05 Graph (4)

College of Computer Science, CQU

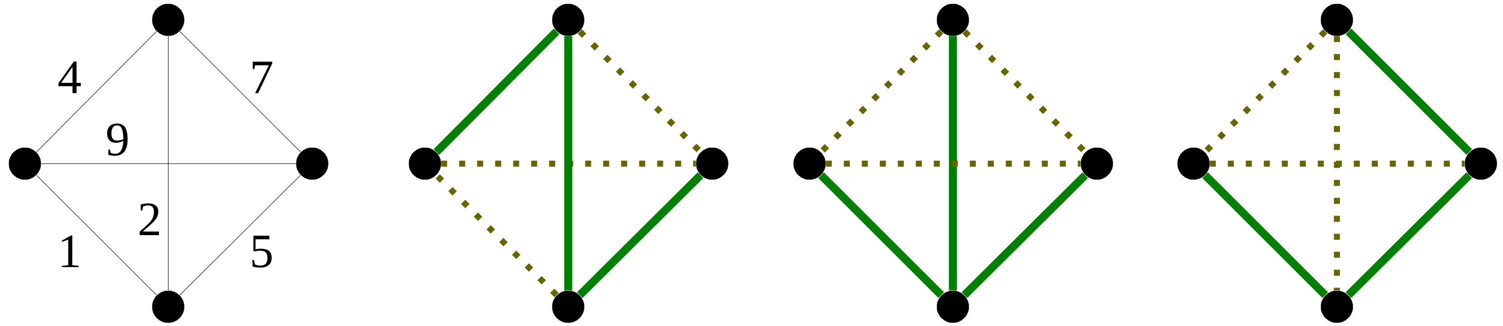
Outline

- Minimum-Cost Spanning Trees
- Kruskal's Algorithm
- Prim's Algorithm

Spanning Tree

- **Spanning tree** - a subset of the edges from a connected graph that:
  - touches all vertices in the graph (**spans** the graph)
  - forms a tree (is connected and contains no cycles)

- **Minimum spanning tree** - spanning tree with lowest total edge cost



Kruskal's Algorithm (扩边法)

- Yet another greedy algorithm

- Initialize all vertices to unconnected
- While there are still unmarked edges
  - Pick the lowest cost edge **`e = (u, v)`** and mark it
  - If **u** and **v** are not already connected, add **e** to the minimum spanning tree and connect **u** and **v**

- How is this like maze generation?
- How is it different?

Kruskal's Algorithm

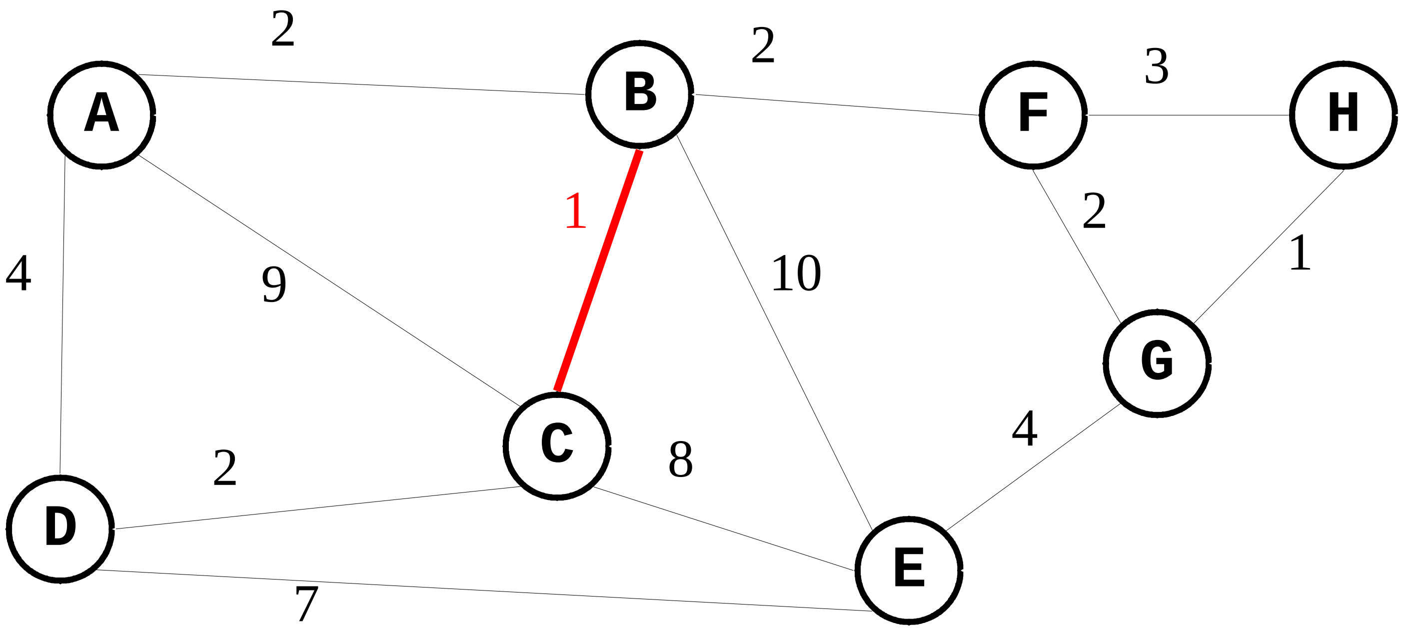
Algorithm:

```
T={ };
 while (T contains less than n-1 edges &&
                 E is not empty ){
         Choose a least cost edge (v,w) from E;
         delete (v,w) from E;
         if ((v,w) does not create a cycle in T)
              add (v,w) into T;
        else discard (v,w);
 }
 if (T contains fewer than n-1 edges)
          print ("No spanning tree\n");
```

