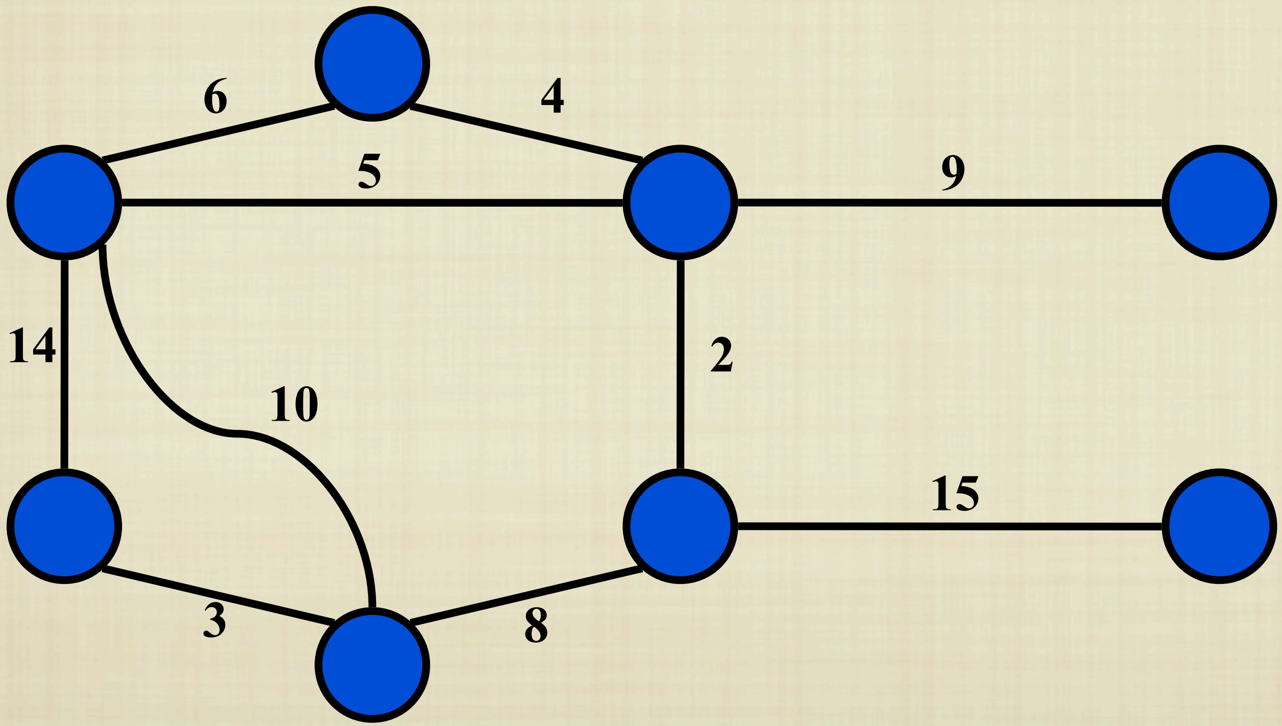
INF01056
Aula 07/08
Graph Algorithms

Prof. João Comba

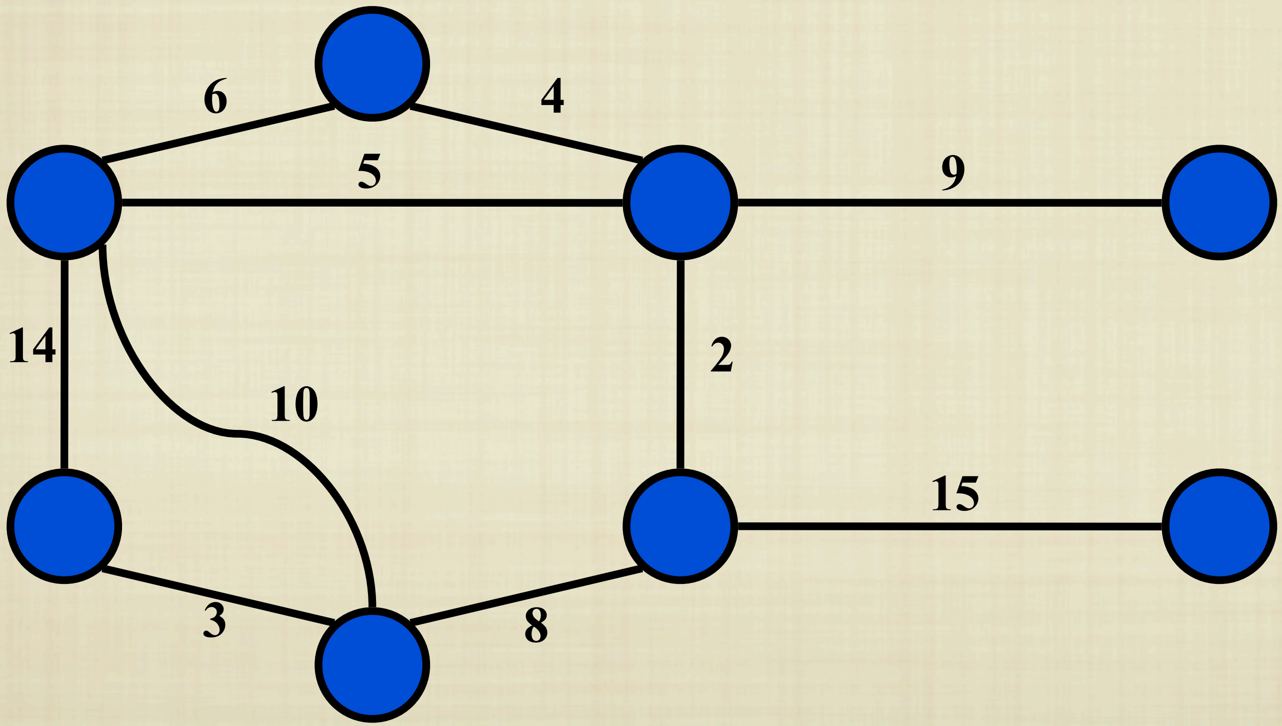
Baseado no Livro Programming Challenges
David Luebke notes - CS332

