]**

An Example to BFS



Rest of Example



Depth-First Search (DFS)

- Unlike in BFS, *depth-first search* (DFS), performs a search going deeper in the graph.
- The search proceeds discovering vertices that are deeper on a path and looks for any left edges of the most recently discovered vertex u.
- If all edges of u are found, DFS backtracks to the vertex t which u was discovered from to find the remaining edges.

Algorithm for Depth-First Search*

```
DFS(Graph G, Vertex s)
{
// initialize all vertices
    for each vertex u∈V[G] {
       color [u]=white;
       from[u]=NULL;
    }
    time=0;
    for each vertex u∈V[G]
       if (color [u]==white)
           DFS-visit(u);
}
```

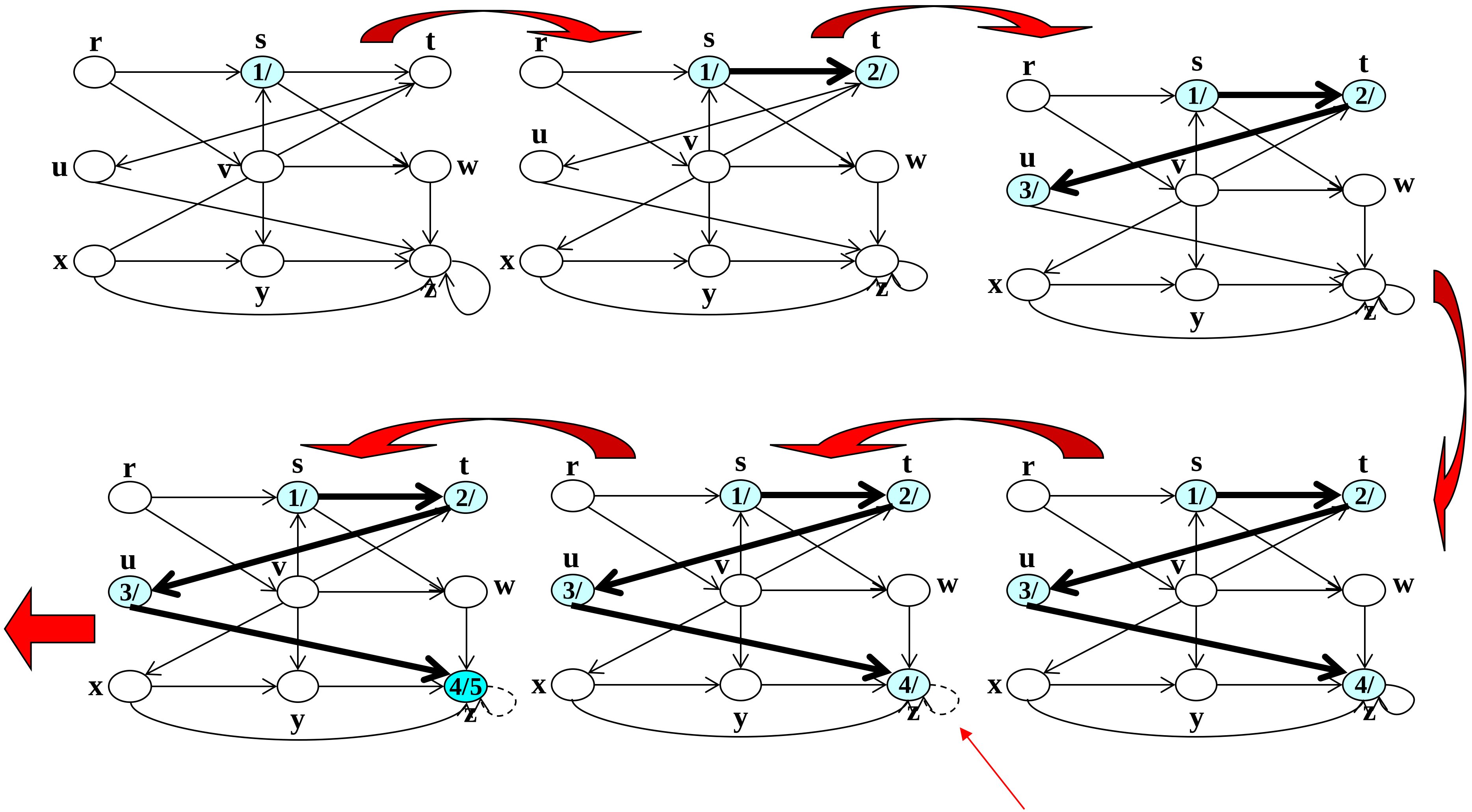
```
DFS-visit(u)
{
     color[u]=gray; //u just discovered
     time++;
     d[u]=time;
     for each v ∈ Adj[u] //check edge (u,v)
           if (color[v] == white) {
                      from[v]=u;
                      DFS-visit(v); //recursive call
           }
     color[u]=black; // u is done processing
     f[u] = time++;
}
```

***From [1]**

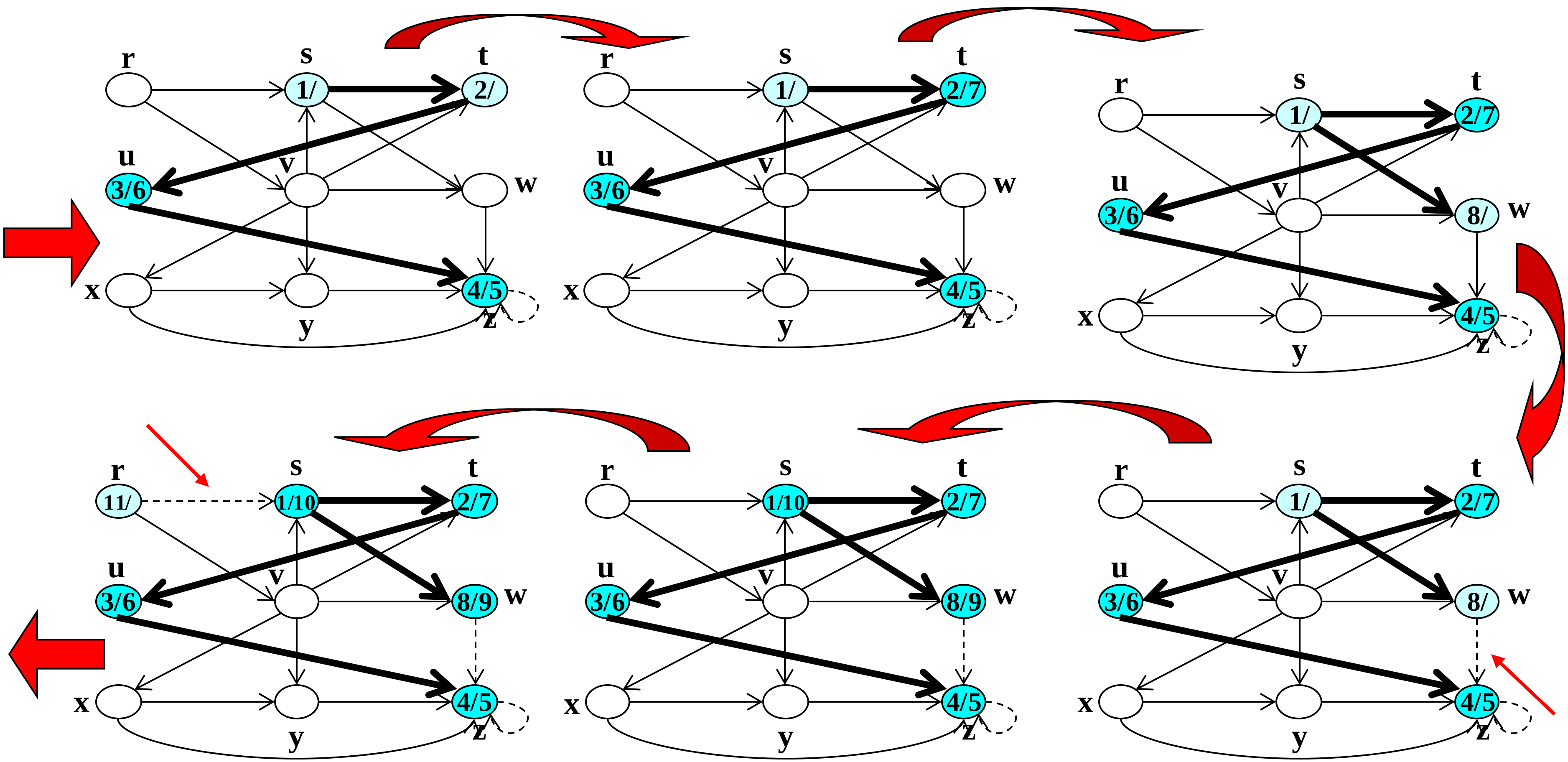
Depth-First Search

- The function DFS() is a “manager” function calling the recursive function DFS-visit(u) for each vertex in the graph.
- DFS-visit(u) starts by **graying** the vertex *u* just discovered. Then it recursively visits and discovers (and hence grays) all those nodes *v* in the adjacency set of *u*, *Adj[u]*. At the end, *u* is finished processing and turns to **black**.
- time in DFS-visit(u) time-stamps each vertex *u* when
  - *u* is discovered using d[u]
  - *u* is done processing using f[u].