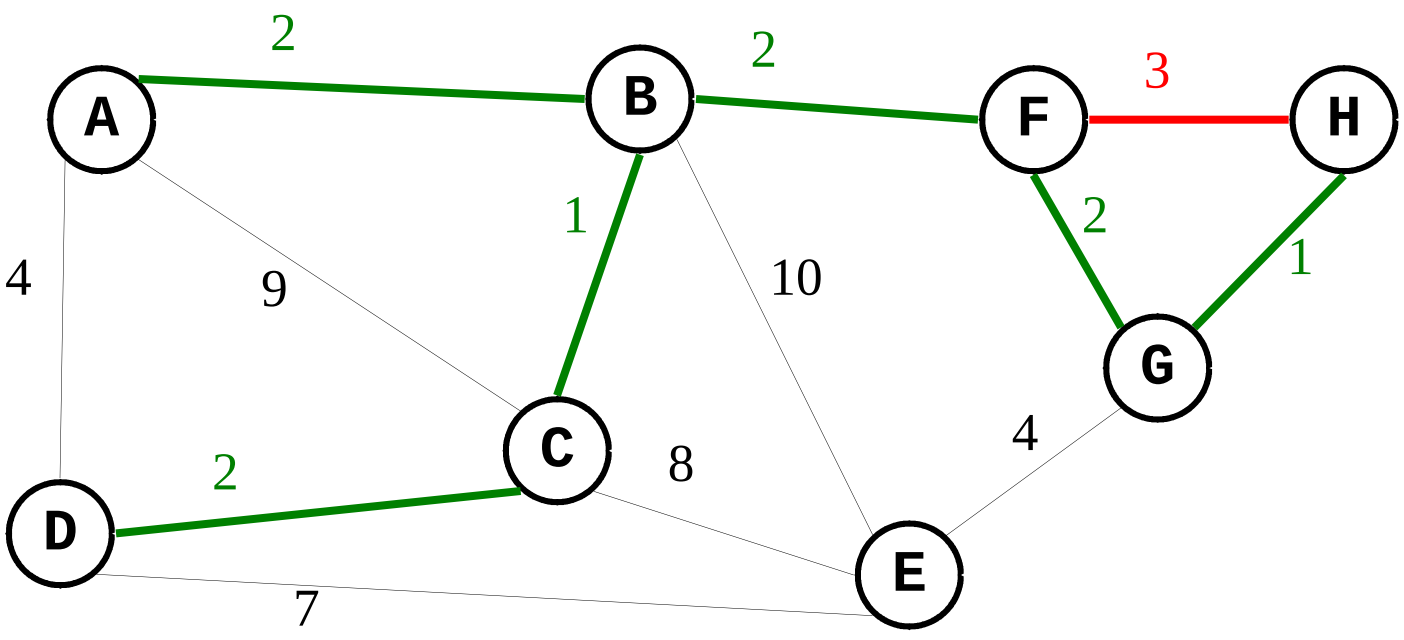
```
class KruskElem {          // An element for the heap
public:
  int from, to, distance; // The edge being stored
  KruskElem() { from = -1;  to = -1; distance = -1; }
  KruskElem(int f, int t, int d)
    { from = f; to = t; distance = d; }
};

void Kruskel(Graph* G) {    // Kruskal's MST algorithm
  ParPtrTree A(G->n());     // Equivalence class array
  KruskElem E[G->e()];      // Array of edges for min-heap
  int i;
  int edgecnt = 0;
  for (i=0; i<G->n(); i++) // Put the edges on the array
    for (int w=G->first(i); w<G->n(); w = G->next(i,w)) {
      E[edgecnt].distance = G->weight(i, w);
      E[edgecnt].from = i;
      E[edgecnt++].to = w;
    }
```

```
  // Heapify the edges
  heap<KruskElem, Comp> H(E, edgecnt, edgecnt);
  int numMST = G->n();        // Initially n equiv classes
  for (i=0; numMST>1; i++) { // Combine equiv classes
    KruskElem temp;
    temp = H.removefirst(); // Get next cheapest edge
    int v = temp.from;  int u = temp.to;
    if (A.differ(v, u)) {  // If in different equiv classes
      A.UNION(v, u);       // Combine equiv classes
      AddEdgetoMST(temp.from, temp.to);  // Add edge to MST
      numMST--;            // One less MST
    }
  }
}
```