Weighted Graphs



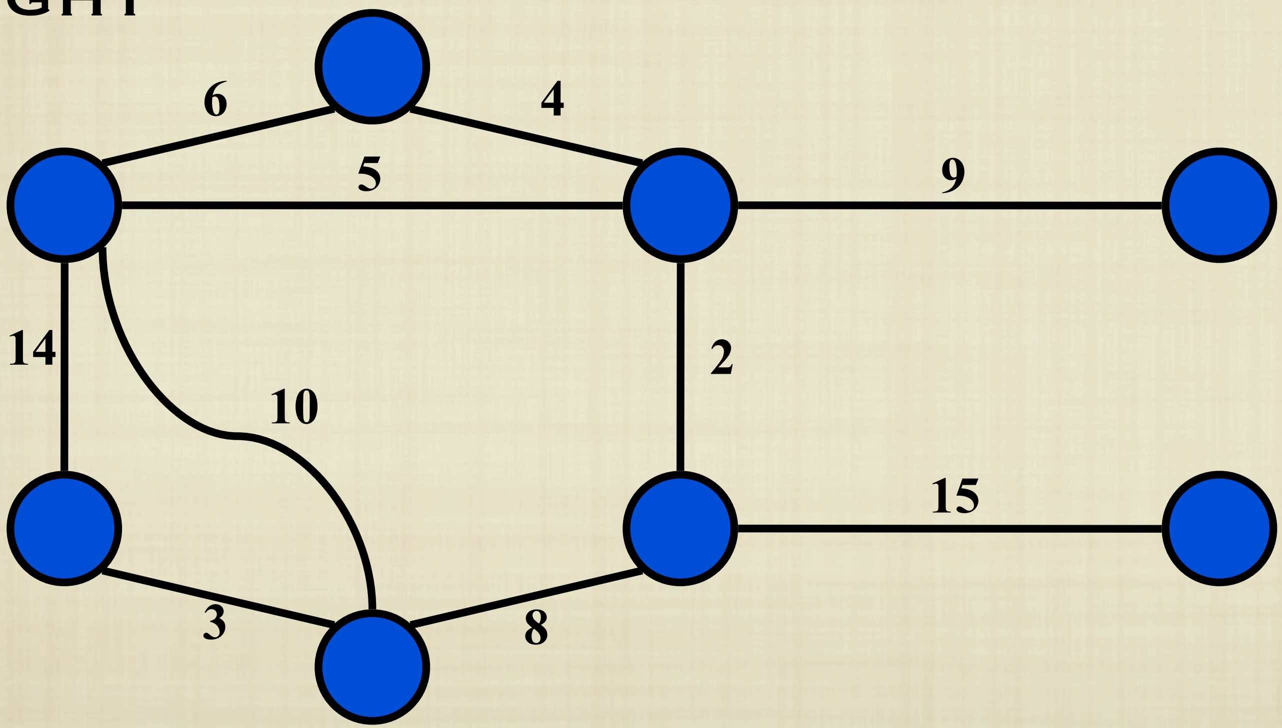
Minimum Spanning Tree

- Problem: given a connected, undirected, weighted graph:



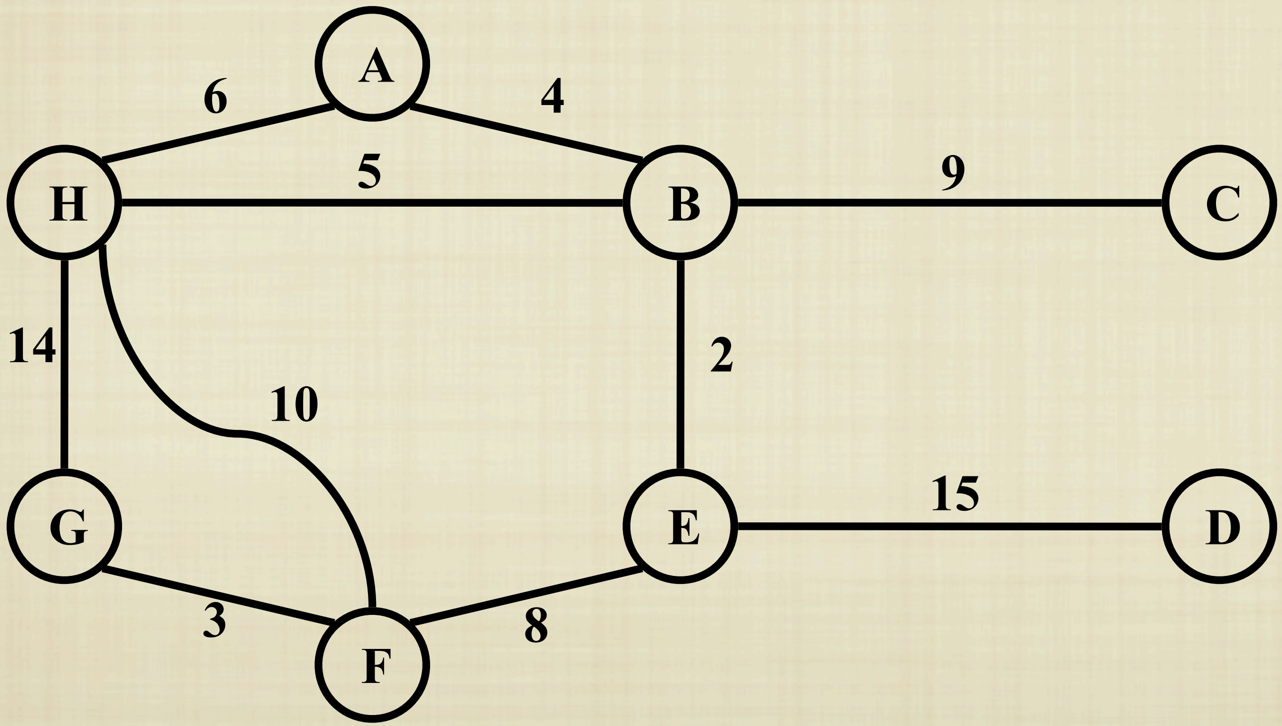
Minimum Spanning Tree

- Problem: given a connected, undirected, weighted graph: find a spanning tree using edges that minimize the total weight