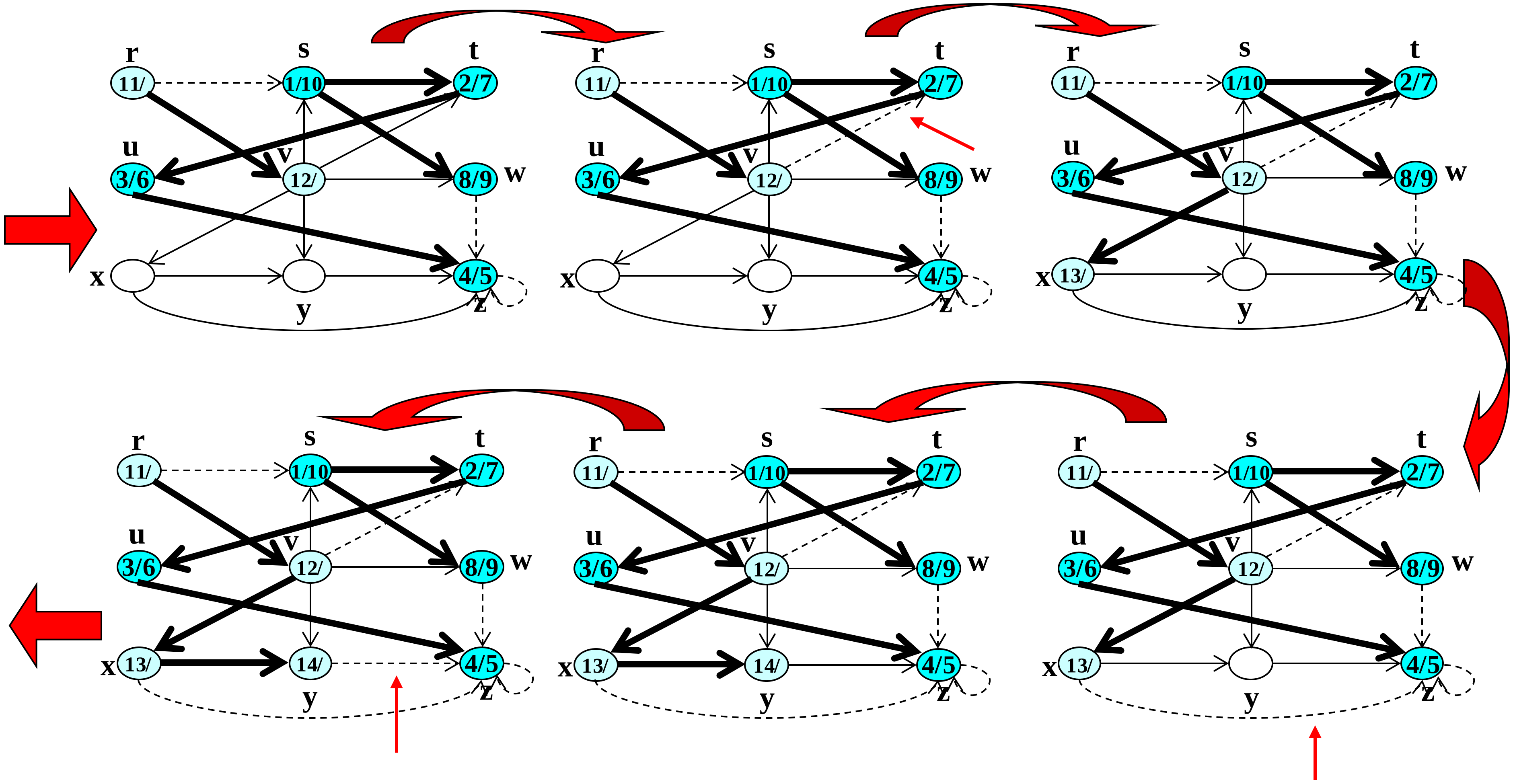
An example to DFS



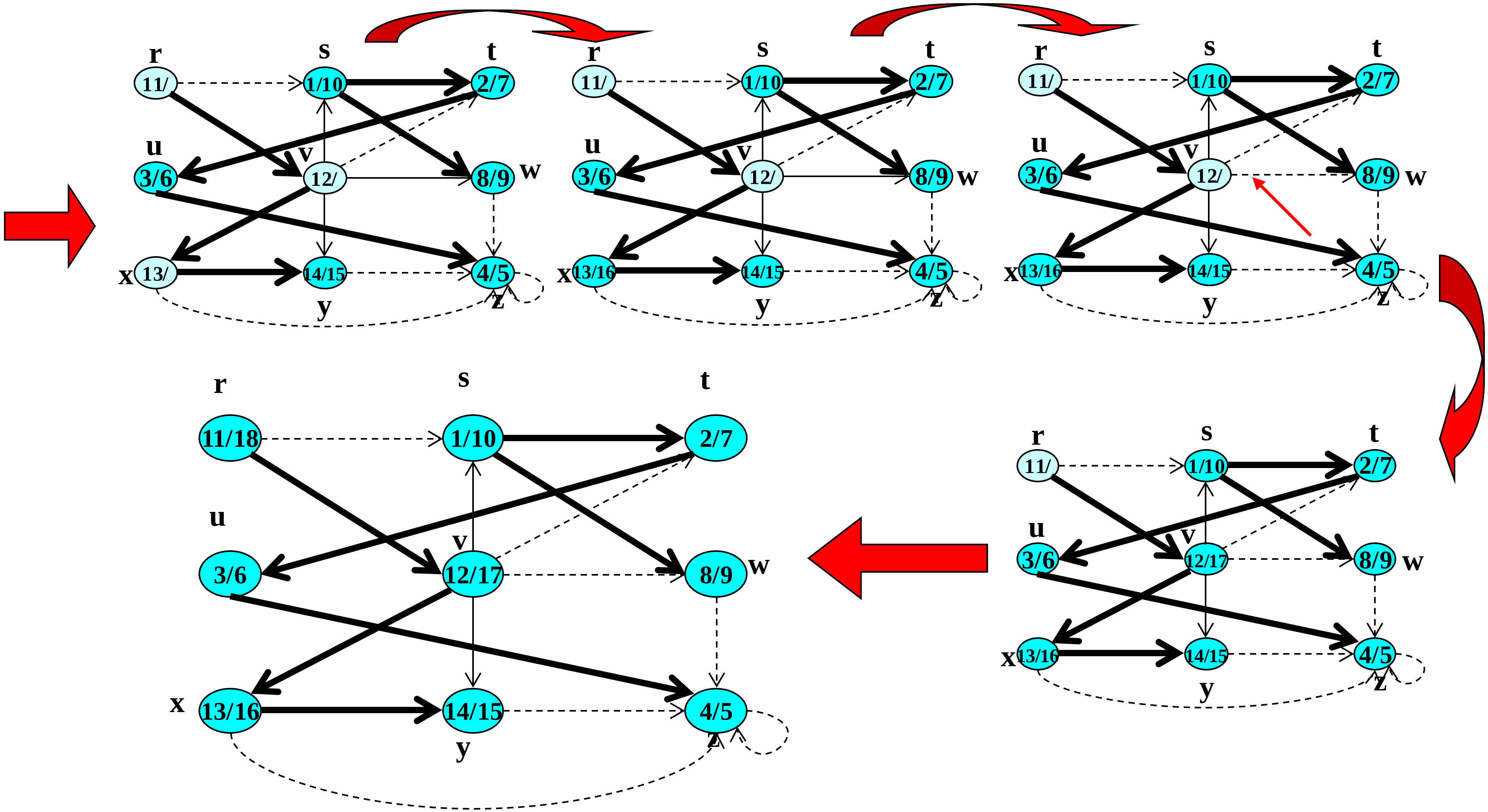
Example cont'd...



Example cont'd...



End of Example



Single-Source Shortest Paths (SSSP)

- SSSP Problem:
- Given a weighted digraph *G (V,E)*, we need to efficiently find the shortest path

  $$p^*=(u_i, u_{i+1},\ldots, u_j,\ldots, u_{k-1},u_k)$$

  between two vertices u_i and u_k.
- The shortest path p^* is the path with the minimum weight among all paths $p_l=(u_i,\ldots,u_k)$, or

$$w(p^*) = \min_l \left[w(p_l) \right]$$

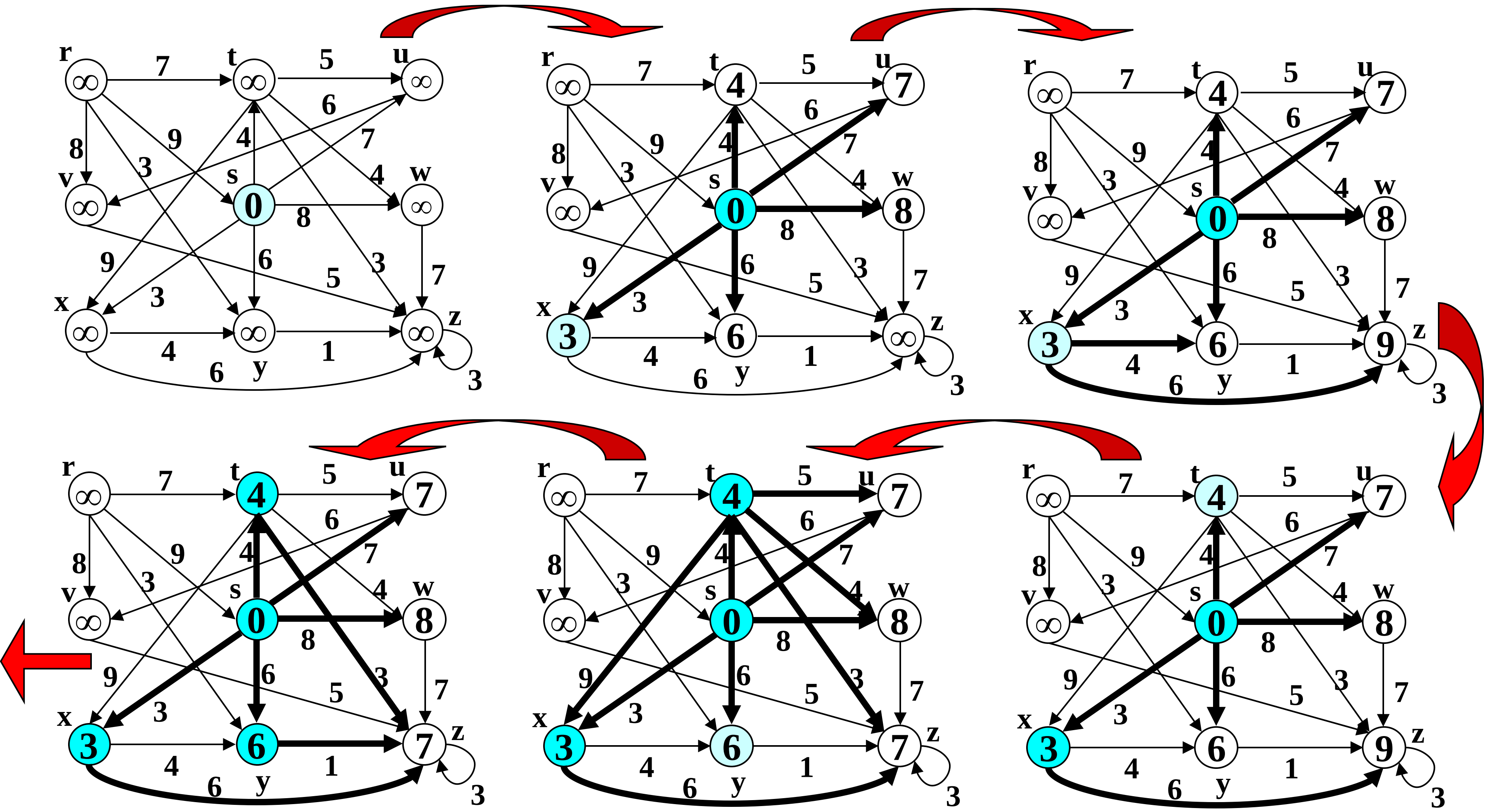
Dijkstra's Algorithm

- Dijkstra's algorithm solves the SSSP problem on a weighted digraph $G=(V,E)$ *assuming no negative weights* exist in G.
- Input parameters for Dijkstra's algorithm
  - the graph G,
  - the weights w,
  - a source vertex s.
- It uses
  - a set V_F holding vertices with final shortest paths from the source vertex s.
  - from[u] and dist[u] for each vertex $u \in V$ as in BFS.
  - A min-heap Q

