最小生成树

张晓平

December 6, 2016

定义 1. 边赋以权值的图称为网或带权图，带权图的生成树也是带权的，生成树T各边的权值总和称为该树的权。

定义 2. 最小生成树（MST）：权值最小的生成树。

生成树和最小生成树的应用：要连通n个城市需要$n-1$条边（线路），可以把边上的权值解释为线路的造价，则最小生成树表示使其造价最小的生成树。构造网的最小生成树必须解决下面两个问题：

1. 尽可能选取权值小的边，但不能构成回路；
2. 选取$n-1$条恰当的边以连通n个顶点。

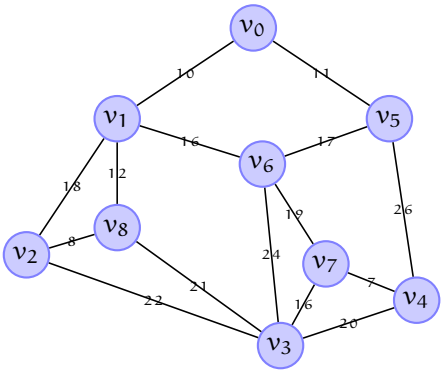


图 1: 带权连通图

性质 1 (MST). 假设$G=(V,E)$是一个连通网，U是顶点集V的一个非空子集。若(u,v)是一条具有最小权值的边，其中$u\in U, v\in V-U$，则必存在一棵包含边(u,v)的最小生成树。

1 Prim算法

1.1 算法描述

假设$G=(V,E)$是连通的，E_{tree}是G上最小生成树中边的集合。算法从$U=\{u_0\}(u_0\in V), E_{tree}=\{\}$开始。重复执行下列操作：

在所有$u\in U, v\in V-U$的边$(u,v)\in E$中找一条权值最小的边(u_0,v_0)并入集合E_{tree}中，同时v_0并入U，直到$U=V$为止。此时，E_{tree}中必有$n-1$条边，$T=(V,E_{tree})$为G的最小生成树。

Prim算法的核心：始终保持E_{tree}中的边集构成一棵生成树。

1.2 算法解释

以图1为例，介绍Prim算法的具体实现过程。

1、选择v_0作为起始点，并设U为当前所找到最小生成树的顶点，E_{tree}为所找到的边，现在状态如图2：

$$U=\{v_0\},\quad E_{tree}=\{\}.$$

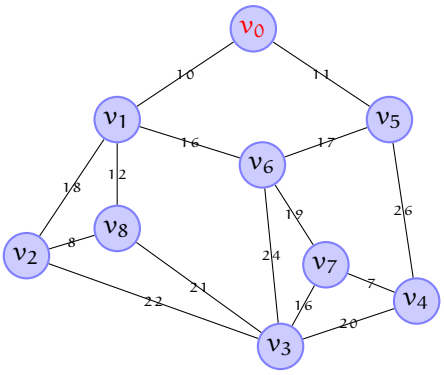


图 2: 初始状态

2、查找一个顶点在U中，另一个顶点在V − U中的最小权值，即在图3的红线中找最小值。通过观察可知边(v_0, v_1)的权值最小，则将v_1加入到U中，(v_0, v_1)加入到E_{tree}，状态如下：

$$U = \{v_0, v_1\}, \quad E_{tree} = \{(v_0, v_1)\}.$$

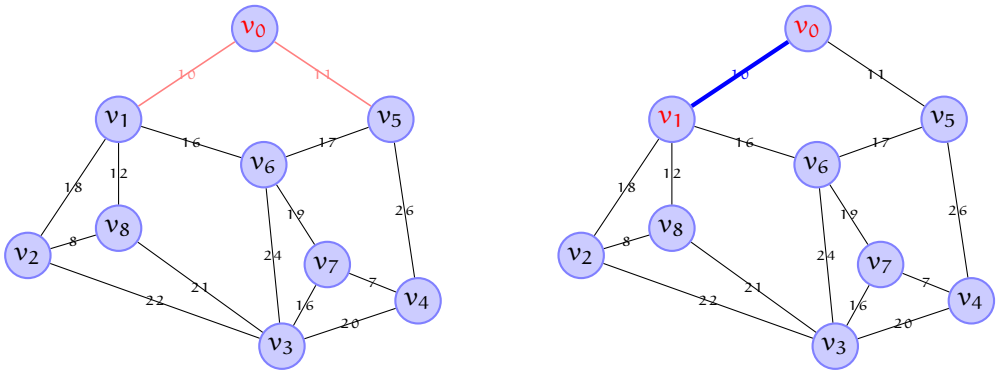


图 3: 状态2

3、继续查找一个顶点在U中，另一个顶点在V − U中的最小权值，即在图4的红线中找最小值。通过观察可知边(v_0, v_5)的权值最小，则将v_5加入到U中，(v_0, v_5)加入到E_{tree}，状态如下：