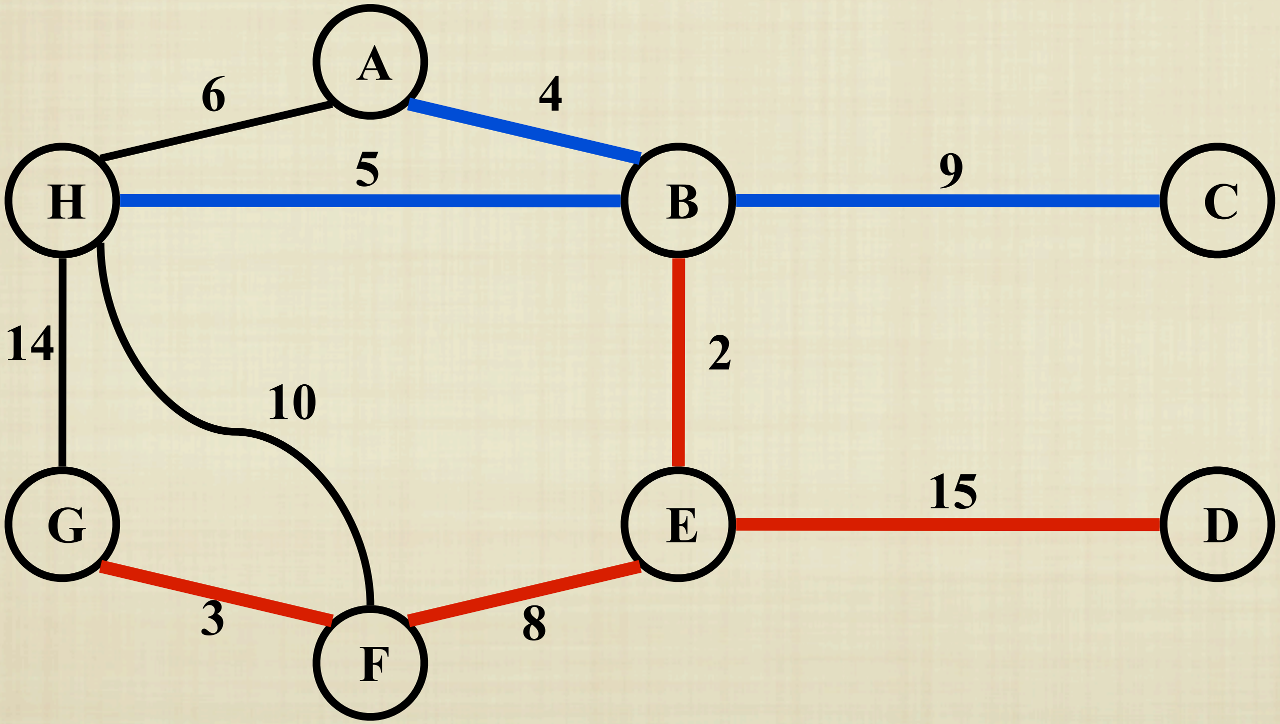
Minimum Spanning Tree

- Which edges form the minimum spanning tree (MST) of the below graph?