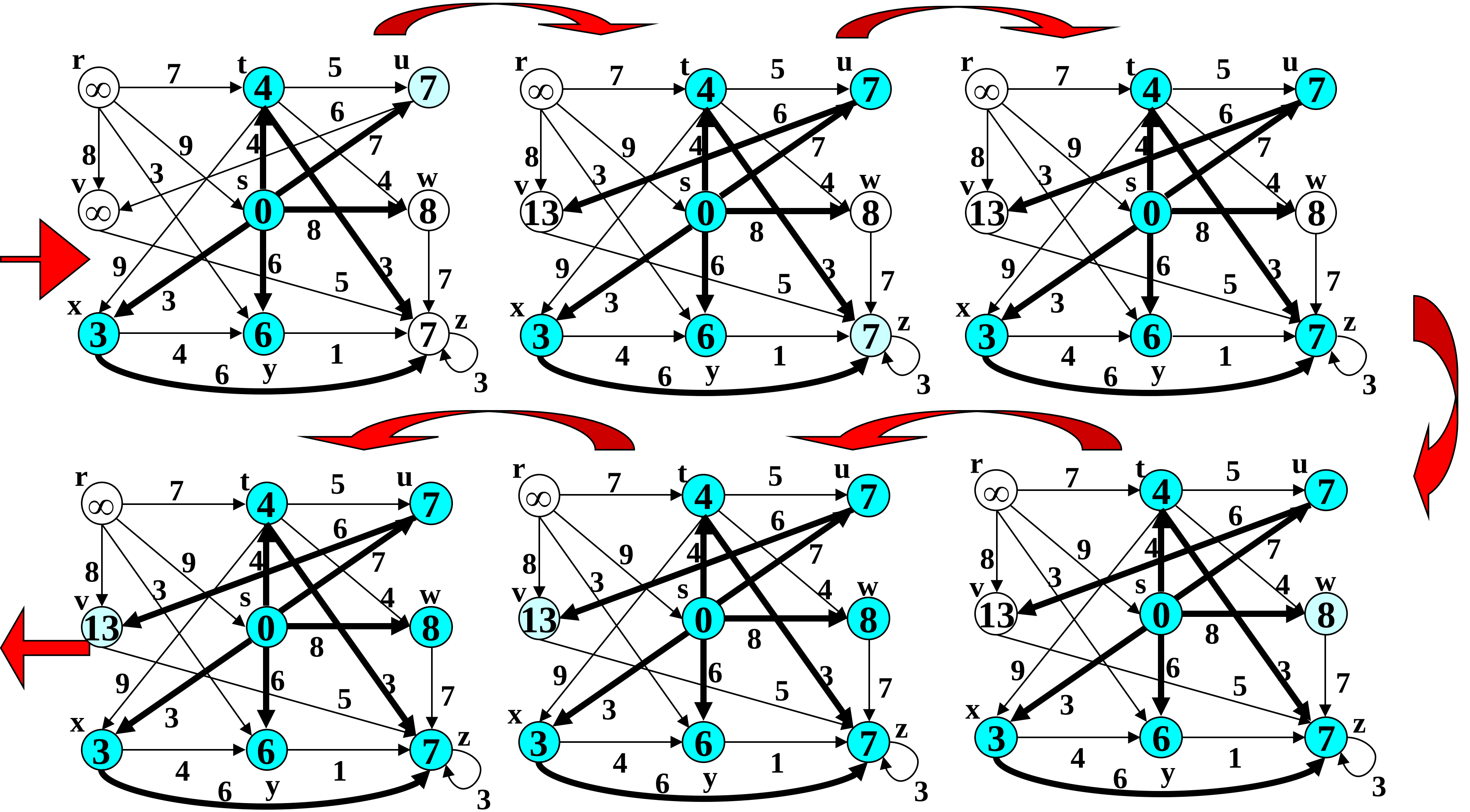
Dijkstra’s Algorithm

```
Dijkstra(Graph G,
         Weights w, Vertex s)
{
    for each vertex u∈V[G] {
      dist [u]=∞;
      from[u]=NULL;
    }
    dist [s]=0;
    V_F = ø;
    Q  = all vertices u ∊ V;
    while (!IsEmpty(Q)) {
      u=deletemin(Q);
      add u to V_F ;
      for each vertex v ∊ Adj(u)
        if (dist[v]>dist[u]+w(u,v)){
           dist[v]=dist[u]+w(u,v));
           from[v]=u;
        }
    } // end of while
} //end of function
```

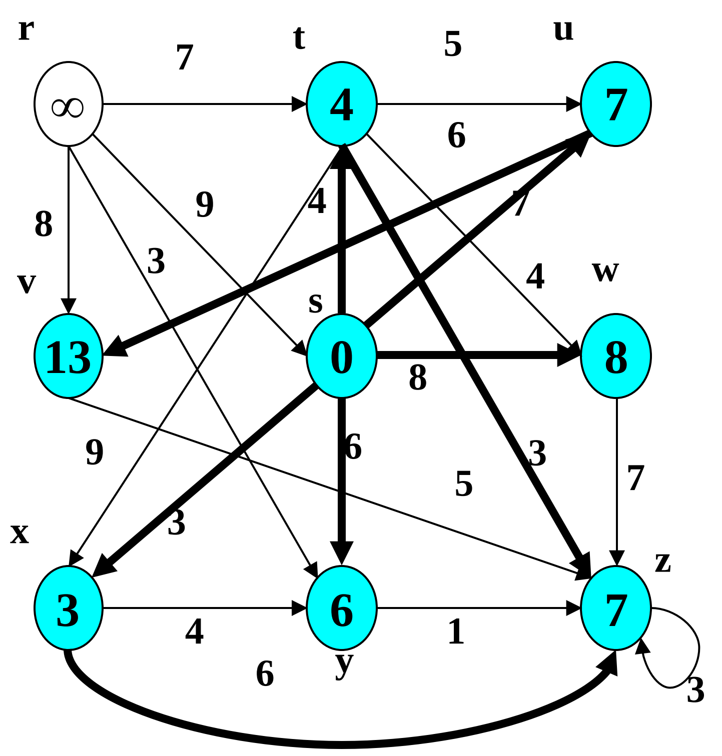
Dijkstra’s Algorithm – An Example



Dijkstra’s Algorithm – An Example



Resulting Shortest Paths



Note that *r* is not reachable from *s*!

Minimum Spanning Trees (MSTs)

- Problem:
- Given a *connected weighted undirected graph* *G=(V,E)*, *find an acyclic subset* $S \subseteq E$, such that *S connects all vertices in G and the sum of the weights of the edges in S are minimum*.
- The solution to the problem is provided by a *minimum spanning tree*.

Minimum Spanning Trees (MSTs)

- MST is
  - a *tree since it connects all vertices by an acyclic subset* of $S \subseteq E$,
  - *spanning since it spans the graph* (connects all its vertices)
  - *minimum since its weights are minimized*.

Prim's Algorithm

- Prim's algorithm operates *similar to Dijkstra's* algorithm to find shortest paths.
- Prim's algorithm *proceeds always with a single tree*.
- It starts with an arbitrary vertex *t*.
- It progressively connects an isolated vertex to the existing tree by adding the edge with the minimum possible weight to the tree.

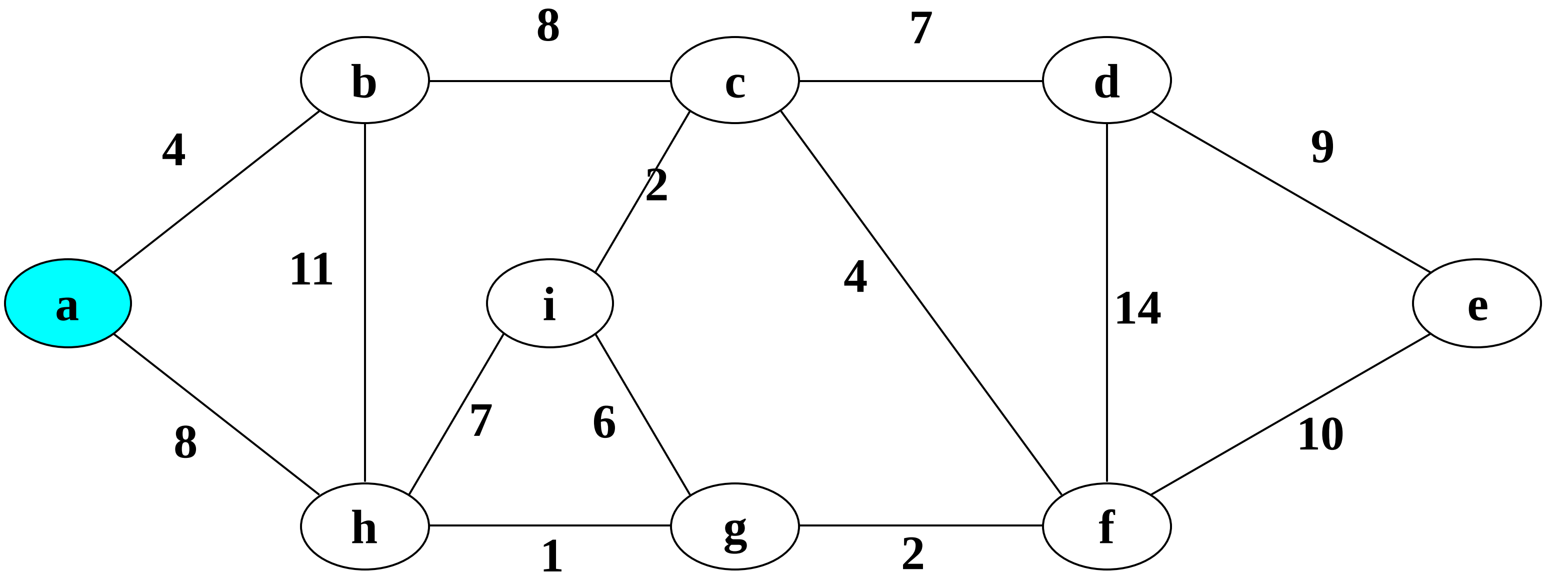
Prim's Algorithm

```
Prim(Graph G,
        Weights w, Vertex t)
{
     for each vertex u∈V[G] {
       dist [u]=∞;
       from[u]=NULL;
     }
     dist [t]=0;
     V_F = ø;
     Q  = all vertices u ∈ V;
```

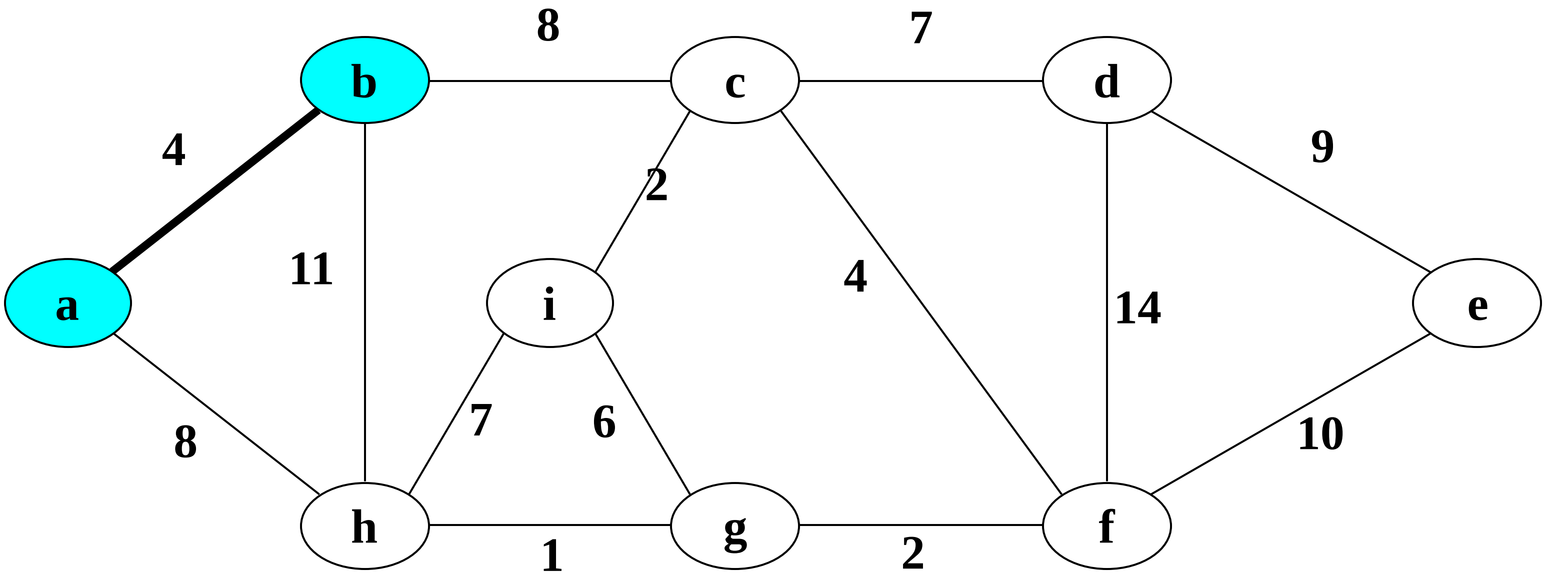
```
while (!IsEmpty(Q)) {
     u=deletemin(Q);       O(VlgV)
     add u to V_F ;
     for each vertex v ∈ Adj(u) O(E)
        if (v∈Q and w(u,v)<dist[v]){
             dist[v]=w(u,v);  O(lgV)
             from[v]=u;
        }
  } // end of while
}  //end of function
```