$$U = \{v_0, v_1, v_5\}, \quad E_{tree} = \{(v_0, v_1), (v_0, v_5)\}.$$

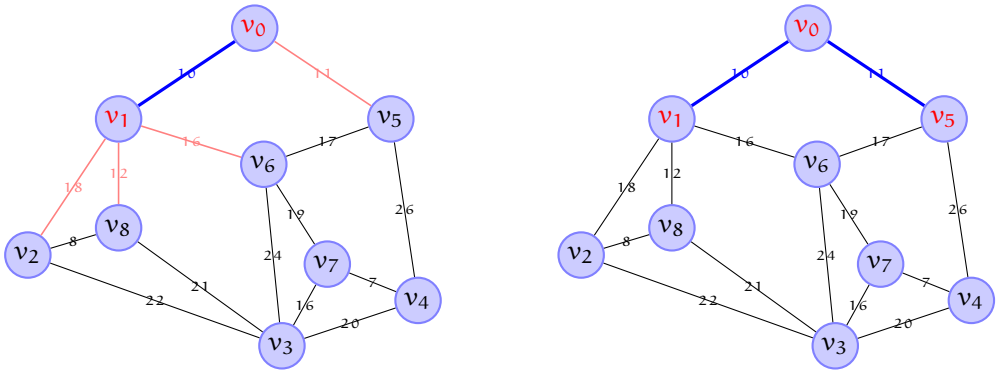


图 4: 状态3

4、继续查找一个顶点在U中，另一个顶点在V − U中的最小权值，即在图5的红线中找最小值。通过观察可知边(v_1, v_8)的权值最小，则将v_8加入到U中，(v_1, v_8)加入到E_{tree}，状态如下：

$$U = \{v_0, v_1, v_5, v_8\}, \quad E_{tree} = \{(v_0, v_1), (v_0, v_5), (v_1, v_8)\}.$$

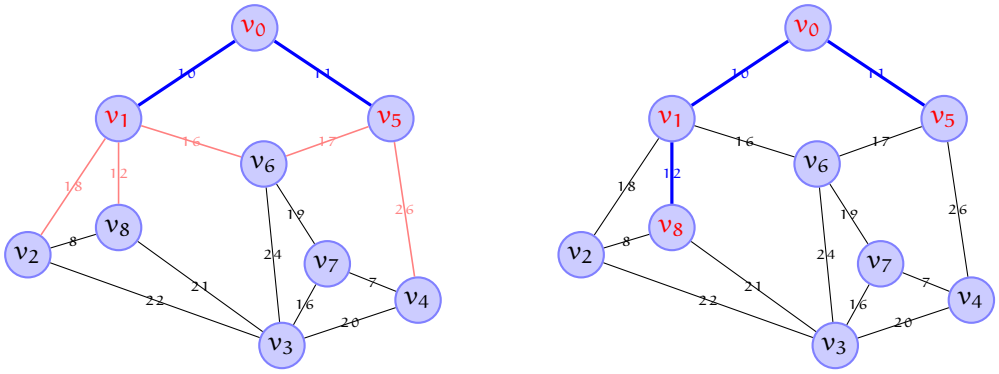


图 5: 状态4

5、继续查找一个顶点在U中，另一个顶点在$V-U$中的最小权值，即在图6的红线中找最小值。通过观察可知边(v_2,v_8)的权值最小，则将v_2加入到U中，(v_2,v_8)加入到E_{tree}，状态如下：

$$U=\{v_0,v_1,v_5,v_8,v_2\},\quad E_{tree}=\{(v_0,v_1),(v_0,v_5),(v_1,v_8),(v_2,v_8)\}.$$

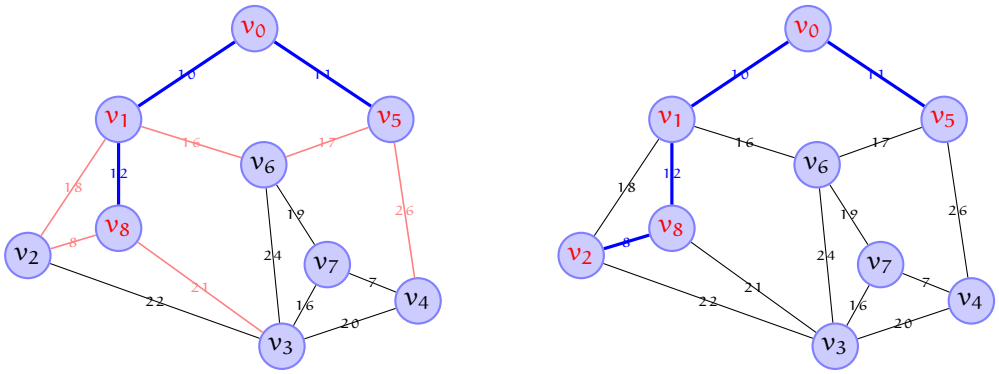


图 6: 状态5

6、继续查找一个顶点在U中，另一个顶点在$V-U$中的最小权值，即在图7的红线中找最小值。通过观察可知边(v_1,v_6)的权值最小，则将v_6加入到U中，(v_1,v_6)加入到E_{tree}，状态如下：

$$U=\{v_0,v_1,v_5,v_8,v_2,v_6\},\quad E_{tree}=\{(v_0,v_1),(v_0,v_5),(v_1,v_8),(v_2,v_8),(v_1,v_6)\}.$$

