第十三章 最小生成树

13.1 社交网络中的图

在社交关系网中，任何两个人都可能直接建立联系，但是不同的人之间直接联络的成本却并不相同，比如廉价的短消息，或是昂贵的越洋电话。假如在 n 个人组成的一个社交群体中，我们希望任何一个人发生的新鲜事都可以成功传达到其他所有人那里去，那么我们至少要保持 n-1 对人与人之间的直接联系，从而将这 n 个人都纳入同一个社交网络里。显然，我们希望一条消息传达到所有人的总成本最小。

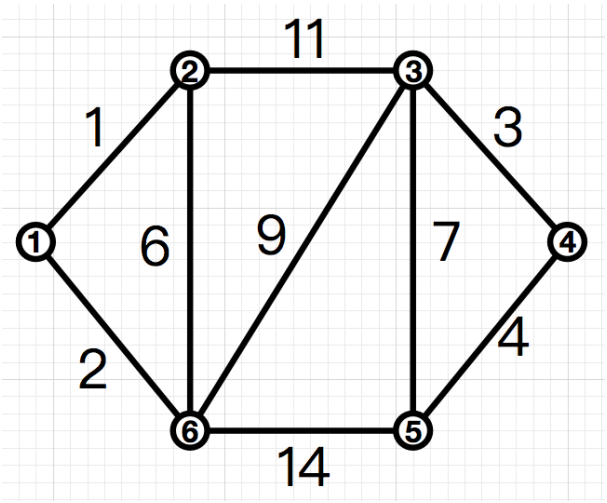


图 13.1. G=(V,E)

我们可以将这样一个社交关系网用连通的无向图（图 13.1）G=(V,E)来表示，V 中的每个顶点都是一个人，E 中的每条边代表某两个人之间可能建立的联系，而边的权重就是两人直接联络时的成本。我们的目标是寻找一个无环的子集$T \subseteq E$，它既可以将所有的顶点连起来（即可以把所有人加入到同一个关系网），又具有最小的权重和（即消息传遍这个关系网所需的总成本最小）。由于 E 的子集 T 连通了 G 中的所有顶点，又没有环结构，所以 T 是图 G 的一个生成树；又因为 T 所包含的所有边的总权重最小，所以 T 是图 G 的一个最小生成树。而寻找这样一个生成树的问题就是最小生成树问题。

13.2 最小生成树算法

接下来介绍两个解决最小生成树问题的方法：Prim 算法和 Kruskal 算法，均为使用贪

心策略的方法。

以图 13.1 为例，本章算法的输入文件为：

```
6 9
1 2 1
2 3 11
3 4 3
4 5 4
5 6 14
1 6 2
2 6 6
3 5 7
3 6 9
```

13.2.1 Prim 算法

首先，我们使用邻接矩阵作为存储图的数据结构。

```
#include <iostream>
using namespace std;

#define M 20
#define INF 99999;
int edge_map[M][M];//邻接矩阵
int dis[M][2];
int mark[M];
int path[M][2];
int vertex_count = 0;

int main()
{
    int vertex_num, edge_num;
    cin >> vertex_num >> edge_num;
```

```
    for (int i = 1; i <= vertex_num; i++)//初始化邻接矩阵
        for (int j = 1; j <= vertex_num; j++)
            if (i == j)
                edge_map[i][j] = 0;
            else
                edge_map[i][j] = INF;
    for (int i = 1; i <= edge_num; i++)//读入边
    {
        int start,end,weight;
        cin >> start >> end >> weight;
        edge_map[start][end] = weight;
        edge_map[end][start] = weight;//无向图
    }
    return 0;
}
```

代码 13.1. 构建邻接矩阵

Prim 算法可以从任意顶点开始构造最小生成树。算法首先建立两个顶点集合 X 和 Y，其中 X 包含 G 中任意一顶点，Y 包含其余所有顶点；然后找出具有最小权重的边(x, y)，其中 x∈X，y∈Y，把顶点 y 从 Y 移动至 X 并把边(x, y)加入到最小生成树中；重复此过程直到 Y 为空，此时即可得到图 G 的最小生成树。该算法的时间复杂度为 O(n^2)。