Running Time: O(V lgV + E lgV)=O(E lgV)

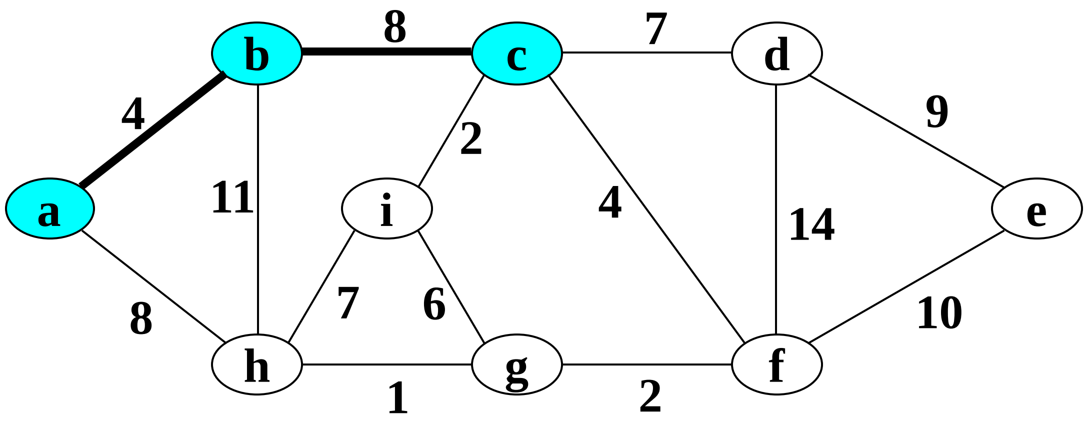
Prim's Algorithm - Example

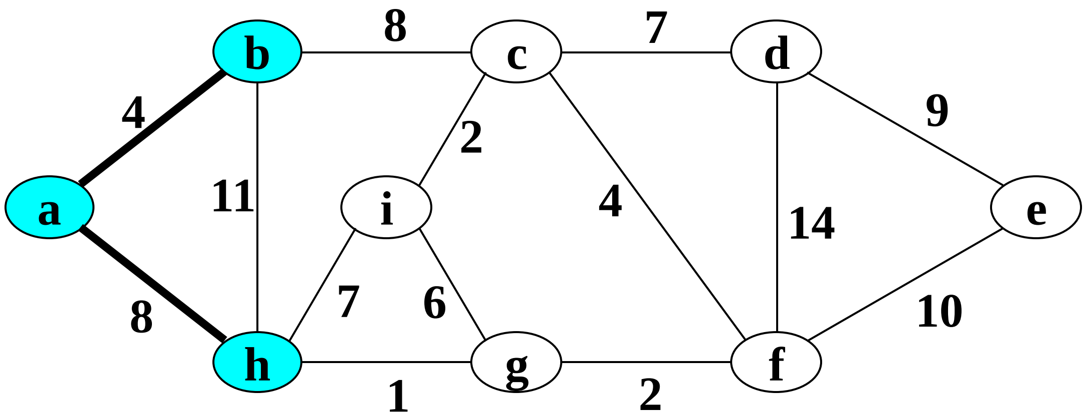


Prim's Algorithm - Example

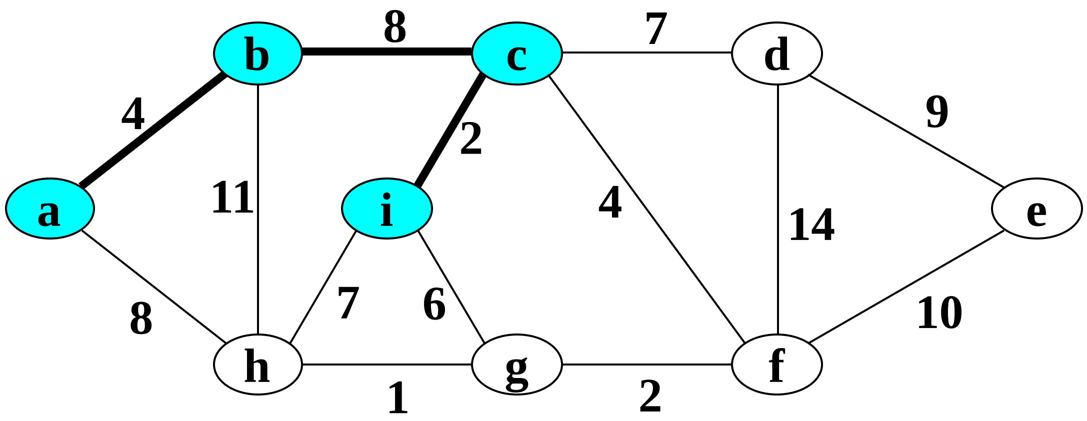


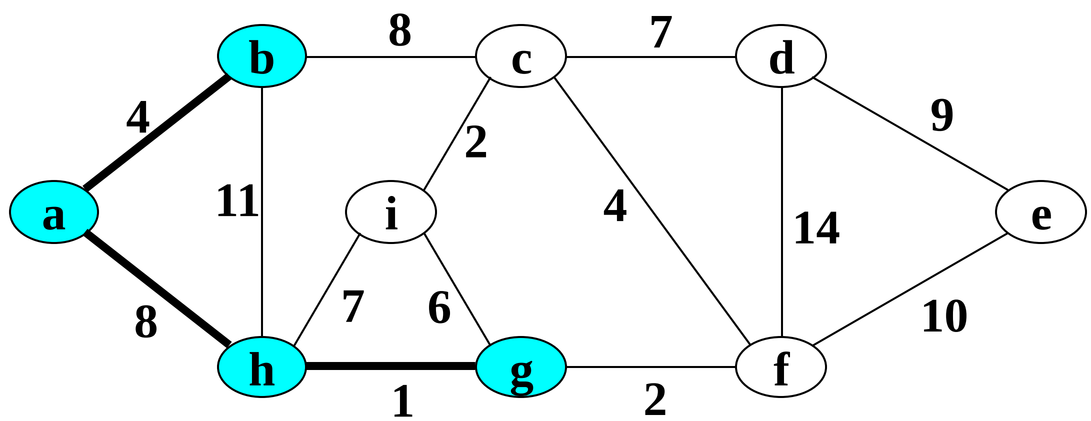
Prim's Algorithm - Example



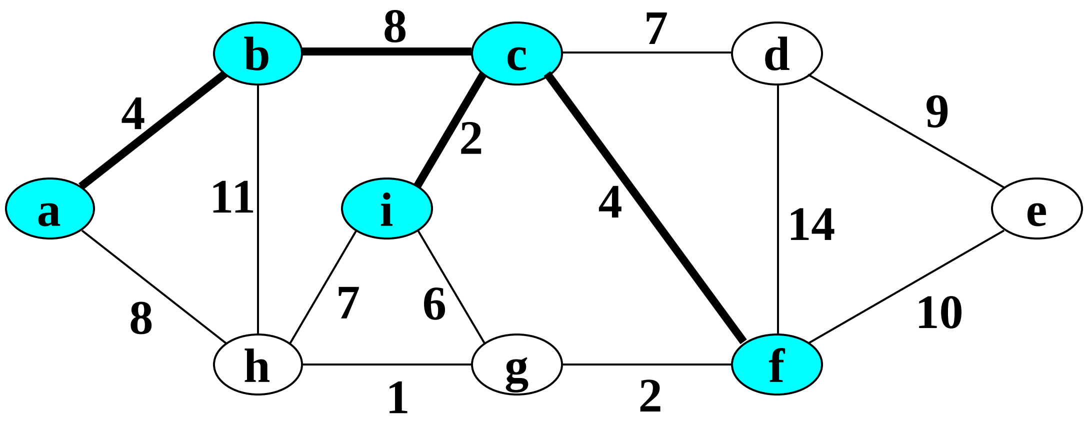


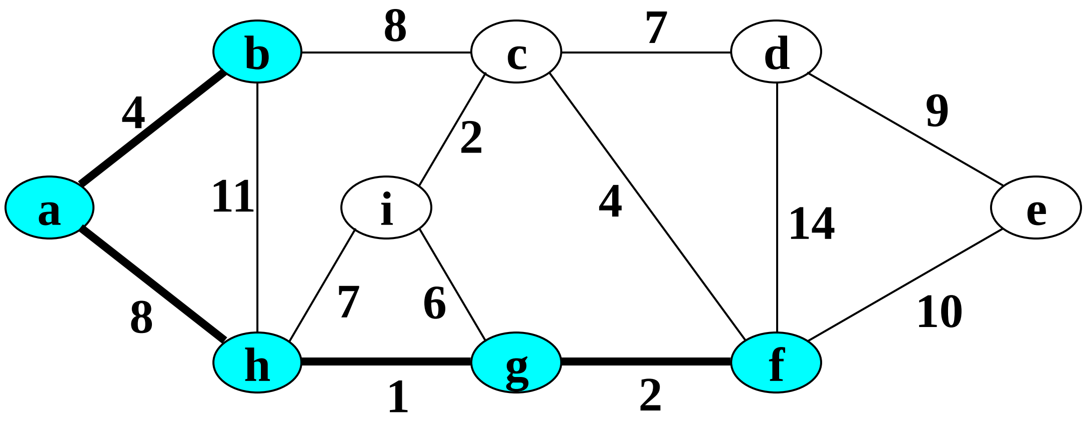
Prim's Algorithm - Example



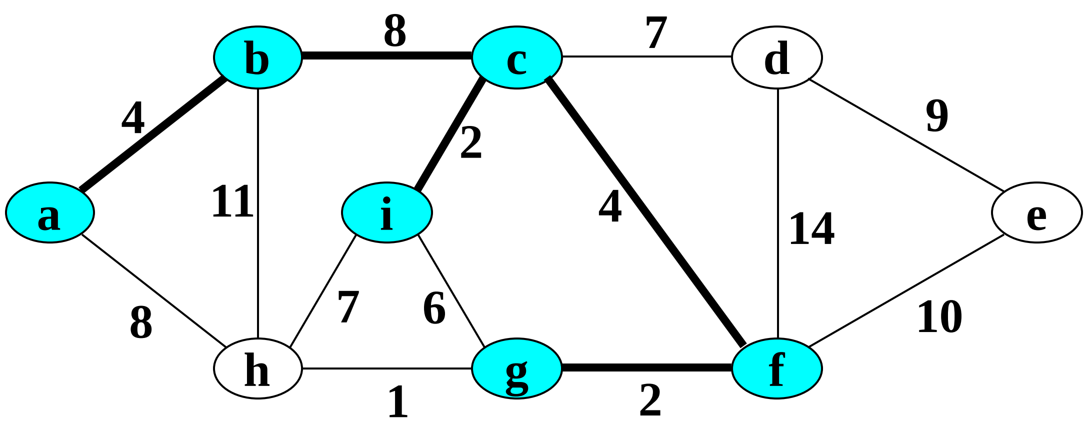


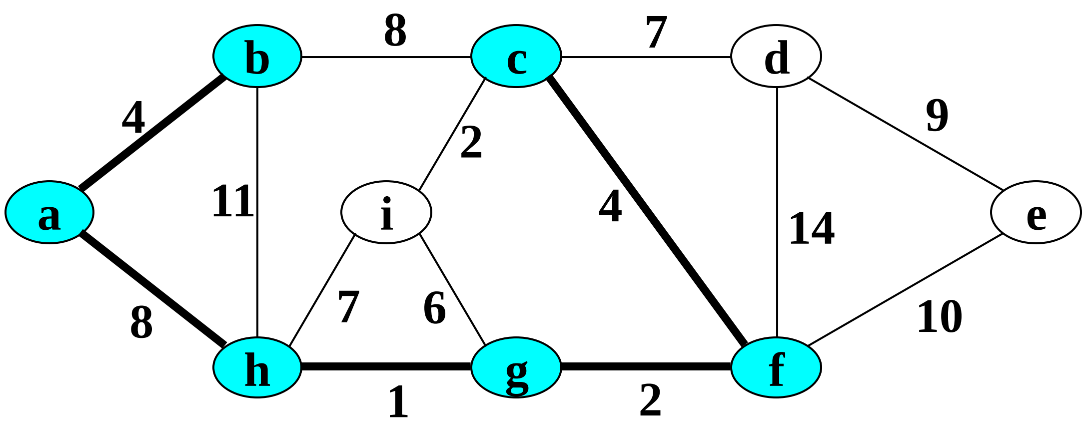
Prim's Algorithm - Example



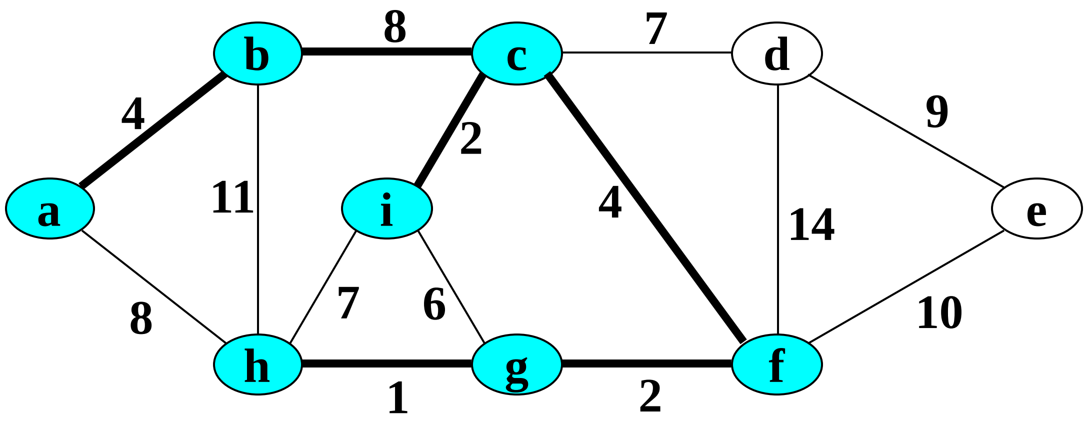