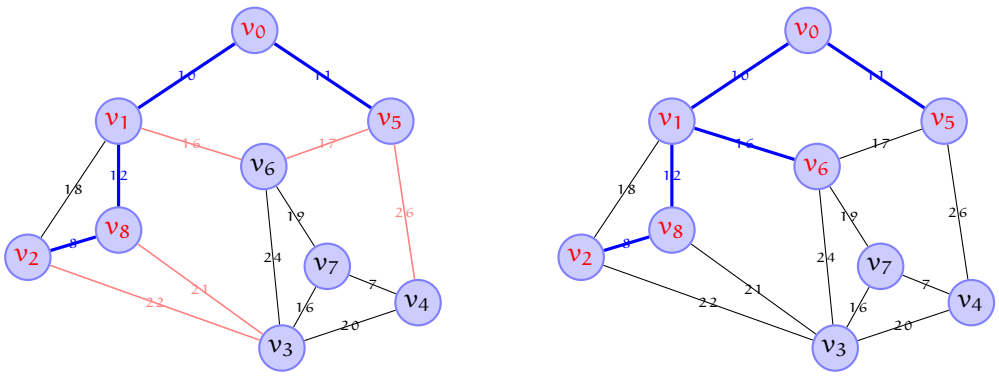


图 7: 状态6

7、继续查找一个顶点在U中，另一个顶点在$V-U$中的最小权值，即在图8的红线中找最小值。通过观察

可知边(v_6, v_7)的权值最小，则将v_7加入到U中，(v_6, v_7)加入到E_{tree}，状态如下：

$$U = \{v_0, v_1, v_5, v_8, v_2, v_6, v_7\},$$
$$E_{tree} = \{(v_0, v_1), (v_0, v_5), (v_1, v_8), (v_2, v_8), (v_1, v_6), (v_6, v_7)\}.$$

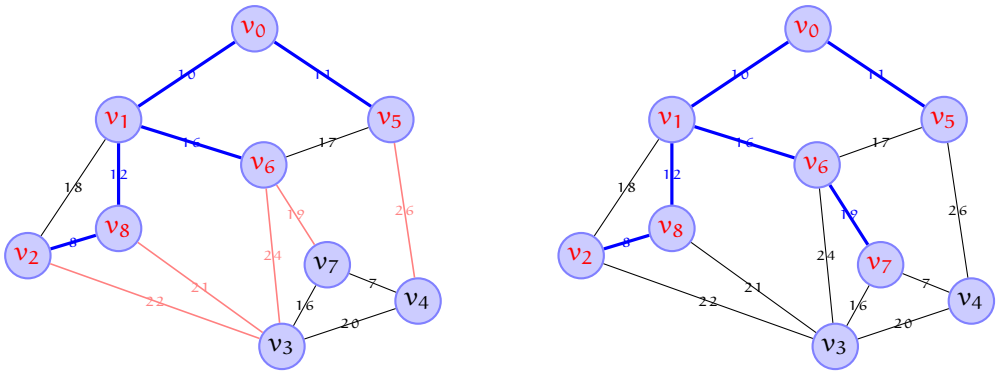


图 8: 状态7

8、继续查找一个顶点在U中，另一个顶点在V − U中的最小权值，即在图9的红线中找最小值。通过观察可知边(v_7, v_4)的权值最小，则将v_4加入到U中，(v_7, v_4)加入到E_{tree}，状态如下：

$$U = \{v_0, v_1, v_5, v_8, v_2, v_6, v_7, v_4\},$$
$$E_{tree} = \{(v_0, v_1), (v_0, v_5), (v_1, v_8), (v_2, v_8), (v_1, v_6), (v_6, v_7), (v_7, v_4)\}.$$