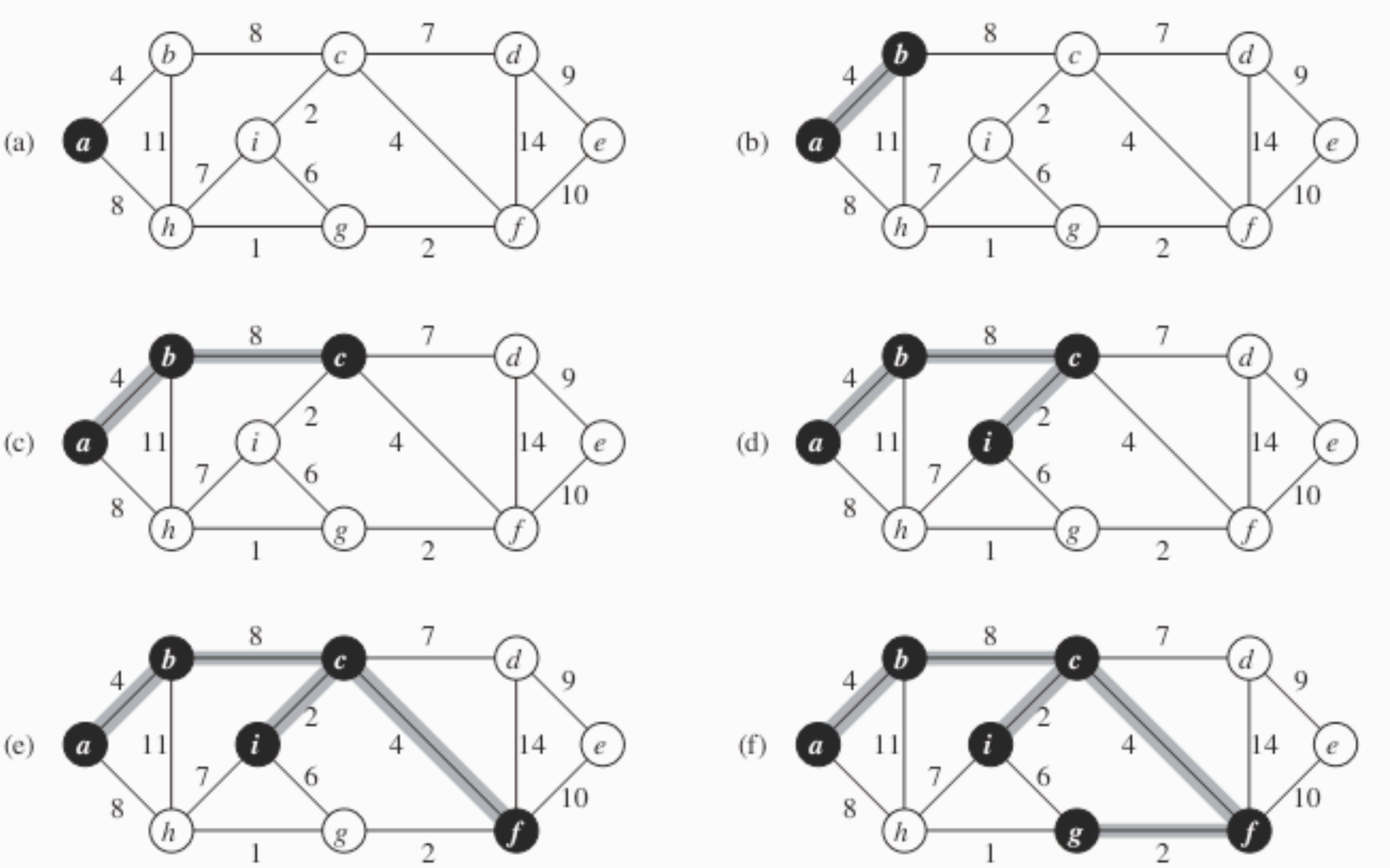


Minimum Spanning Tree

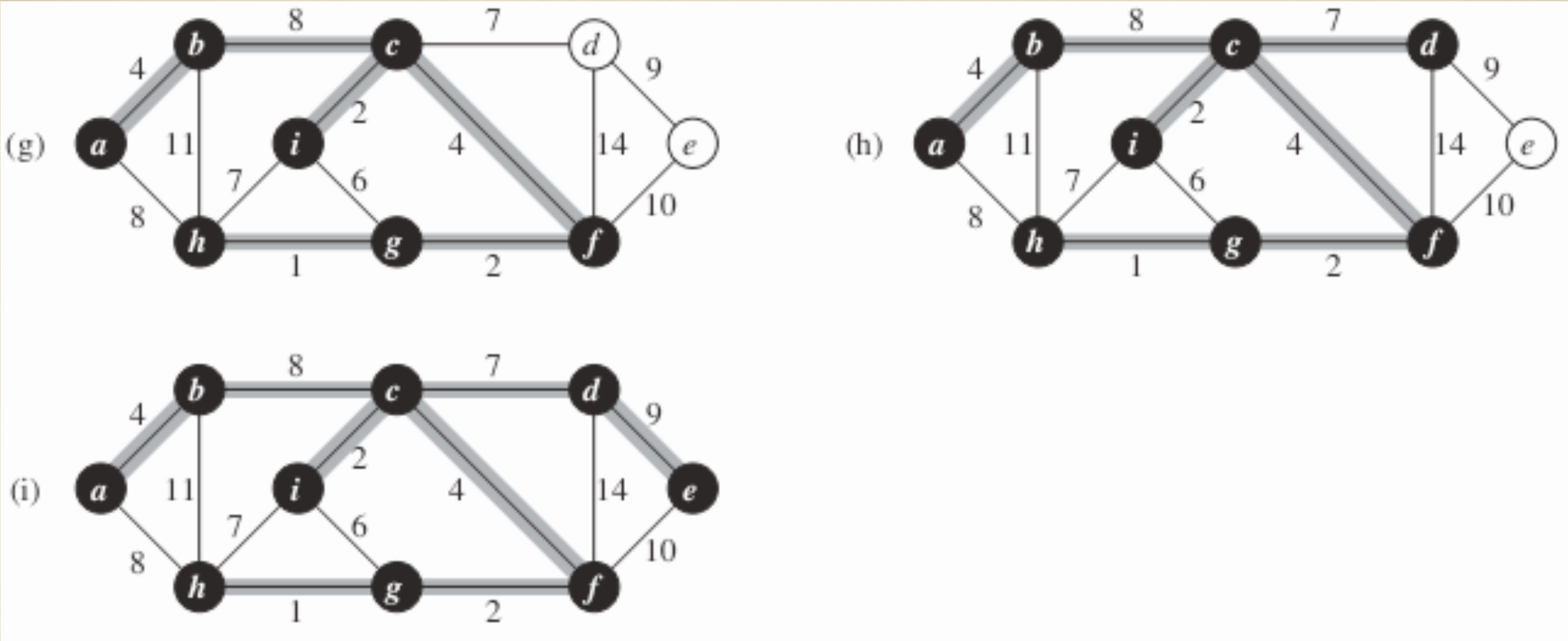
- Answer



Prim's Algorithm



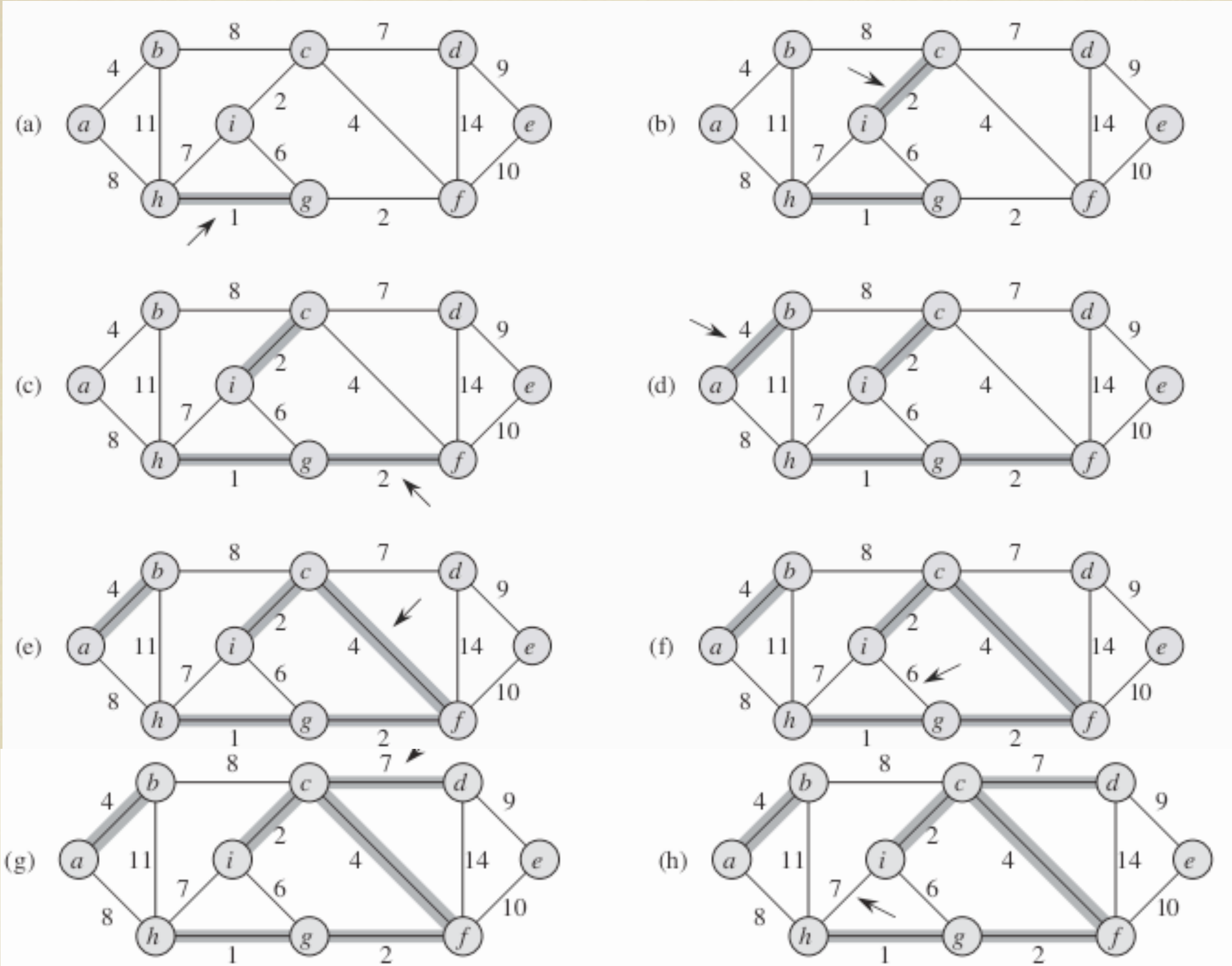
Prim's Algorithm



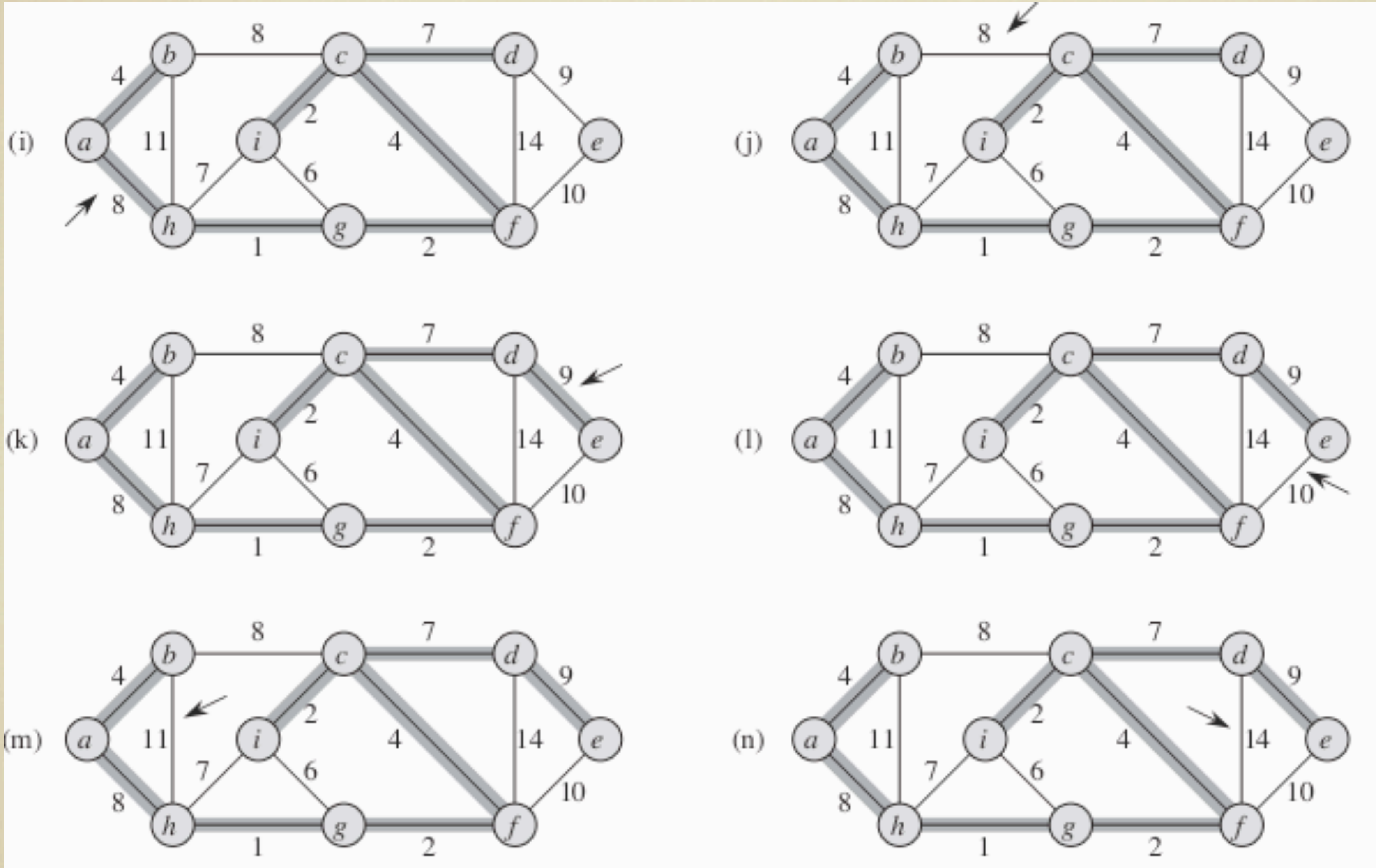
Prim's Algorithm

```
vector< vector< pair<int,int> > > g;

int prim()
{
  priority_queue< pair<int,int>, vector< pair<int,int> >, greater< pair<int,int> > > q;
  bool vis[n] = {0};
  pair<int,int> p; int v,w, ret = 0;
  q.push( make_pair(0,0) );
  while (!q.empty())
    {
      p = q.top(); q.pop();
      v = p.second;
      w = p.first;
      if (vis[v] != 0) continue;
      vis[v] = 1;
      ret += w;

      for (int i = 0; i < g[v].size(); ++i)
         if ( !vis[ g[v][i].first ] )
           {