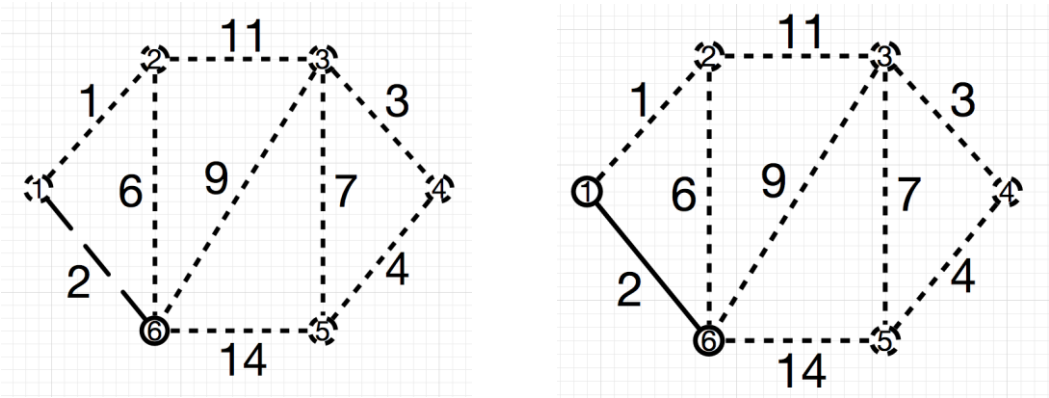


图 13.2.

以图 13.1 为例，首先选择顶点 6 作为初始顶点建立集合 X 和 Y，找出连接集合 X 与 Y 的最小权重边（1，6）（如图 13.2（a）），然后将顶点 1 移入集合 X 并将边（1，6）加入到最小生成树中（如图 13.2（b））。

接下来寻找连接集合 X 与 Y 的最小权重边（1，2）（如图 13.3（a）），将顶点 2 移入集合 X 并将边（1，2）加入到最小生成树中（如图 13.3（b））。

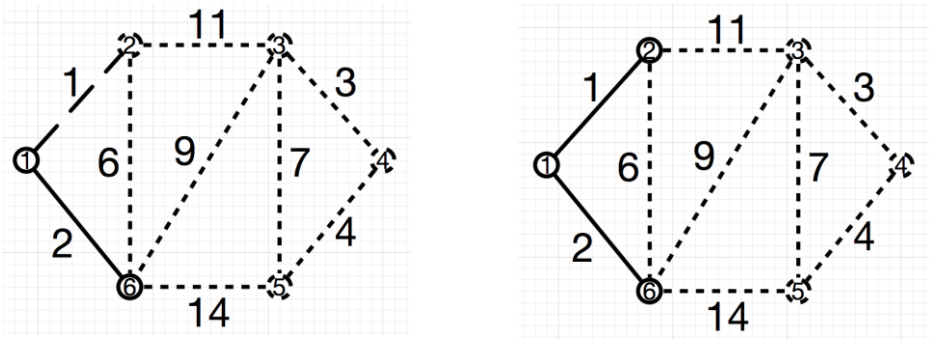


图 13.3.

接下来寻找连接集合 X 与 Y 的最小权重边（3，6）（如图 13.4（a）），将顶点 3 移入集合 X 并将边（3，6）加入到最小生成树中（如图 13.4（b））。

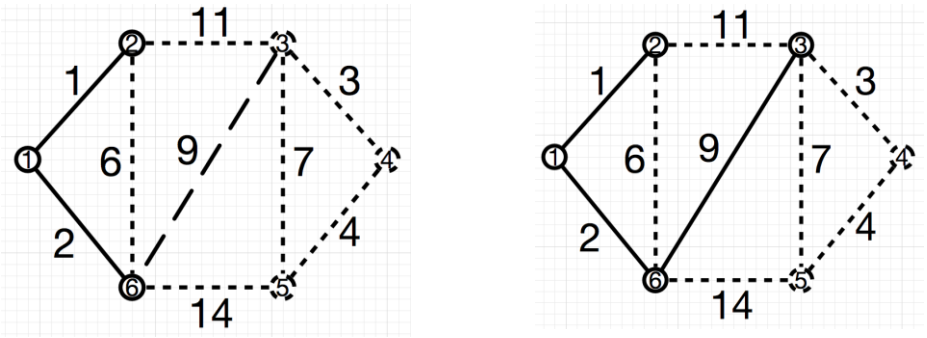


图 13.4.