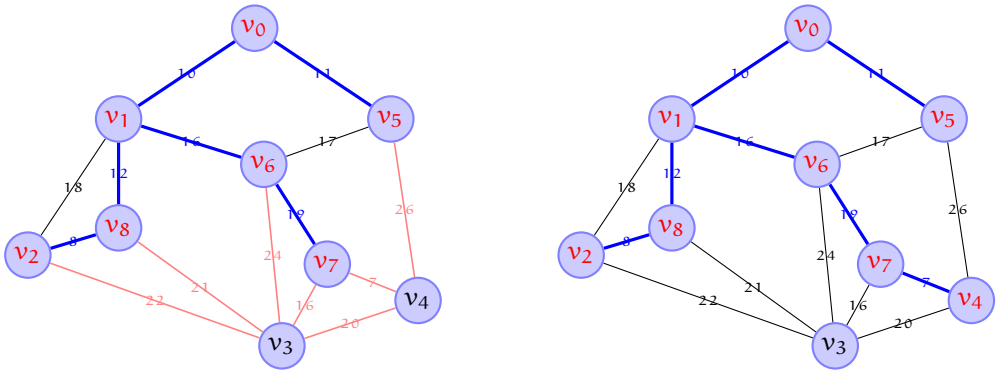


图 9: 状态8

9、继续查找一个顶点在U中，另一个顶点在V − U中的最小权值，即在图10的红线中找最小值。通过观察可知边(v_7, v_3)的权值最小，则将v_3加入到U中，(v_7, v_3)加入到E_{tree}，状态如下：

$$U = \{v_0, v_1, v_5, v_8, v_2, v_6, v_7, v_4, v_3\},$$
$$E_{tree} = \{(v_0, v_1), (v_0, v_5), (v_1, v_8), (v_2, v_8), (v_1, v_6), (v_6, v_7), (v_7, v_4), (v_7, v_3)\}.$$

1.3 代码实现

```
// A C program for Prim's Minimum Spanning Tree (MST) algorithm.
// The program is for adjacency matrix representation of the graph

#include <stdio.h>
#include <limits.h>
#define TRUE 1
#define FALSE 0
typedef int bool;
```

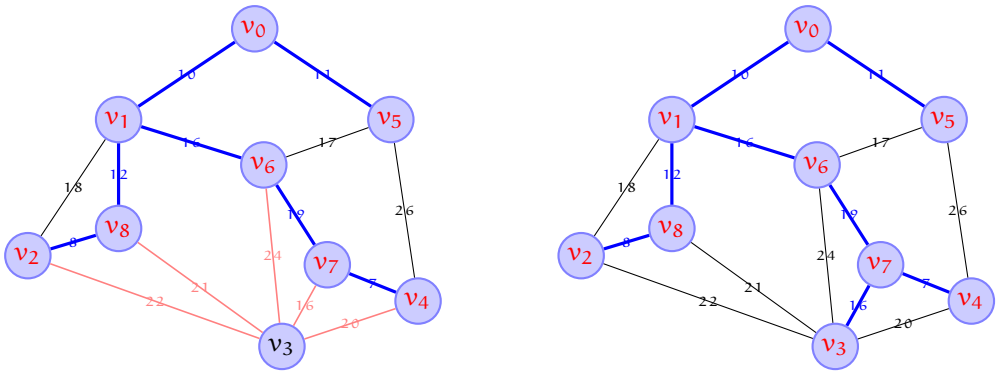
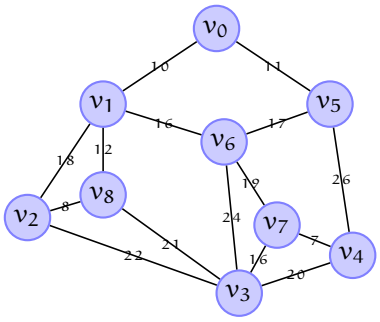


图 10: 状态9



$$
\begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \end{array}
\begin{array}{c}
\begin{array}{ccccccccc} v_0 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 \end{array} \\
\begin{pmatrix}
0 & 10 & 0 & 0 & 0 & 11 & 0 & 0 & 0 \\
10 & 0 & 18 & 0 & 0 & 0 & 16 & 0 & 12 \\
0 & 0 & 0 & 22 & 0 & 0 & 0 & 0 & 8 \\
0 & 0 & 22 & 0 & 20 & 0 & 0 & 16 & 21 \\
0 & 0 & 0 & 20 & 0 & 26 & 0 & 7 & 0 \\
11 & 0 & 0 & 0 & 26 & 0 & 17 & 0 & 0 \\
0 & 16 & 0 & 0 & 0 & 17 & 0 & 19 & 0 \\
0 & 0 & 0 & 16 & 7 & 0 & 19 & 0 & 0 \\
0 & 12 & 8 & 21 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\end{array}
$$