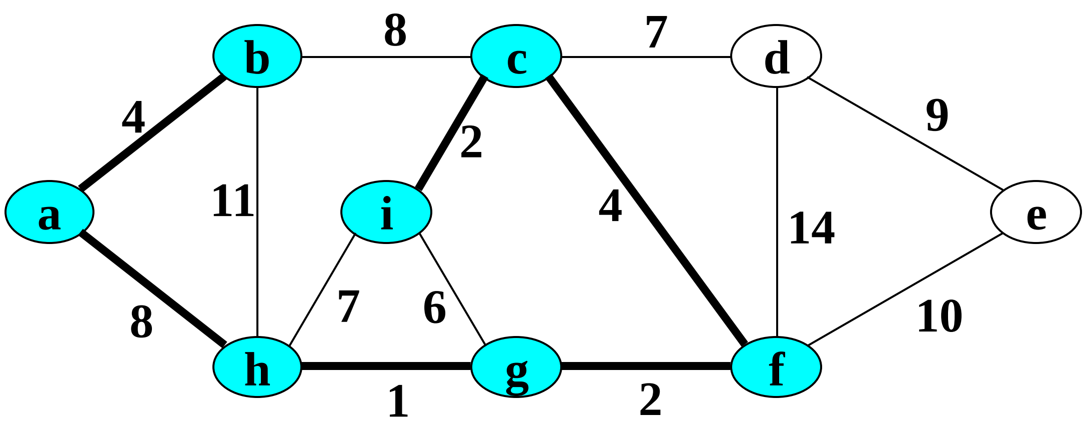
Prim’s Algorithm - Example



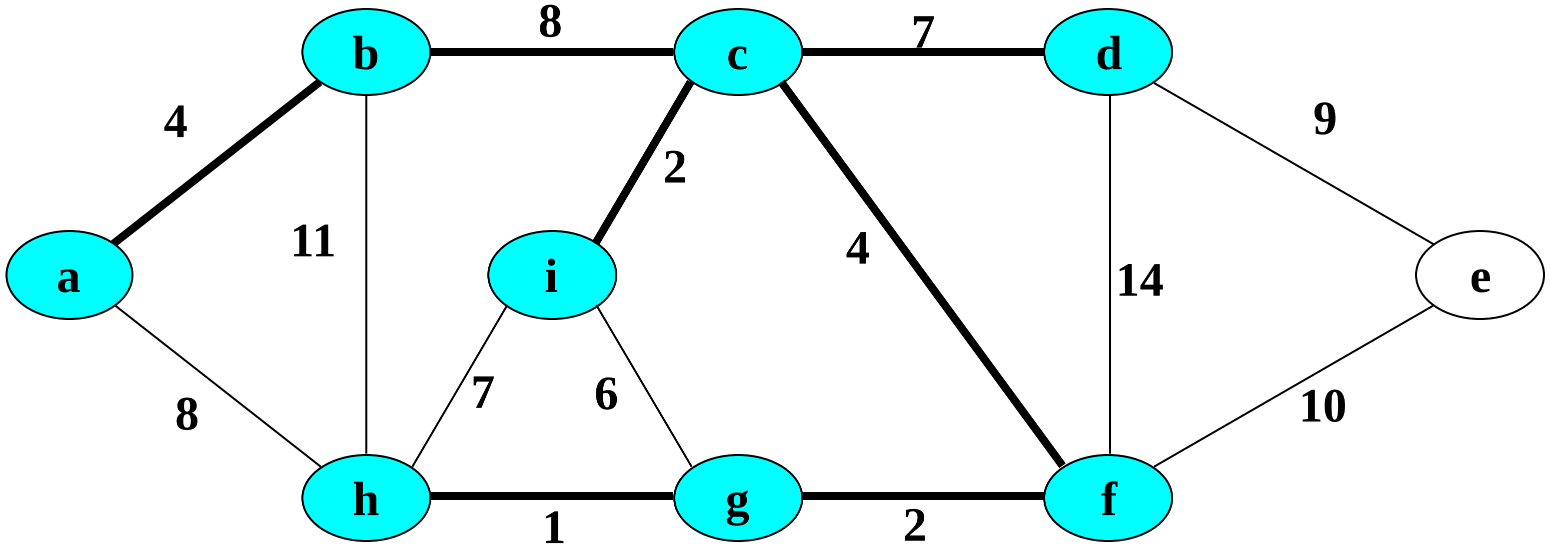


Prim’s Algorithm - Example

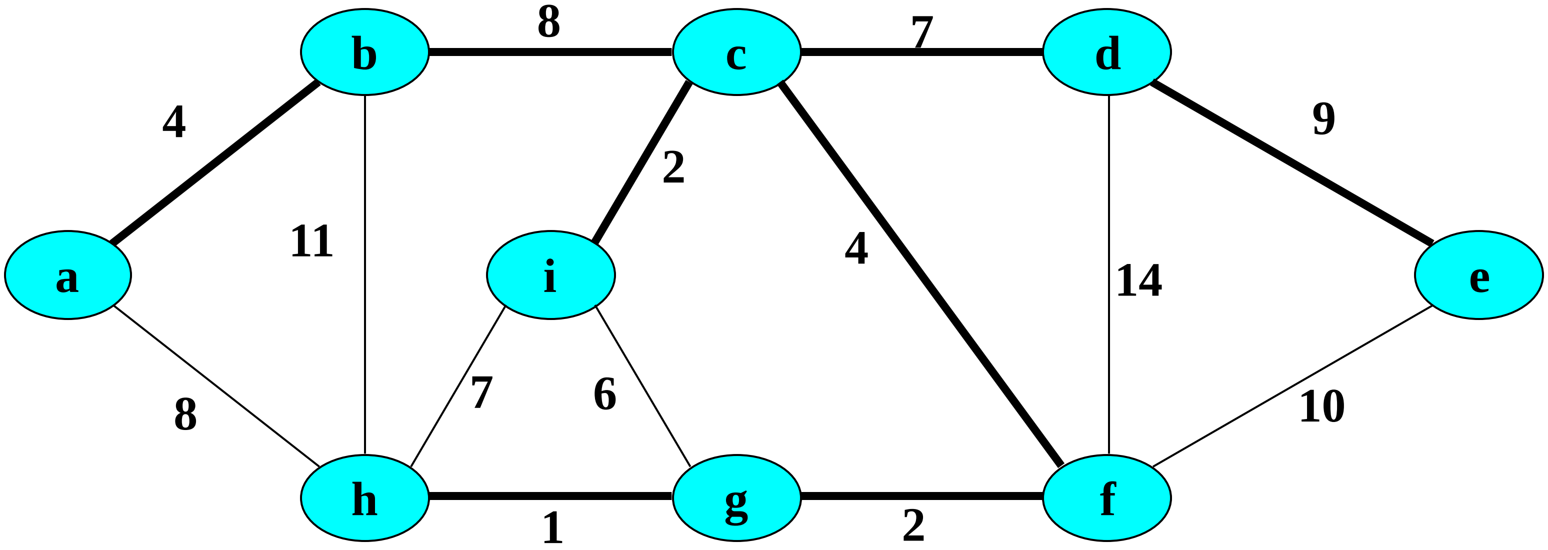




Prim’s Algorithm - Example



Prim's Algorithm - Example



Kruskal's Algorithm

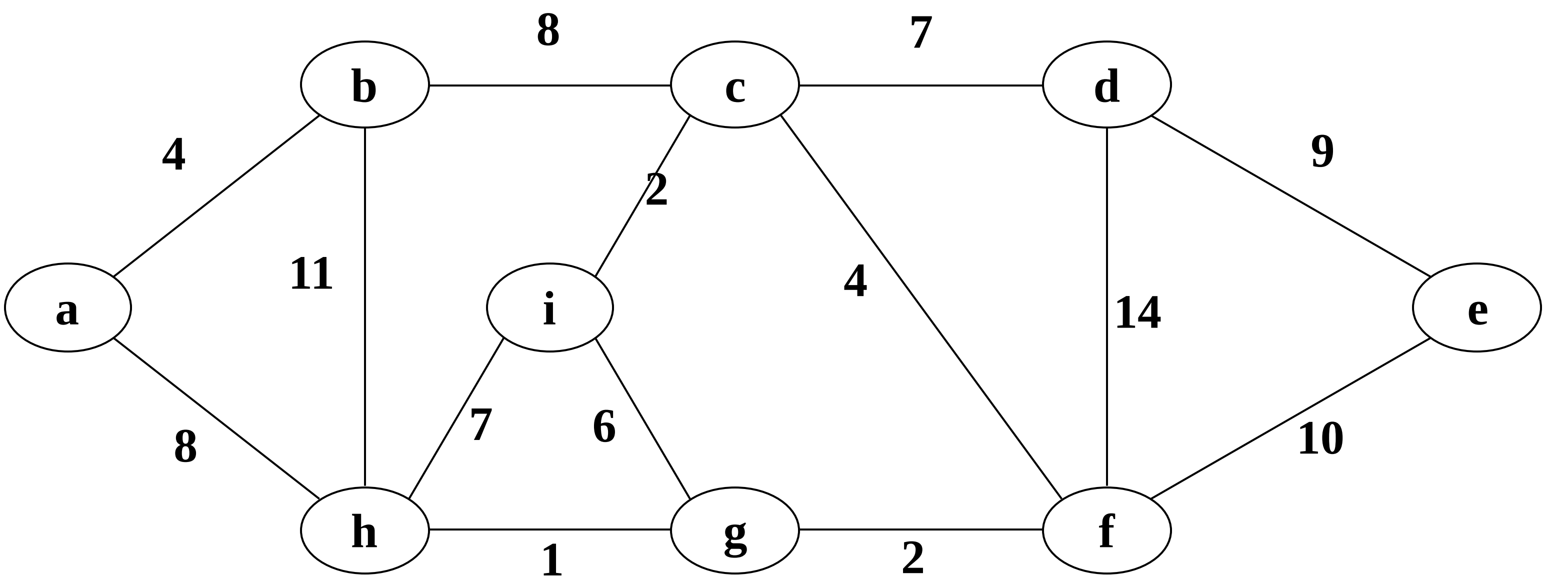
- Kruskal's Algorithm is *another greedy algorithm*.
- It is about *finding the least weight and connecting with that two trees in the forest*.
- *Initially*, there exists *a forest of many single-node trees*.

Kruskal's Algorithm