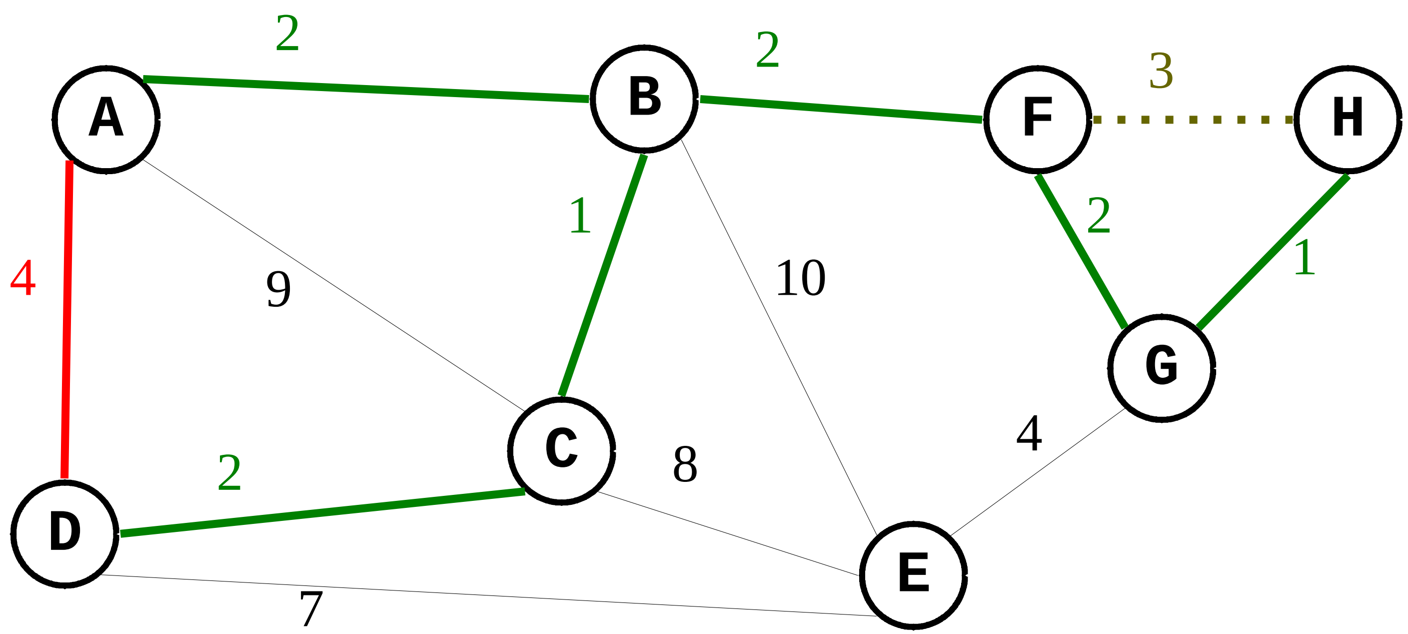
Kruskal's Algorithm in Action (1/5)