             q.push( make_pair( g[v][i].second, g[v][i].first ) );
           }
    }
  return ret;
}
```

Kruskal's Algorithm



Kruskal's Algorithm

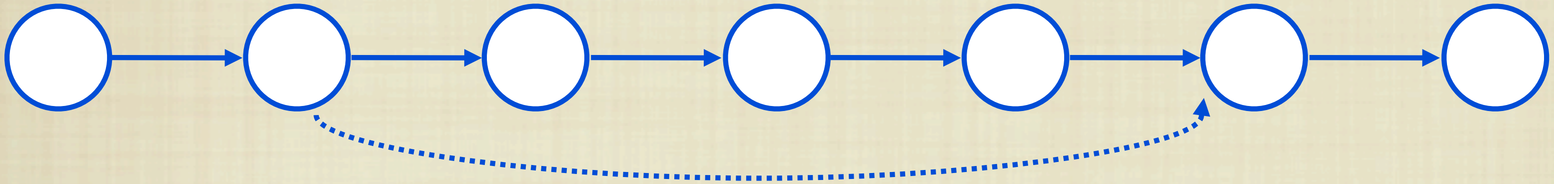


Single-Source Shortest Path

- Problem: given a weighted directed graph G, find the minimum-weight path from a given source vertex s to another vertex v
  - "Shortest-path" = minimum weight
  - Weight of path is sum of edges

Shortest Path Properties

- we have *optimal substructure*: the shortest path consists of shortest subpaths:



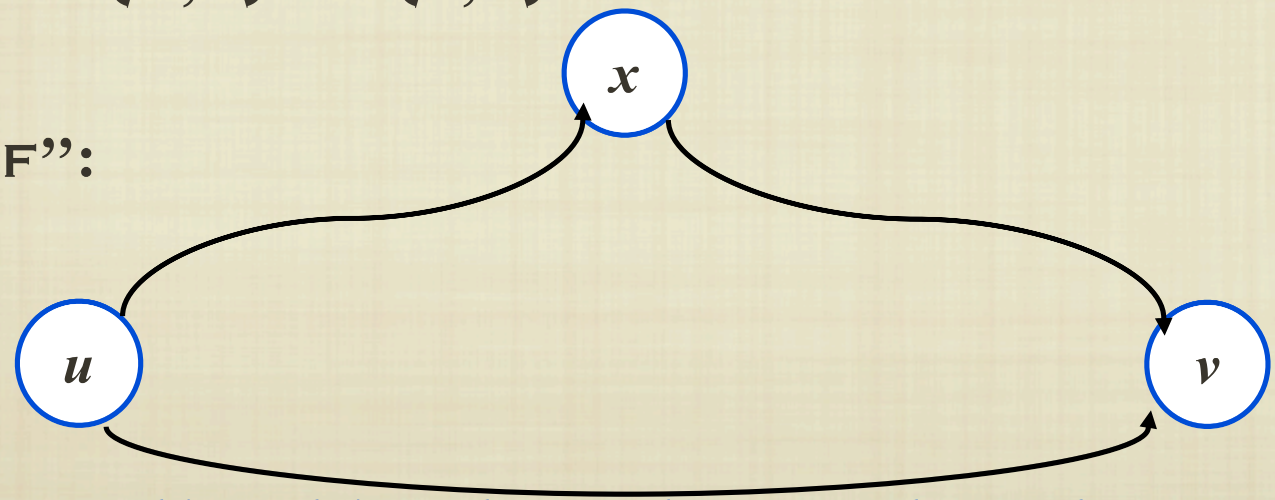
- Proof: suppose some subpath is not a shortest path
  - There must then exist a shorter subpath
  - Could substitute the shorter subpath for a shorter path
  - But then overall path is not shortest path. Contradiction