图 11: 邻接矩阵

```
// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with minimum key value, from
// the set of vertices not yet included in MST
int minKey(int key[], bool mstSet[])
{
  // Initialize min value
  int min = INT_MAX, min_index;

  for (int v = 0; v < V; v++)
    if (mstSet[v] == FALSE && key[v] < min)
      min = key[v], min_index = v;
  return min_index;
}

// A utility function to print the constructed MST stored in parent[]
void MST_print(int parent[], int n, int graph[V][V])
{
  printf("Edge␣␣␣Weight\n");
  for (int i = 1; i < V; i++)
    printf("%d␣-␣%d␣␣␣␣%d␣\n", parent[i], i, graph[i][parent[i]]);
}

// Function to construct and print MST for a graph represented using adjacency
// matrix representation
void MST_prim(int graph[V][V])
{
  int parent[V]; // Array to store constructed MST
  int key[V];   // Key values used to pick minimum weight edge in cut
```

```
  bool mstSet[V];  // To represent set of vertices not yet included in MST

  // Initialize all keys as INFINITE
  for (int i = 0; i < V; i++)
    key[i] = INT_MAX, mstSet[i] = FALSE;

  // Always include first 1st vertex in MST.
  key[0] = 0;     // Make key 0 so that this vertex is picked as first vertex
  parent[0] = -1; // First node is always root of MST

  // The MST will have V vertices
  for (int count = 0; count < V-1; count++)
  {
    // Pick the minimum key vertex from the set of vertices
    // not yet included in MST
    int u = minKey(key, mstSet);

    // Add the picked vertex to the MST Set
    mstSet[u] = TRUE;

    // Update key value and parent index of the adjacent vertices of
    // the picked vertex. Consider only those vertices which are not yet
    // included in MST
    for (int v = 0; v < V; v++)

      // graph[u][v] is non zero only for adjacent vertices of m
      // mstSet[v] is FALSE for vertices not yet included in MST
      // Update the key only if graph[u][v] is smaller than key[v]
      if (graph[u][v] && mstSet[v] == FALSE && graph[u][v] <  key[v])
        parent[v]  = u, key[v] = graph[u][v];
  }

  // print the constructed MST
  MST_print(parent, V, graph);
}


// driver program to test above function
int main(void)
{
  /* Let us create the following graph
       2    3
   (0)--(1)--(2)
    |   / \   |
  6| 8/   \5 |7
    | /     \ |
    (3)-------(4)
         9
  */
  int graph[V][V] = {
    {0, 2, 0, 6, 0},
    {2, 0, 3, 8, 5},
    {0, 3, 0, 0, 7},
    {6, 8, 0, 0, 9},
    {0, 5, 7, 9, 0},
  };

  // Print the solution
  MST_prim(graph);

  return 0;
}
```

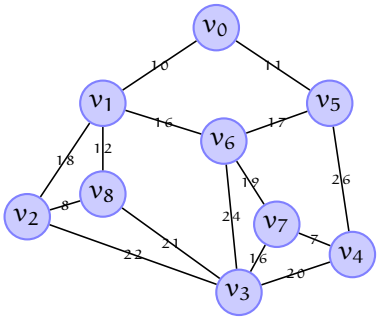
2 Kruskal算法

Kruskal算法与Prim算法的不同之处在于，Kruskal算法在找最小生成树结点之前，需要对所有带权边做从小到大排序。将排序好的带权边依次加入到最小生成树中，如果加入时产生回路就跳过这条边，加入下一条边。当所有顶点都加入到最小生成树中之后，就找出了最小生成树。

2.1 算法描述

Kruskal算法步骤：设原图为$G=(V,E)$，

1. 新建图G'，G'中拥有原图中相同的顶点，但没有边；
2. 将原图中所有的边按权值从小到大排序；
3. 从权值最小的边开始，如果这条边邻接的两个顶点于图G'中不在同一个连通分量中，则添加这条边到图G'中；
4. 重复3，直至图G'中所有的顶点都在同一个连通分量中。



	begin	end	weight
edge[0]	4	7	7
edge[1]	2	8	8
edge[2]	0	1	10
edge[3]	0	5	11
edge[4]	1	8	12
edge[5]	3	7	16
edge[6]	1	6	16
edge[7]	5	6	17
edge[8]	1	2	18
edge[9]	6	7	19
edge[10]	3	4	20
edge[11]	3	8	21
edge[12]	2	3	22
edge[13]	3	6	24
edge[14]	4	5	26

图 12: 边集数组

2.2 算法解释

以图12为例（注意到所有的边均按权值从小到大排序），介绍Kruskal算法的具体实现过程。

0、新建图G'，包含原图G的所有顶点，但不含边，见图14。

1、在G中查找权值为$edge[0].weight=7$的边(v_4,v_7)，因v_4,v_7在G'中不属于同一连通分量，故将它们添加至G'，见图14。
2、在G中查找权值为$edge[1].weight=8$的边(v_2,v_8)，因v_2,v_8在G'中不属于同一连通分量，故将它们添加至G'，见图15。