接下来寻找连接集合 X 与 Y 的最小权重边（3，4）（如图 13.5（a）），将顶点 4 移入集合 X 并将边（3，4）加入到最小生成树中（如图 13.5（b））。

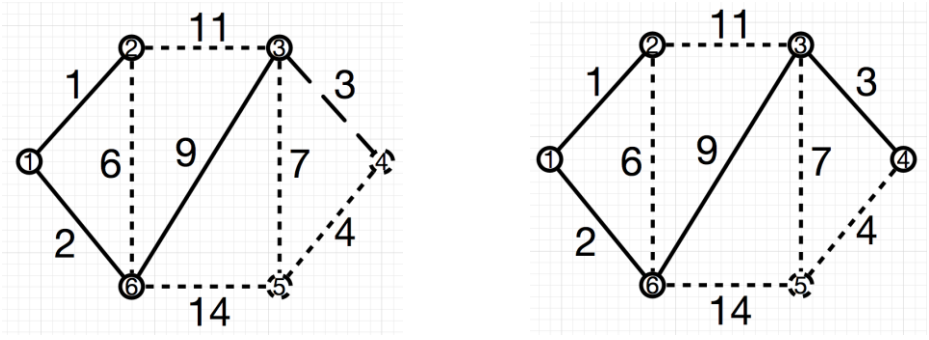


图 13.5.

最后寻找连接集合 X 与 Y 的最小权重边（4，5）（如图 13.6（a）），将顶点 5 移入集合 X 并将边（4，5）加入到最小生成树中，此时集合 Y 为空，图 13.6（b）中所示即为此图的最小生成树。

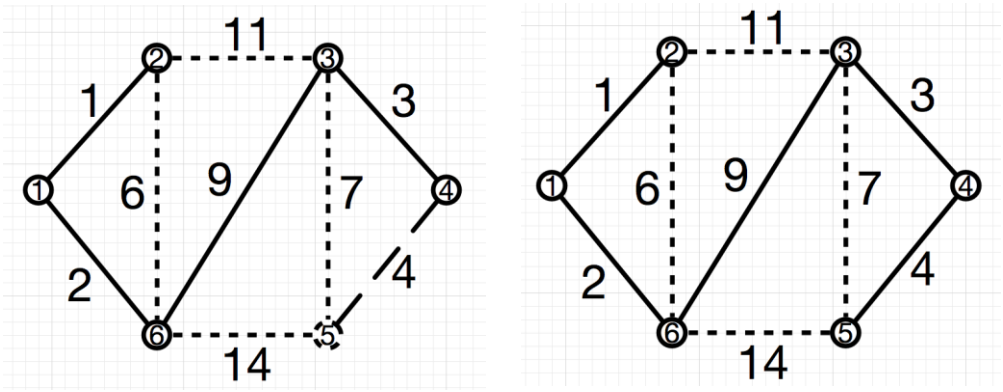


图 13.6.

下面给出 Prim 算法的代码实现。

```
for (int i = 1; i <= vertex_num; i++)
//初始化dis[],这里是一号顶点到其他顶点的距离
{
    dis[i][0]  = edge_map[1][i];
    dis[i][1]  = 1;
    mark[i] = 0;
}

mark[1] = 1;
vertex_count++;

while (vertex_count < vertex_num)
{
    int min = INF;
    int k = 0;
    for (int i = 1; i <= vertex_num; i++)
        if (mark[i] == 0 && min > dis[i][0])
        {
            min = dis[i][0];
```

```
20              k = i;
21          }
22      if (k == 0)
23          break;
24      mark[k] = 1;
25      path[vertex_count][0]=dis[k][1];
26      path[vertex_count][1]=k
27      vertex_count++;
28      for (int i = 1; i <= vertex_num; i++)
29          if (mark[i] == 0 && dis[i][0] > edge_map[k][i])
30          {
31              dis[i][0]=edge_map[k][i];
32              dis[i][1]=k;
33          }
34  }
35
36  for (int i = 1; i<vertex_num; i++)
37      cout<<path[i][0]<<"\" -> \""<< path[i][1] <<endl;
```

说明当前的边即为最短边

看完发现，prim算法与dijkstra算法基本一致
主要区别是prim算法每次只看直接联通边，
而dijkstra看的是从起点到某一条边的权重

代码 13.2. Prim 算法

13.2.2 Kruskal 算法

在本算法中我们使用邻接表作为存储图的数据结构。

```
1   #include <iostream>
2   #include <vector>
3   #include <algorithm>
4   #include <fstream>
5   using namespace std;
6
7   class Edge
8   {
9   public:
```

```
10      int u;
11      int v;
12      int w;
13      friend bool operator<(const Edge& E1,const Edge& E2)
14      {
15          return E1.w < E2.w;
16      }
17  };
18
19  vector<Edge> edges;//邻接矩阵
20
21  int main()
22  {
23      int n,m;
24      cin >> n >> m;
25      edges.assign(m,Edge());
26      for (int i = 0; i < m; i++)
27          cin >> edges[i].u >> edges[i].v >> edges[i].w;
28      sort(edges.begin(),edges.end());
29
30      Kruskal(edges,n);//实现详见下文
31      return 0;
32  }
```