Shortest Path Properties

- Define $\delta(u,v)$ to be the weight of the shortest path from u to v

- Shortest paths satisfy the *triangle inequality*: $\delta(u,v) \leq \delta(u,x) + \delta(x,v)$

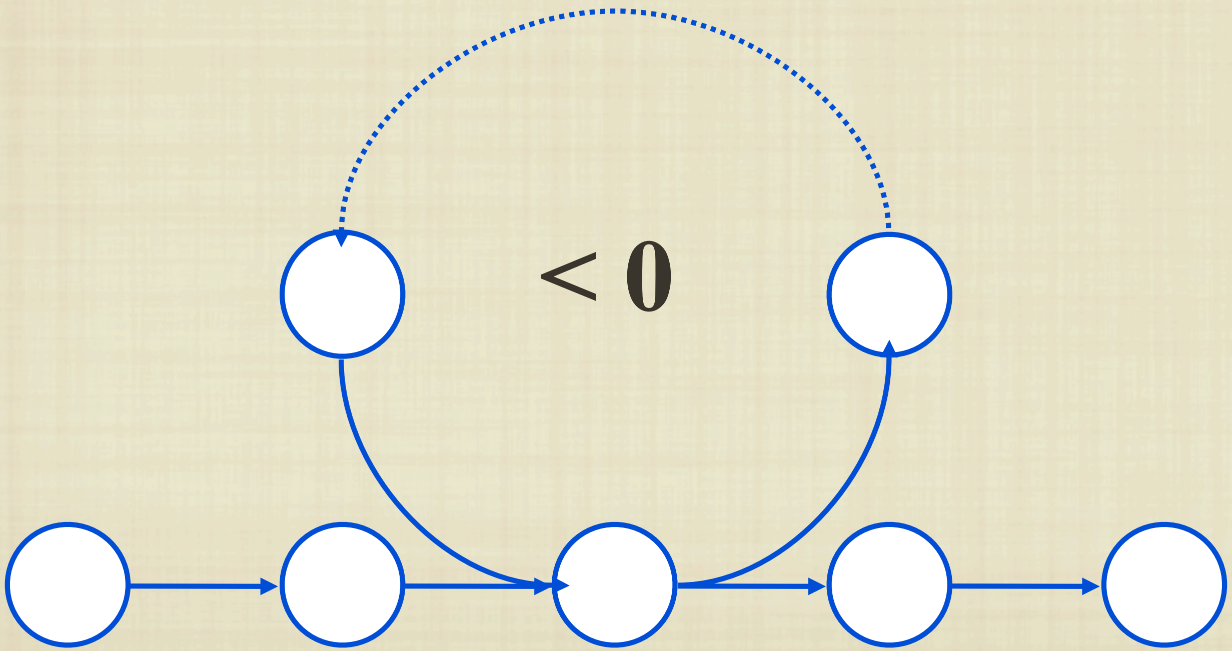
- "Proof":



This path is no longer than any other path

Shortest Path Properties

- In graphs with negative weight cycles, some shortest paths will not exist:



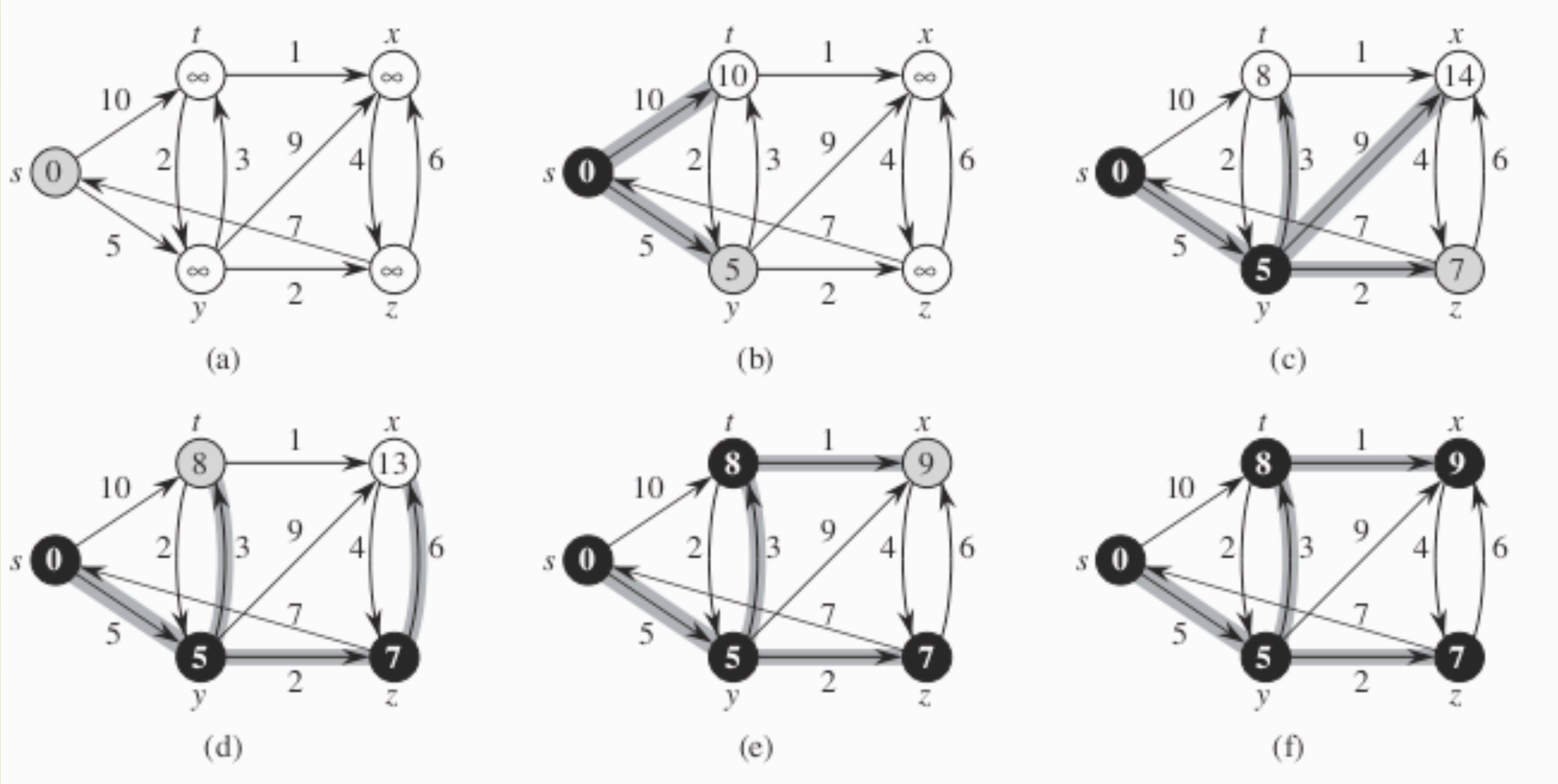
Dijkstra's Algorithm

- no negative edge weights
- Similar to breadth-first search
  - Grow a tree gradually, advancing from vertices taken from a queue
- Also similar to Prim's algorithm for MST
  - Use a priority queue keyed on d[v]