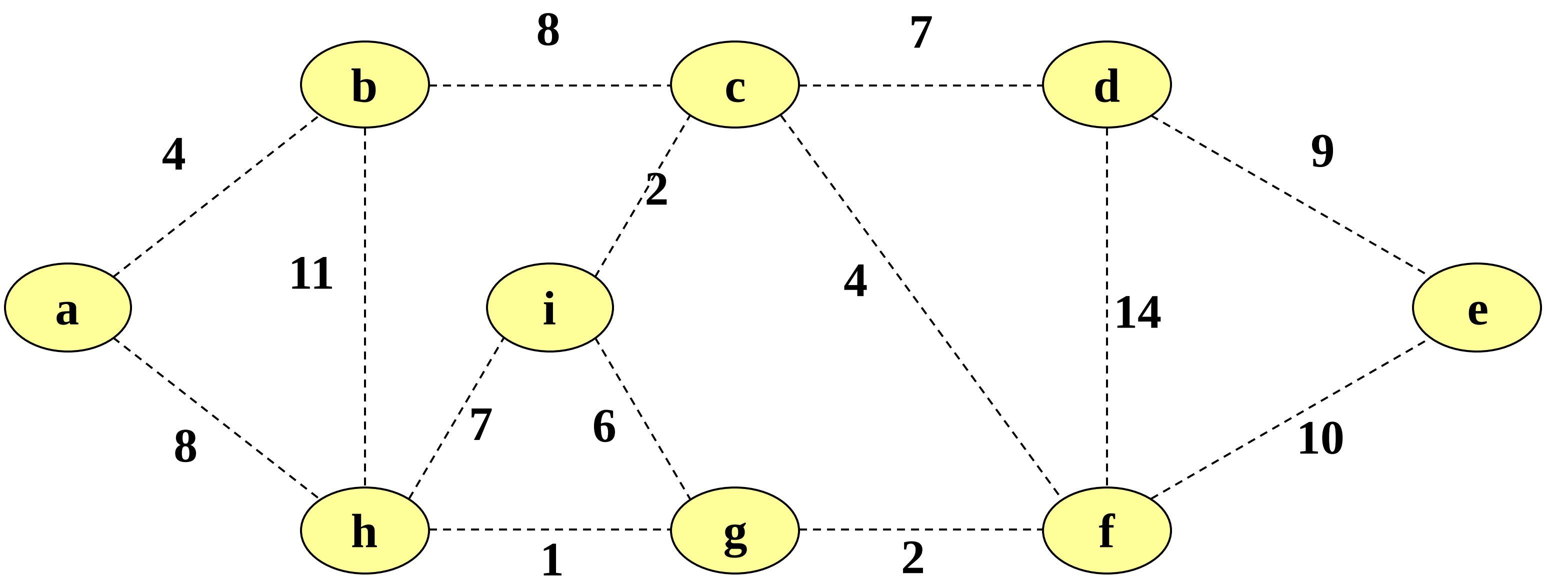
```
Kruskal(Graph G,
      Weights w)
{
    for each vertex u∈V[G] {
           make each vertex to a single-element tree;
    }
    sort edges in ascending order by their weight w;              O(E lgE)
    for each edge (u,v) ∈E                                         O(E)
          if (u and v are in two different trees) {               lgE=O(lgV)
            add (u,v) to the MST;                                  since |E|<|V|^2
            combine both trees;
          }
    dist [u]=0;
    return;
```