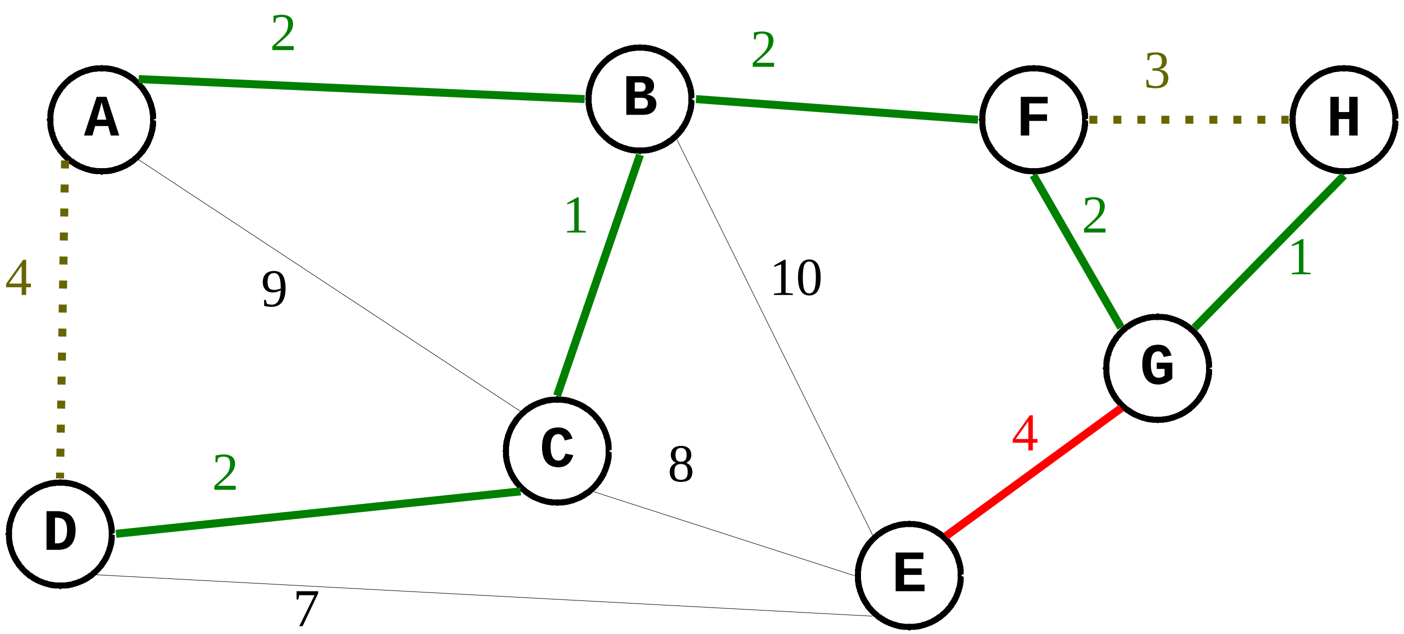
Kruskal's Algorithm in Action (2/5)