Dijkstra Algorithm

```
vector< vector< pair<int,int> > > g;

int dijkstra(int a, int b)
{
  priority_queue< pair<int,int>, vector< pair<int,int> >, greater< pair<int,int> > > q;
  vector< int > d(n, -1);
  pair<int,int> p; int v,w;
  q.push( make_pair(0,a) );
  while (!q.empty())
    {
      p = q.top(); q.pop();
      v = p.second;
      w = p.first;
      if ( d[v] != -1 ) continue;
      if (v == b) return w;
      d[v] = w;

      for (int i = 0; i < g[v].size(); ++i)
        if ( d[ g[v][i].first ] == -1 )
          {
            q.push( make_pair( w+g[v][i].second, g[v][i].first ) );
          }
    }
  return -1;
}
```

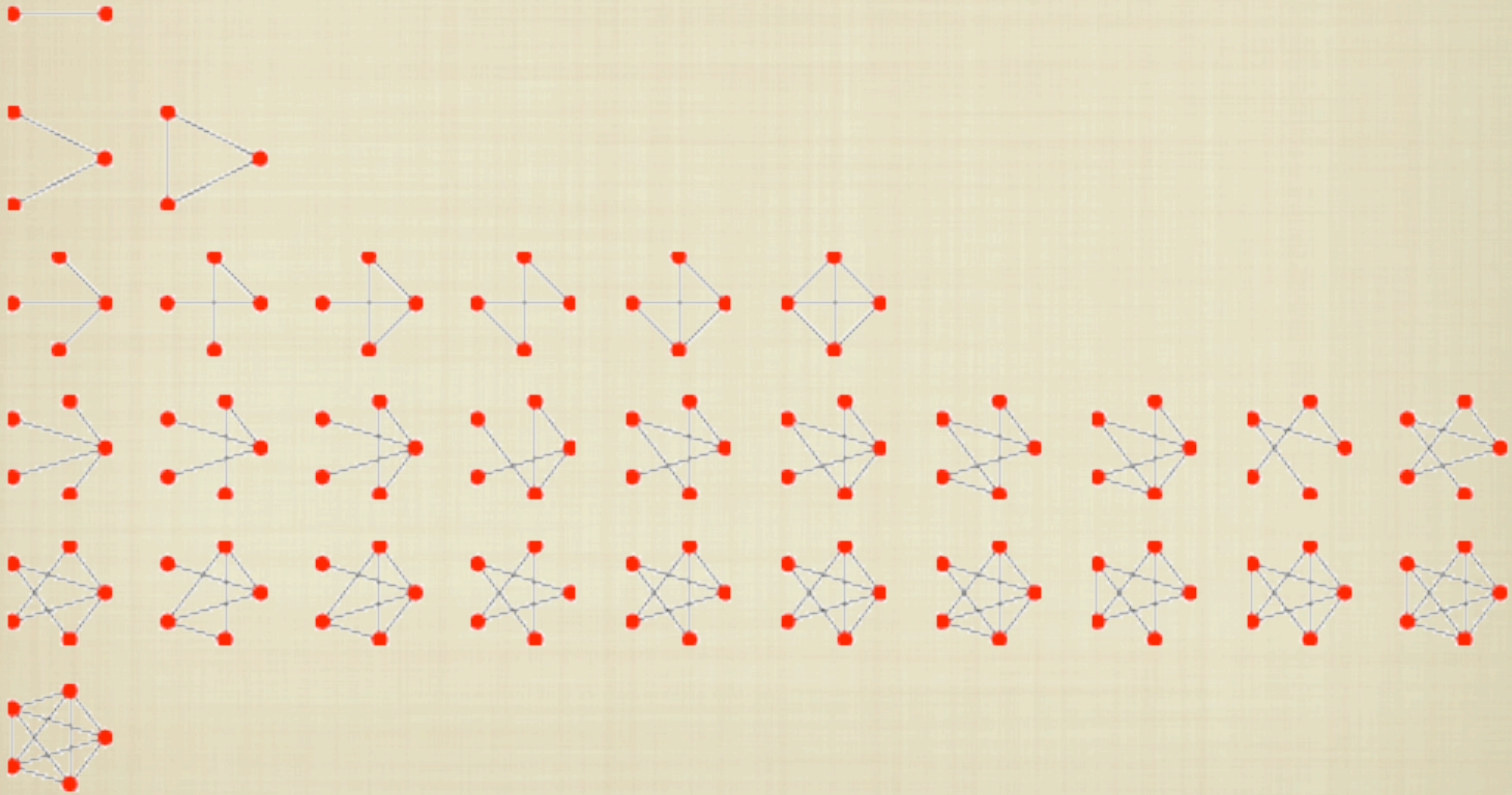
Dijkstra Algorithm



Floyd

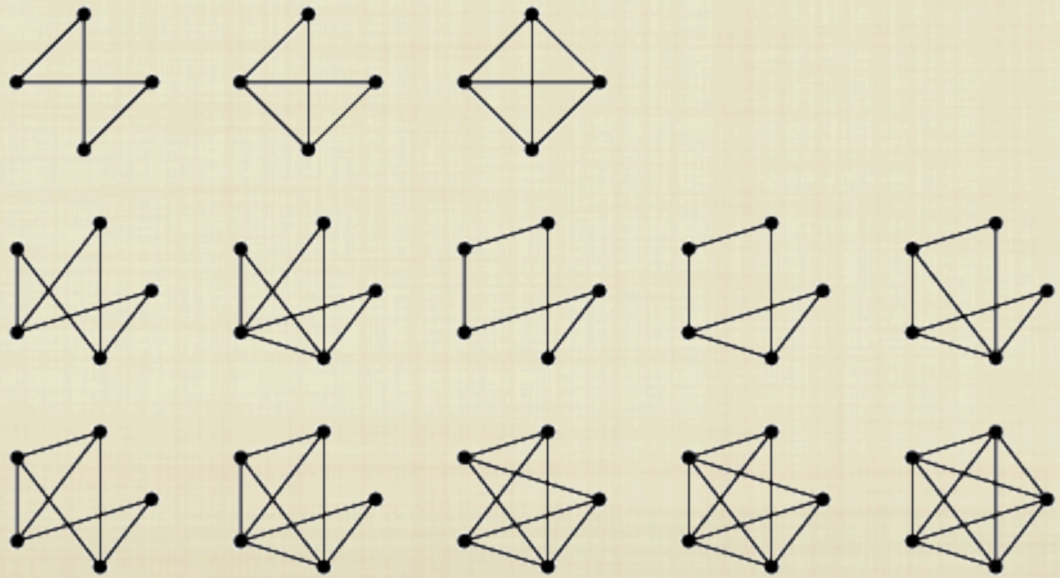
```
int main() {
  int V, E, u, v, w, AdjMatrix[200][200];
  scanf("%d %d", &V, &E);
  for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++)
      AdjMatrix[i][j] = INF;
    AdjMatrix[i][i] = 0;
  }

  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);
    AdjMatrix[u][v] = w; // directed graph
  }

  for (int k = 0; k < V; k++) // common error: remember that loop order is k->i->j
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
        AdjMatrix[i][j] = min(AdjMatrix[i][j], AdjMatrix[i][k] + AdjMatrix[k][j]);

  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      printf("APSP(%d, %d) = %d\n", i, j, AdjMatrix[i][j]);

  return 0;
}
```