Running Time: $O(E \lg E + E) = O(E \lg E) = O(E \lg V)$

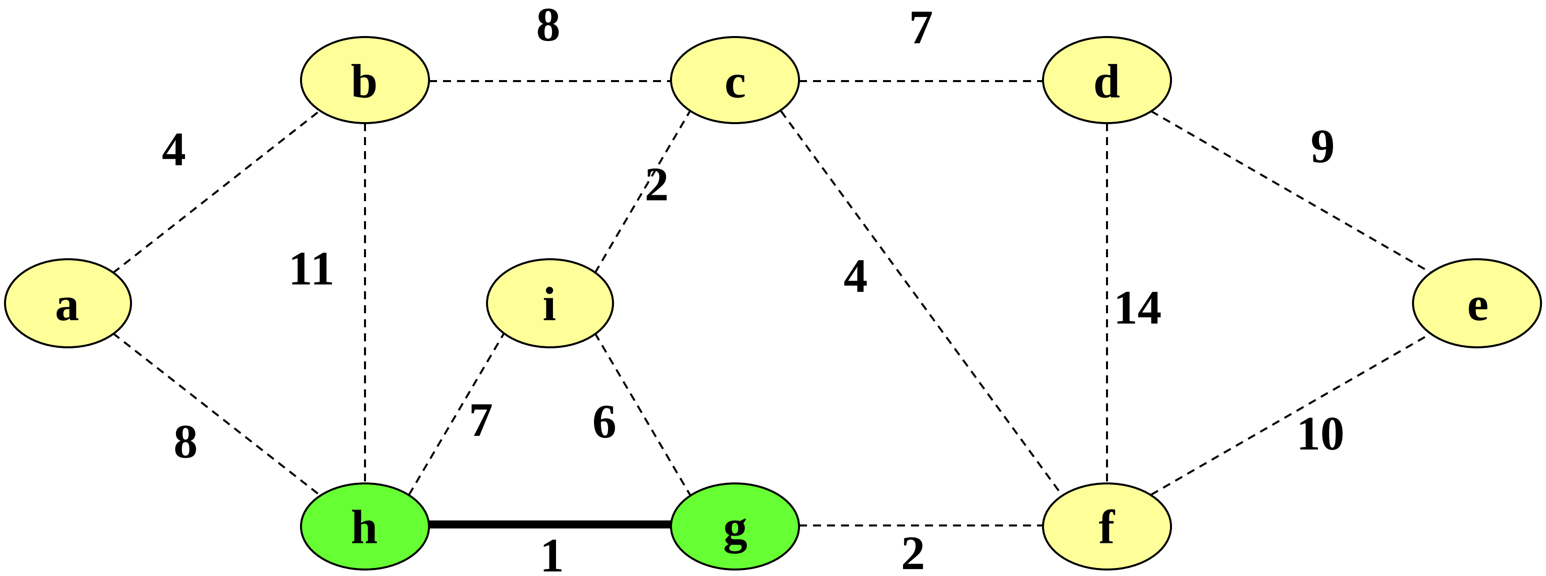
Kruskal's Algorithm - Example



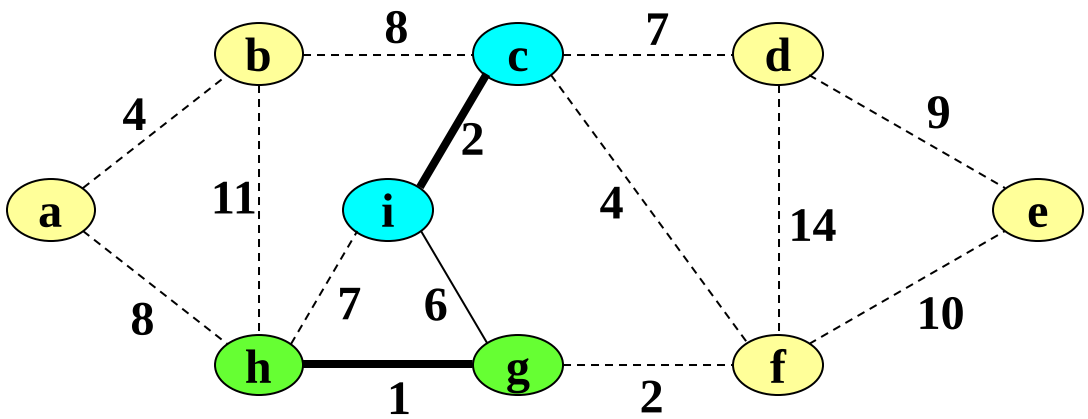
Kruskal's Algorithm - Example

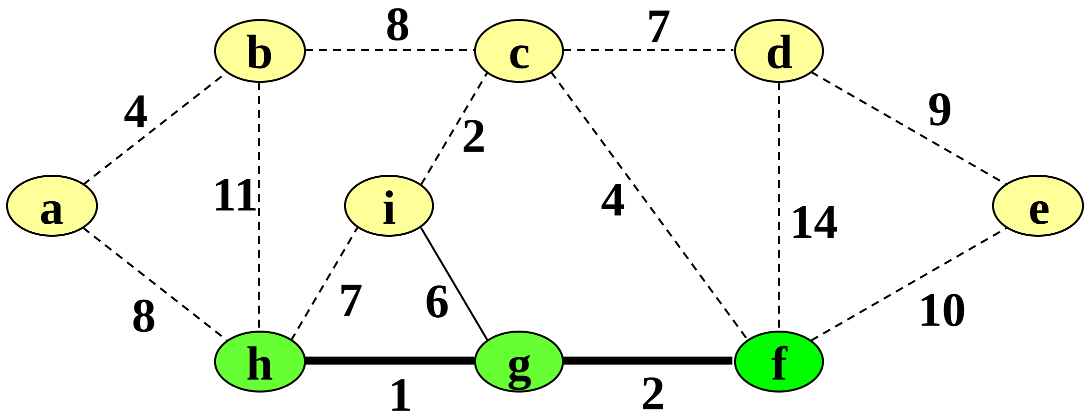


Kruskal's Algorithm - Example

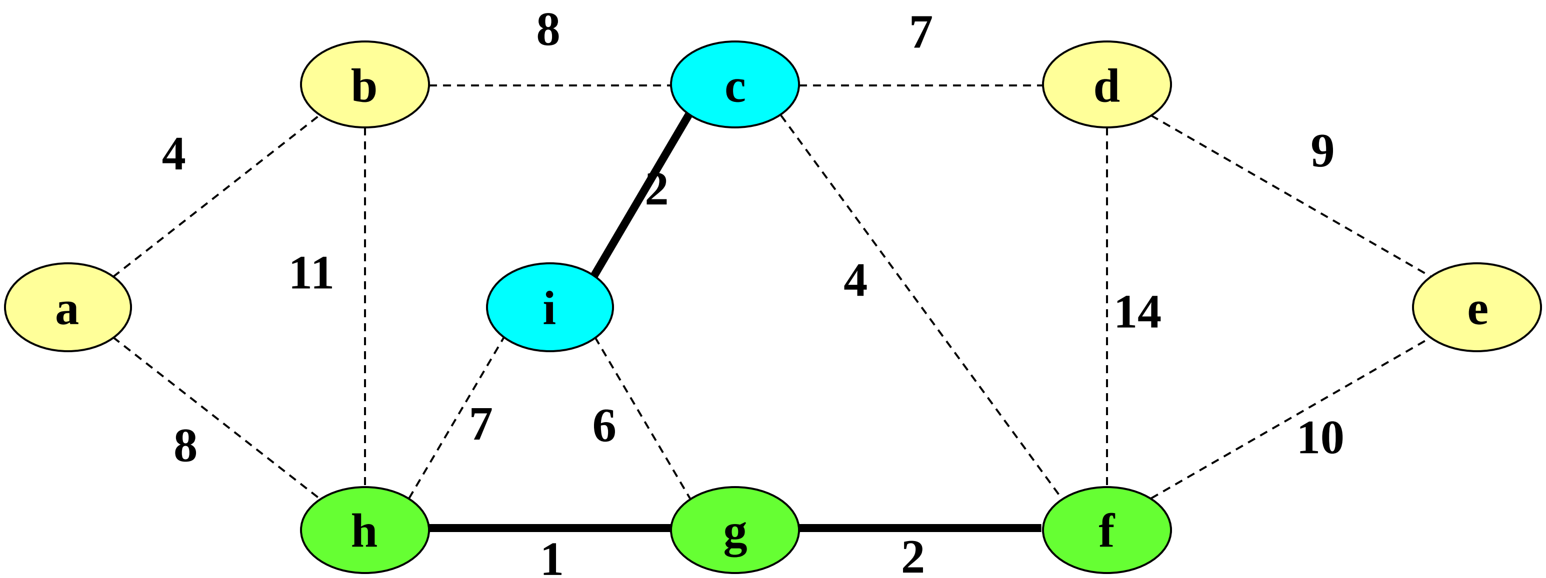


Kruskal's Algorithm – Example
Two Alternatives

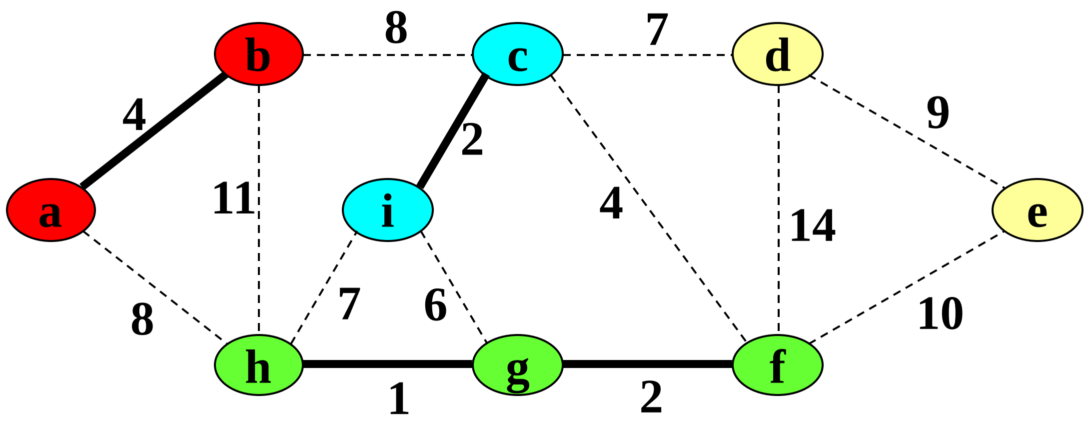


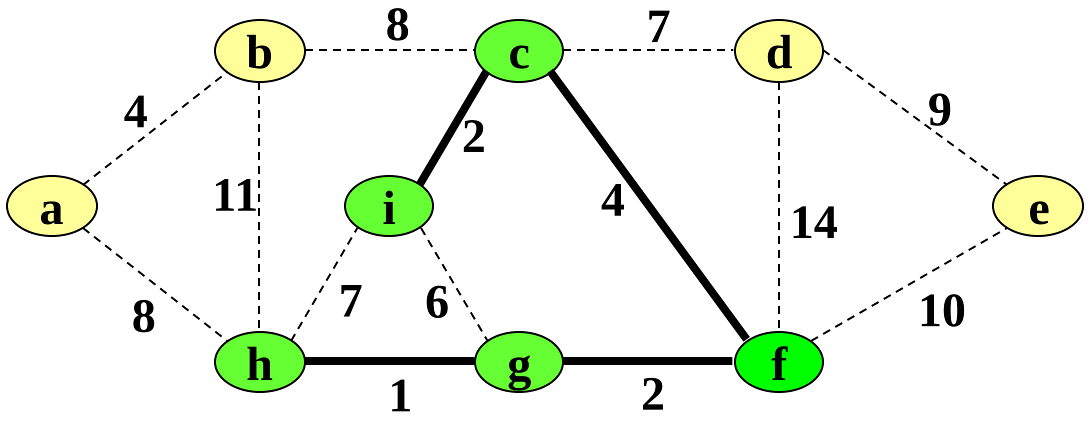


Kruskal's Algorithm - Example

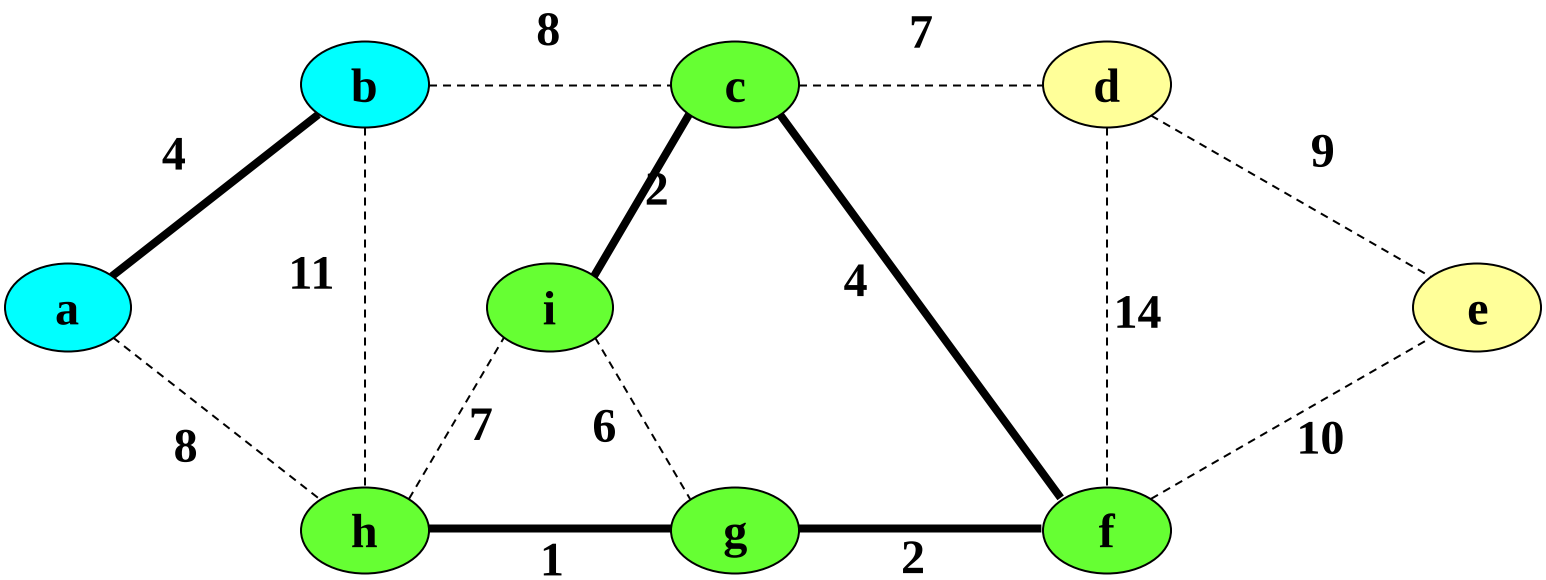


Kruskal's Algorithm – Example Two Alternatives

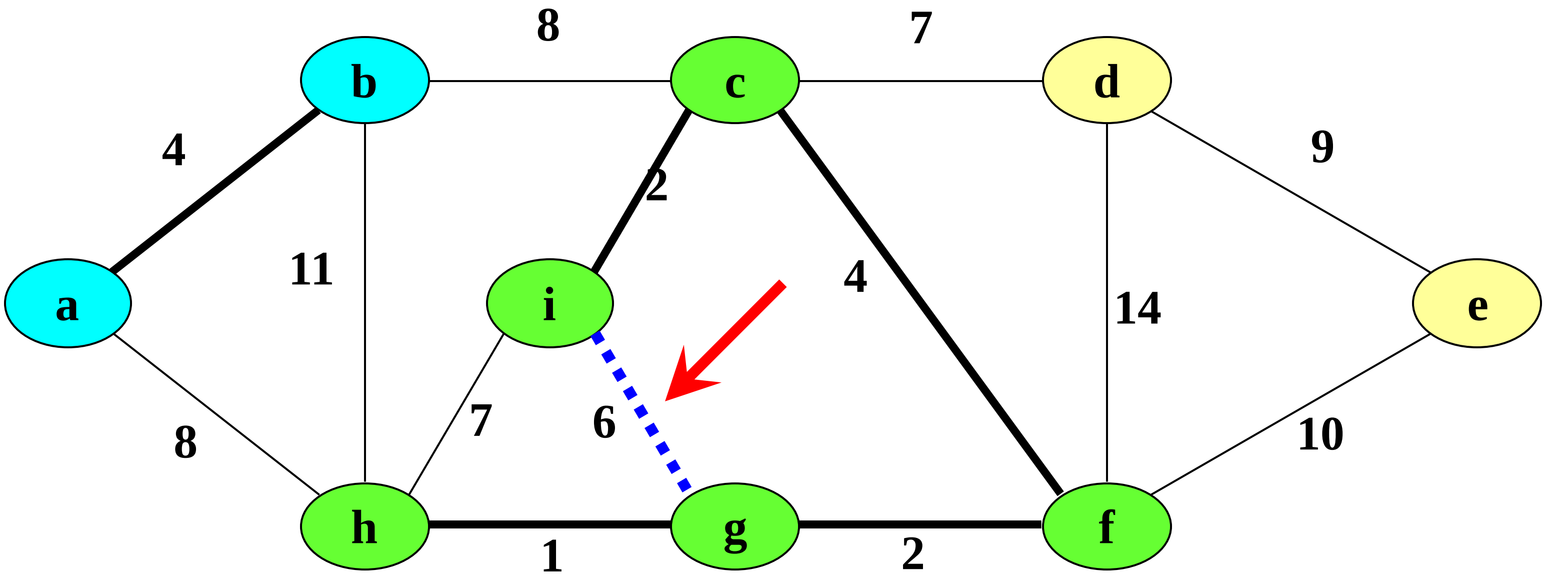




Kruskal’s Algorithm - Example

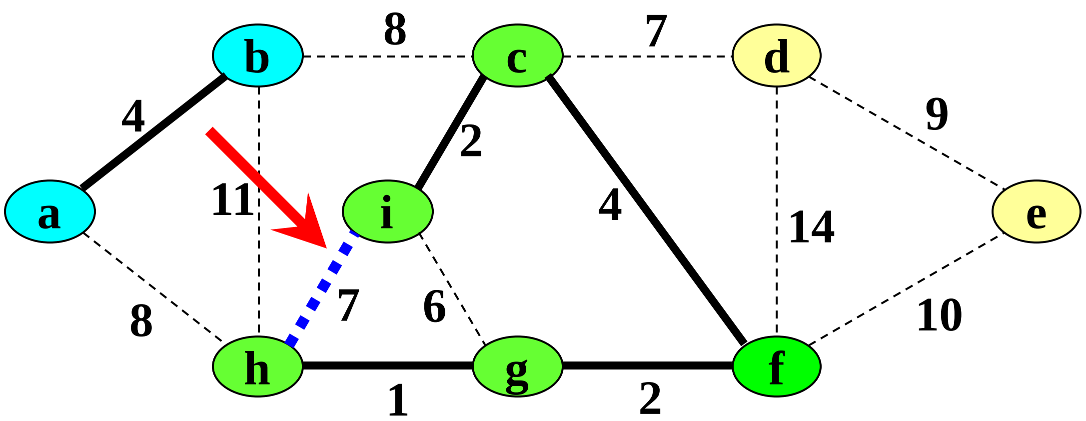