3、在G中查找权值为$edge[2].weight=10$的边(v_0,v_1)，因v_0,v_1在G'中不属于同一连通分量，故将它们添加至G'，见图16。
4、在G中查找权值为$edge[3].weight=11$的边(v_0,v_5)，因v_0,v_5在G'中不属于同一连通分量，故将它们添加至G'，见图17。

5、在G中查找权值为$edge[4].weight=12$的边(v_1,v_8)，因v_1,v_8在G'中不属于同一连通分量，故将它们添加至G'，见图18。
6、在G中查找权值为$edge[5].weight=16$的边(v_3,v_7)，因v_3,v_7在G'中不属于同一连通分量，故将它们添加至G'，见图19。

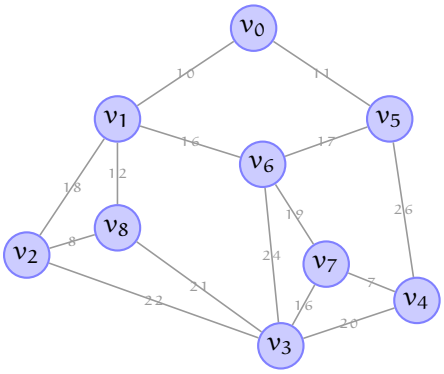


图 13: G′状态0

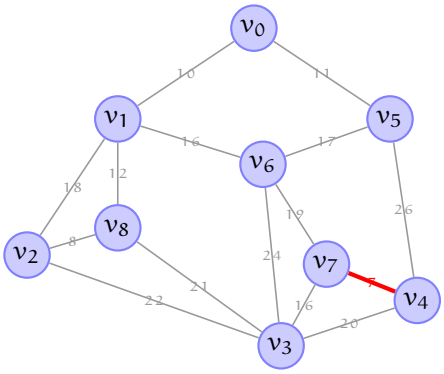


图 14: G′状态1

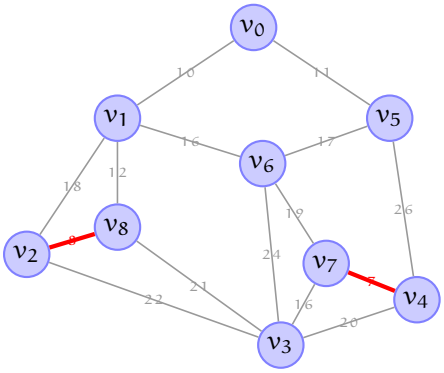


图 15: G′状态2

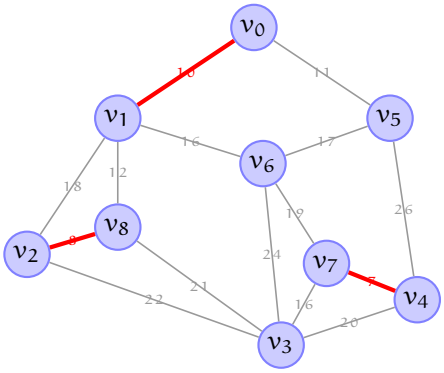


图 16: G′状态3

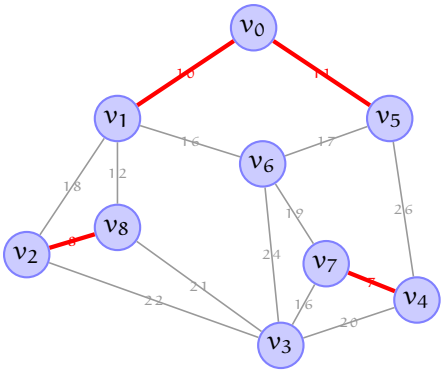


图 17: G′状态4

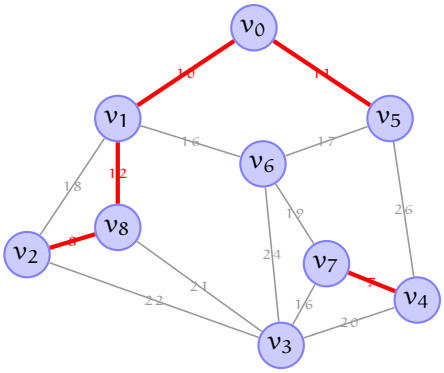


图 18: G′状态5

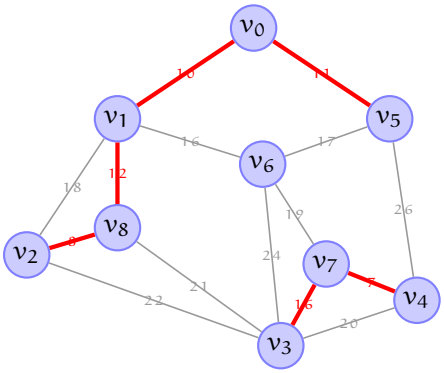


图 19: G′状态6

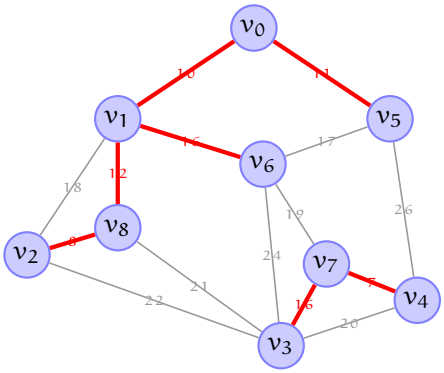


图 20: G′状态7

7、在G中查找权值为$edge[6].weight = 16$的边(v_1, v_6)，因v_1, v_6在G′中不属于同一连通分量，故将它们添加至G′，见图20。
8、在G中找权值为$edge[7].weight = 17$的边(v_5, v_6)，因v_5, v_6在G′中属于同一连通分量，忽略（因添加至G′会构成回路）；在G中继续查找权值为$edge[8].weight = 18$的边(v_1, v_2)，因v_1, v_2在G′中属于同一连通分量，忽略；在G中继续查找权值为$edge[9].weight = 19$的边(v_6, v_7)，因v_6, v_7在G′中不属于同一连通分量，故将它们添加至G′，见图21。

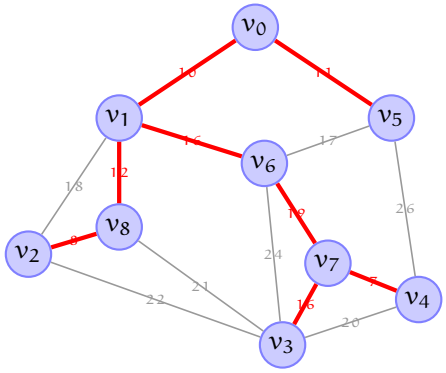


图 21: G′状态8

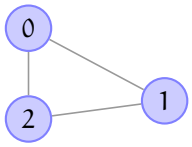
至此，所有顶点在G′中都属于同一连通分量，最小生成树已经找到。

2.3 并查集(Disjoint-set data structure)

并查集是一种数据结构，用于处理一些不相交集合(Disjoint Sets)的合并及查询问题。需要实现的操作有：（1）合并两个集合；（2）判断两个元素是否属于同一集合。

这里，我们将讨论并查集的一个应用：即检测图中是否有回路。并查算法(Union-Find Algorithm)可用于检测无向图是否包含回路，该算法要求图中不能有自回路。

考虑下图：



创建一个一维数组parent[]，将其初始化为-1（表示每个子集都是单元素的）。

```
 0  1  2
-1 -1 -1
```

接着遍历每一条边。

Edge 0-1：查询顶点0和1所在的集合。因它们在不同的集合中，故合并它们。合并过程中，让顶点1作为顶点0的双亲（反之亦可）。

```
0  1  2    <---- 1 is made of parent of 0
1 -1 -1
```

Edge 1-2：顶点1在集合1中，而顶点2在集合2中，故合并它们。合并过程中，可让顶点0作为顶点1的双亲或反之。

```
0  1  2    <---- 2 is made of parent of 1
1  2 -1
```

Edge 0-2：顶点0和顶点2都在集合2中，故添加此边将会出现回路。

```
// A union-find algorithm to detect cycle in a graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent an edge in graph
typedef struct
```

```
{
  int src, dest;
} Edge;