Connected Graphs



there is a path from any point to any other point in the graph

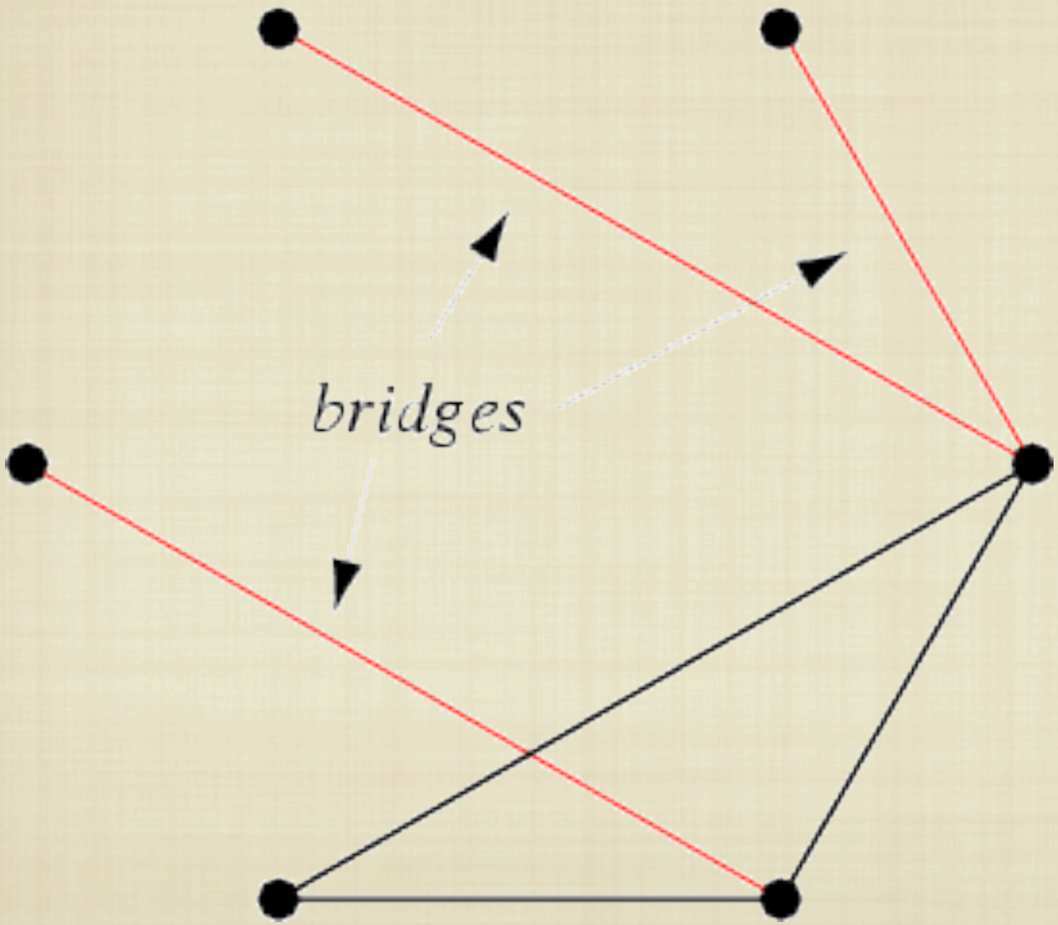
Bi-Connected Graphs



a **biconnected** graph is a connected graph with no articulation vertices.

In other words, a **biconnected** graph is connected and **nonseparable**, meaning that if any vertex were to be removed, the graph will remain connected.

Bridges

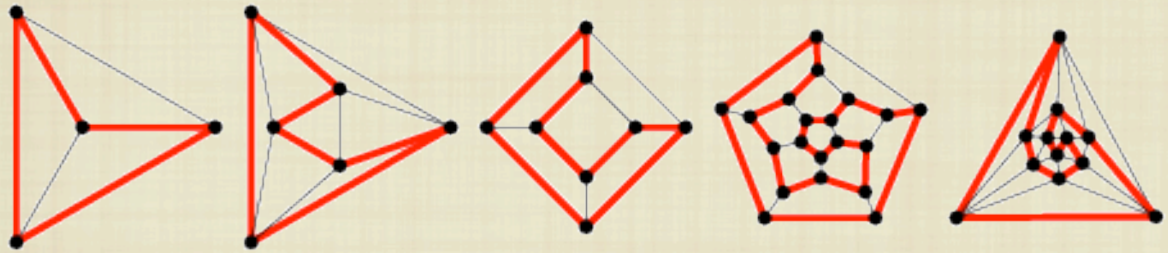


Eulerian Cycle

- **Eulerian Cycle:**
  - walk on the graph edges of a graph which uses each graph edge in the original graph exactly once.
  - connected graphs have an Eulerian tour iff it is connected and every vertex is of even degree
  - Algorithm:
    - find a simple cycle in the graph using the DFS-based algorithm
    - Deleting the edges on this cycle leaves each vertex with even degree.
    - Once we have partitioned the edges into edge-disjoint cycles, we can merge these cycles arbitrarily at common vertices to build an Eulerian cycle.

Hamiltonian Tour

- **Hamiltonian Cycle:**
  - A Hamiltonian cycle is a graph cycle (i.e., closed loop) through a graph that visits each node exactly once
  - By convention, the trivial graph on a single node is considered to posses a Hamiltonian circuit
  - but the connected graph on two nodes is not.



  - If graph small, solve via backtracking.
  - Each Hamiltonian cycle is described by a permutation of the vertices.
  - Backtrack whenever there does not exist an edge from the latest vertex to an unvisited one.