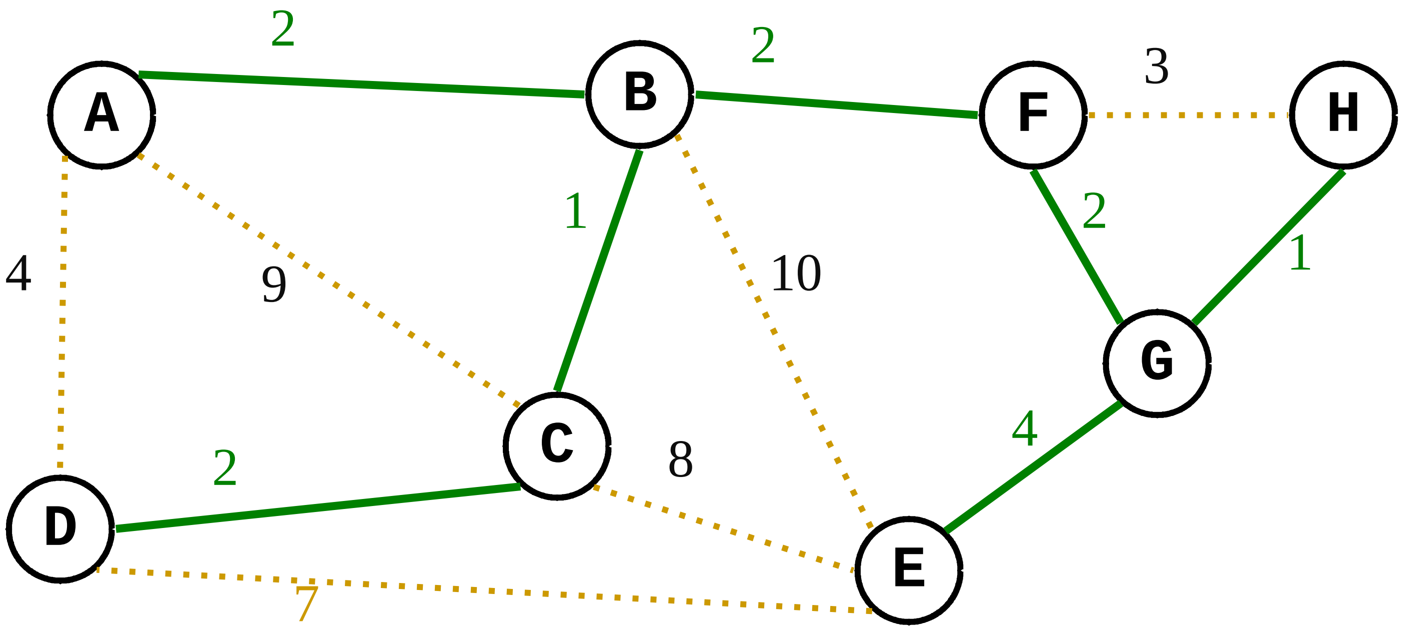
Kruskal's Algorithm in Action (3/5)



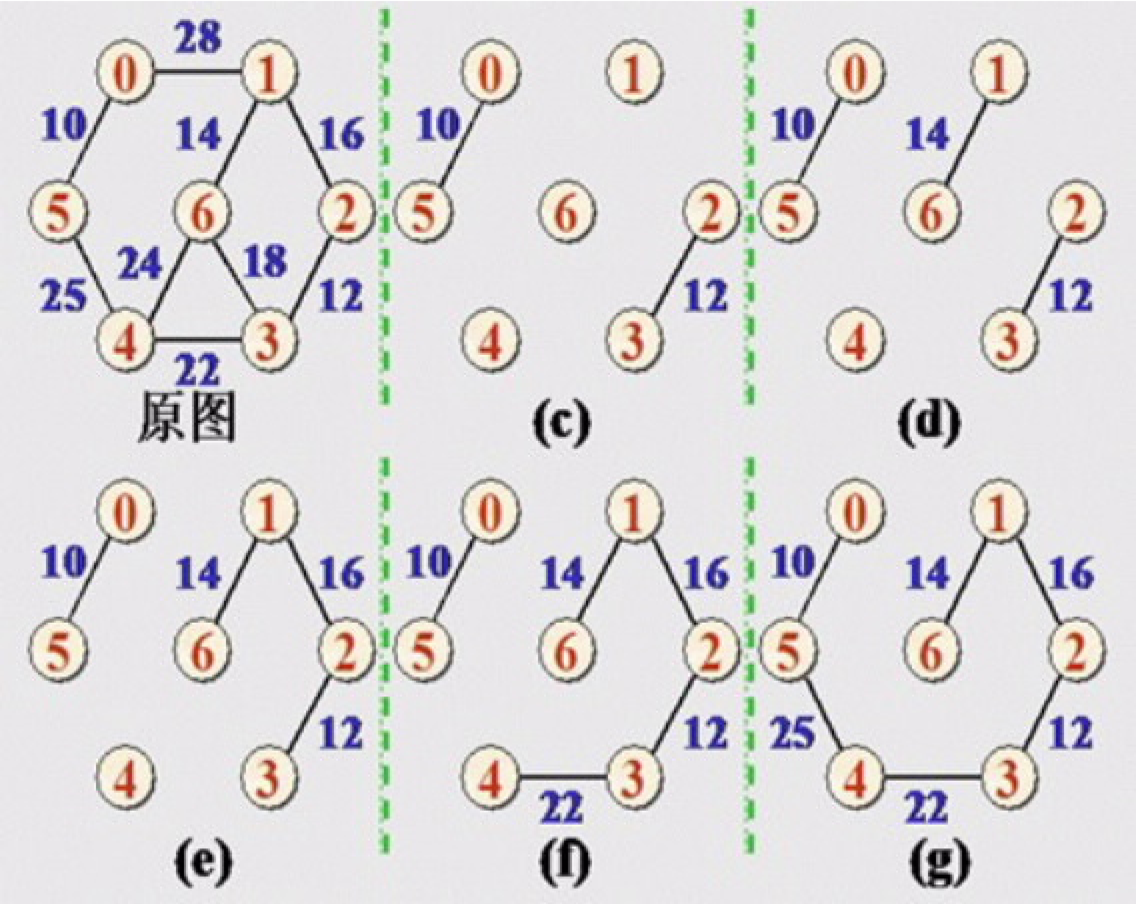
Kruskal's Algorithm in Action (4/5)