// a structure to represent a graph
typedef struct
{
  // V-> Number of vertices, E-> Number of edges
  int V, E;

  // graph is represented as an array of edges
  Edge * edge;
} Graph;

// Creates a graph with V vertices and E edges
Graph * createGraph(int V, int E)
{
  Graph * graph = (Graph *) malloc( sizeof(Graph) );
  graph->V = V;
  graph->E = E;

  graph->edge = (Edge *) malloc( graph->E * sizeof( Edge ) );

  return graph;
}

// A utility function to find the subset of an element i
int Find(int parent[], int i)
{
  if (parent[i] == -1)
    return i;
  return Find(parent, parent[i]);
}

// A utility function to do union of two subsets
void Union(int parent[], int x, int y)
{
  int xset = Find(parent, x);
  int yset = Find(parent, y);
  parent[xset] = yset;
}

// The main function to check whether a given graph contains
// cycle or not
int isCycle( Graph * graph )
{
  // Allocate memory for creating V subsets
  int * parent = (int *) malloc( graph->V * sizeof(int) );

  // Initialize all subsets as single element sets
  memset(parent, -1, sizeof(int) * graph->V);

  // Iterate through all edges of graph, find subset of both
  // vertices of every edge, if both subsets are same, then
  // there is cycle in graph.
  for(int i = 0; i < graph->E; ++i)
  {
    int x = Find(parent, graph->edge[i].src);
    int y = Find(parent, graph->edge[i].dest);

    if (x == y)
      return 1;
```

```
    Union(parent, x, y);
  }
  return 0;
}

// Driver program to test above functions
int main(void)
{
  /* Let us create following graph
   0
   |  \
   |    \
   1-----2 */
  int V = 3, E = 3;
  Graph * graph = createGraph(V, E);

  // add edge 0-1
  graph->edge[0].src = 0;
  graph->edge[0].dest = 1;

  // add edge 1-2
  graph->edge[1].src = 1;
  graph->edge[1].dest = 2;

  // add edge 0-2
  graph->edge[2].src = 0;
  graph->edge[2].dest = 2;

  if (isCycle(graph))
    printf( "graph␣contains␣cycle.\n" );
  else
    printf( "graph␣doesn't␣contain␣cycle.\n" );

  return 0;
}
```