Kruskal's Algorithm - Example

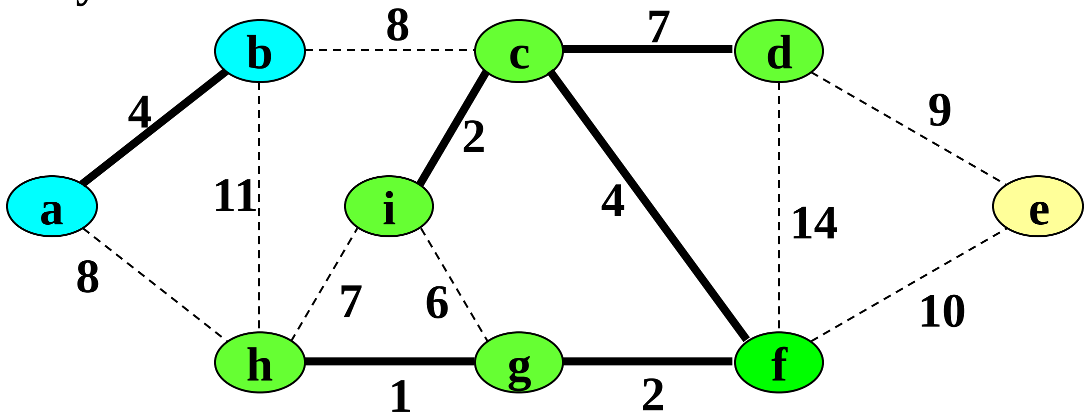


Edge not accepted! It builds a cycle!

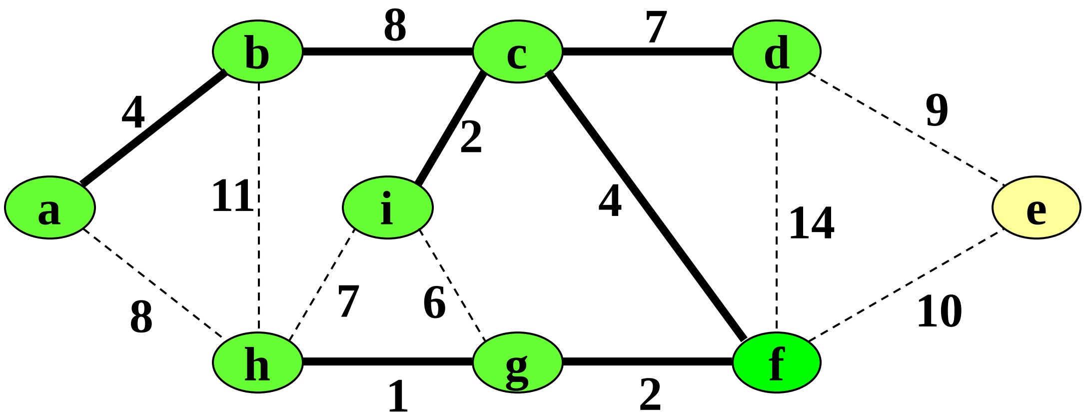
Kruskal's Algorithm – Example
Two Alternatives

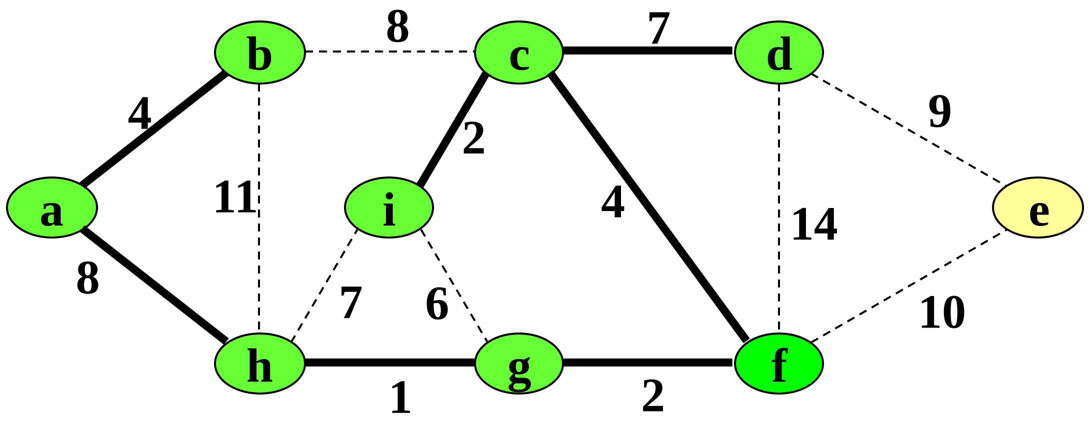


Edge not accepted! It builds a cycle!

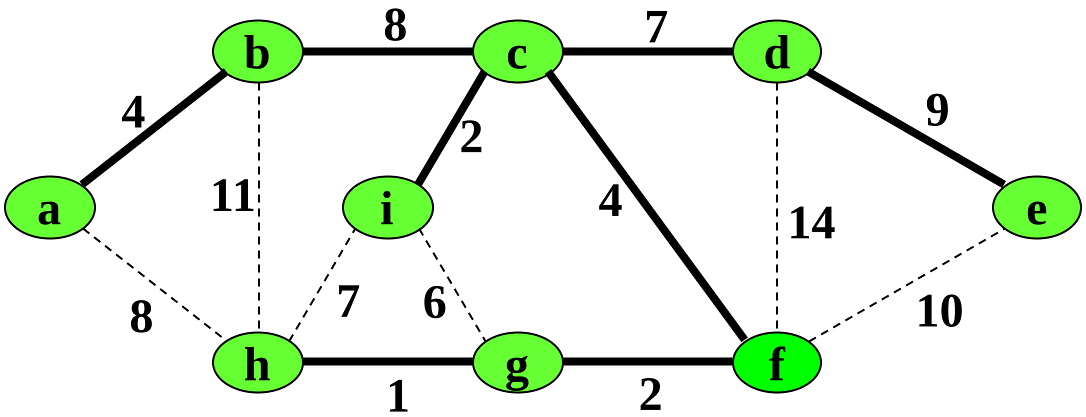


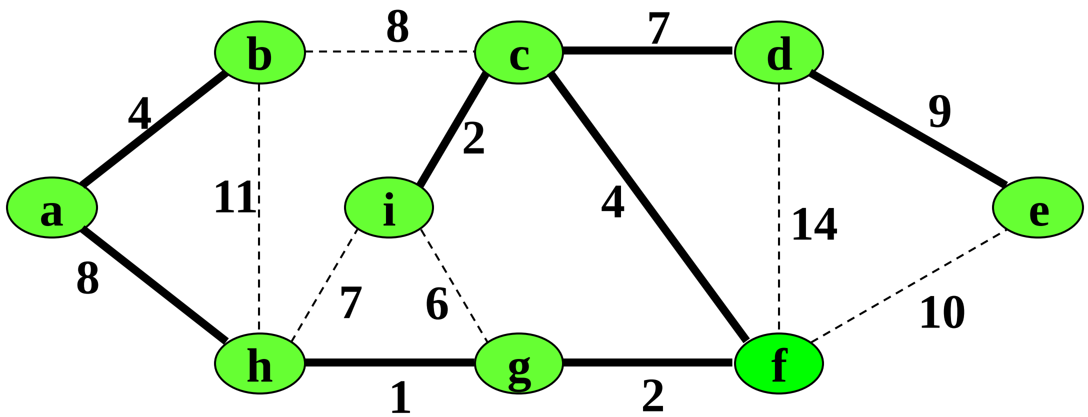
Kruskal’s Algorithm – Example Two Alternatives





Kruskal’s Algorithm – Example
Two Alternatives





References

- [1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, "Introduction to Algorithms," 2nd Edition, 2003, MIT Press
- [2] M.A. Weiss, "Data Structures and Algorithm Analysis in C," 2nd Edition, 1997, Addison Wesley