Kruskal's Algorithm Completed (5/5)



Another Example



Prim’s Algorithm

- Prim’s Algorithm (a variation of Dijkstra’s Algorithm) also finds Minimum Spanning Trees:
  - Pick an initial node
  - Until graph is connected:

    Choose edge (u,v) which is of minimum cost among edges where u is in tree but v is not

    Add (u,v) to the tree

- Same “greedy” proof, same asymptotic complexity

Prim's Algorithm(扩点法)

Process:

```
T={ };
TV = { 0 }; // start with vertex 0 and no edges
 while (T contains fewer than n-1 edges) {
      let (u,v) be a least cost edge such that
            u∈TV and not v∈TV;
      if (there is no such edges) break;
      add v into Tv;
      add (u, v) into T;
 }
if (T contains fewer than n-1 edges)
      print("No spanning tree");
```

Prim’s Algorithm(扩点法)

```
void Prim(Graph* G, int* D, int s) { // Prim's MST algorithm
  int V[G->n()];                     // Store closest vertex
  int i, w;
  for (i=0; i<G->n(); i++) {         // Process the vertices
    int v = minVertex(G, D);
    G->setMark(v, VISITED);
    if (v != s)
      AddEdgetoMST(V[v], v);         // Add edge to MST
    if (D[v] == INFINITY) return;    // Unreachable vertices
    for (w=G->first(v); w<G->n(); w = G->next(v,w))
      if (D[w] > G->weight(v,w)) {
        D[w] = G->weight(v,w);       // Update distance
        V[w] = v;                    // Where it came from
      }
  }
}
```

Prim’s Algorithm(扩点法)

```
// Prim's MST algorithm: priority queue version
void Prim(Graph* G, int* D, int s) {
  int i, v, w;                 // "v" is current vertex
  int V[G->n()];               // V[I] stores I's closest neighbor
  DijkElem temp;
  DijkElem E[G->e()];          // Heap array with lots of space
  temp.distance = 0; temp.vertex = s;
  E[0] = temp;                 // Initialize heap array
  heap<DijkElem, DDComp> H(E, 1, G->e()); // Create heap
```