2.4 利用并查集实现Kruskal算法

```
// C++ program for Kruskal's algorithm to find Minimum Spanning Tree
// of a given connected, undirected and weighted graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent a weighted edge in graph
typedef struct
{
  int src, dest, weight;
} Edge;

// a structure to represent a connected, undirected and weighted graph
typedef struct
{
  // V-> Number of vertices, E-> Number of edges
  int V, E;

  // graph is represented as an array of edges. Since the graph is
  // undirected, the edge from src to dest is also edge from dest
  // to src. Both are counted as 1 edge here.
  Edge * edge;
} Graph;
```

```
// Creates a graph with V vertices and E edges
Graph * createGraph(int V, int E)
{
  Graph * graph = (Graph *) malloc( sizeof(Graph) );
  graph->V = V;
  graph->E = E;

  graph->edge = (Edge *) malloc( graph->E * sizeof(Edge) );

  return graph;
}

// A utility function to find the subset of an element i
int Find(int parent[], int i)
{
  if (parent[i] == -1)
    return i;
  return Find(parent, parent[i]);
}

// A utility function to do union of two subsets
void Union(int parent[], int x, int y)
{
  int xset = Find(parent, x);
  int yset = Find(parent, y);
  parent[xset] = yset;
}

// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void * a, const void * b)
{
  Edge * a1 = (Edge *)a;
  Edge * b1 = (Edge *)b;
  return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's algorithm
void MST_Kruskal(Graph * graph)
{
  int V = graph->V;
  Edge result[V];  // Tnis will store the resultant MST
  int e = 0;  // An index variable, used for result[]
  int i = 0;  // An index variable, used for sorted edges
  // Allocate memory for creating V subsets
  int * parent = (int *) malloc( graph->V * sizeof(int) );

  // Initialize all subsets as single element sets
  memset(parent, -1, sizeof(int) * graph->V);

  // Step 1:  Sort all the edges in non-decreasing order of their weight
  // If we are not allowed to change the given graph, we can create a copy of
  // array of edges
  qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);


  // Number of edges to be taken is equal to V-1
  while (e < V - 1)
    {
      // Step 2: Pick the smallest edge. And increment the index
      // for next iteration
      Edge next_edge = graph->edge[i++];
```

```
      int x = Find(parent, next_edge.src);
      int y = Find(parent, next_edge.dest);

      // If including this edge does't cause cycle, include it
      // in result and increment the index of result for next edge
      if (x != y)
        {
          result[e++] = next_edge;
          Union(parent, x, y);
        }
      // Else discard the next_edge
    }

  // print the contents of result[] to display the built MST
  printf("Following␣are␣the␣edges␣in␣the␣constructed␣MST\n");
  for (i = 0; i < e; ++i)
    printf("%d␣--␣%d␣==␣%d\n", result[i].src, result[i].dest,
       result[i].weight);
  return;
}

// Driver program to test above functions
int main(void)
{
  /* Let us create following weighted graph
          10
     0--------1
     |  \     |
    6|   5\   |15
     |      \ |
     2--------3
          4
  */
  int V = 4;  // Number of vertices in graph
  int E = 5;  // Number of edges in graph
  Graph * graph = createGraph(V, E);


  // add edge 0-1
  graph->edge[0].src = 0;
  graph->edge[0].dest = 1;
  graph->edge[0].weight = 10;

  // add edge 0-2
  graph->edge[1].src = 0;
  graph->edge[1].dest = 2;
  graph->edge[1].weight = 6;

  // add edge 0-3
  graph->edge[2].src = 0;
  graph->edge[2].dest = 3;
  graph->edge[2].weight = 5;

  // add edge 1-3
  graph->edge[3].src = 1;
  graph->edge[3].dest = 3;
  graph->edge[3].weight = 15;

  // add edge 2-3
  graph->edge[4].src = 2;
  graph->edge[4].dest = 3;
  graph->edge[4].weight = 4;
```

```
    MST_Kruskal(graph);

    return 0;
}
```

运行结果

```
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
```