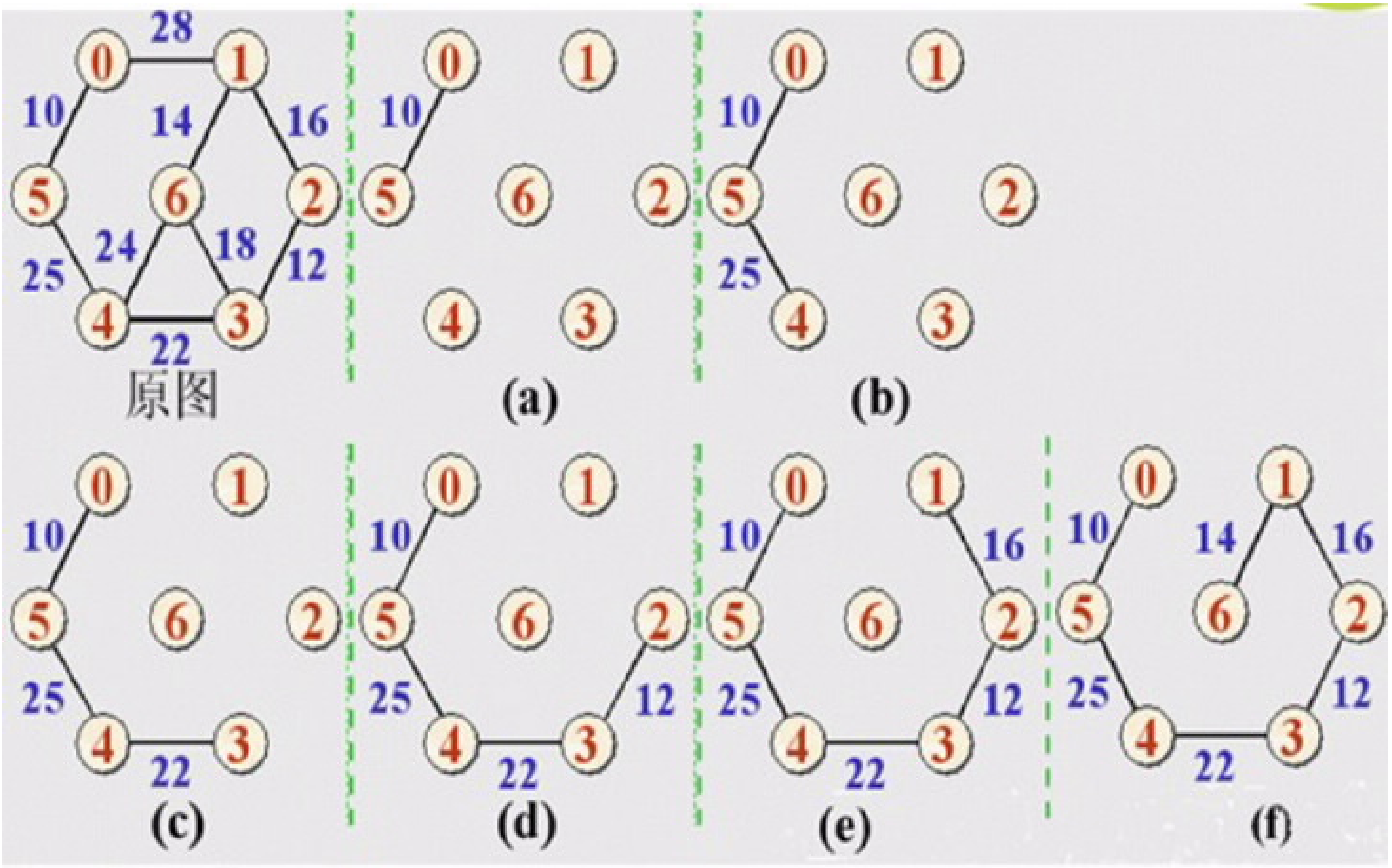
Prim's Algorithm(扩点法)

```
for (i=0; i<G->n(); i++) {              // Now build MST
  do {
    if(H.size() == 0) return; // Nothing to remove
    temp = H.removefirst();
    v = temp.vertex;
  } while (G->getMark(v) == VISITED);
  G->setMark(v, VISITED);
  if (v != s) AddEdgetoMST(V[v], v); // Add edge to MST
  if (D[v] == INFINITY) return;       // Ureachable vertex
```

Prim's Algorithm(扩点法)

```
  for (w=G->first(v); w<G->n(); w = G->next(v,w))
    if (D[w] > G->weight(v, w)) {     // Update D
      D[w] = G->weight(v, w);
      V[w] = v;              // Update who it came from
      temp.distance = D[w]; temp.vertex = w;
      H.insert(temp);   // Insert new distance in heap
    }
  }
}
```

Example of Prim's Algorithm



Knowledge Points

- Chapter 11, pp.402-409

Homework

- P410, 11.17
- P411,11.18-11.22 P410, 11.17
- P411,11.18-11.22

-End-