起点，
终点，
边的权重

代码 13.3. 构建邻接表

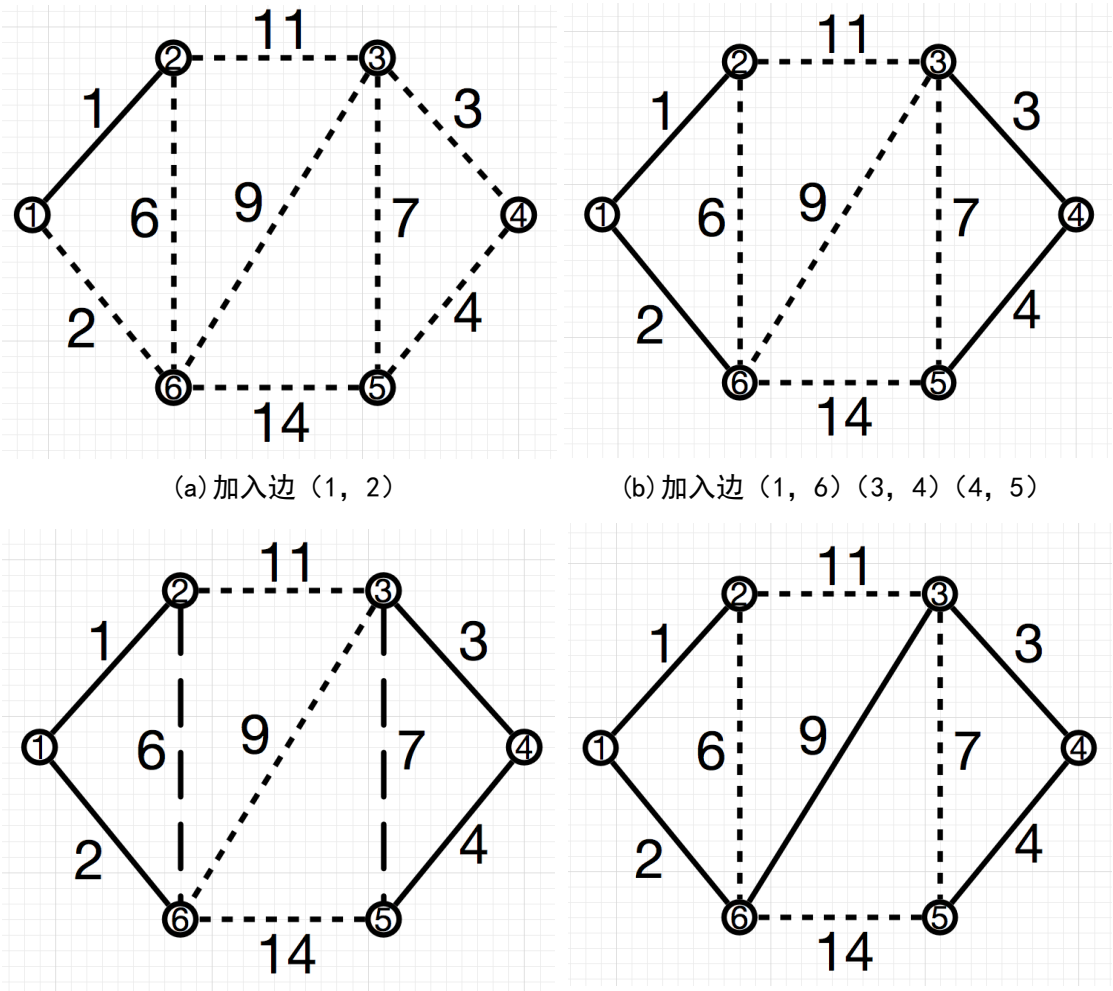
Kruskal 算法维护一个由许多生成树组成的森林，这些生成树逐渐合并直到产生唯一的一棵树为止。算法首先将所有边按权重以非降序排列，然后构建一个仅包含顶点不包含任何边的森林(V,T)，重复从排序表中取出权值最小的边，如果这条边加入 T 后不会在 T 中形成环，那么将此边加入 T，否则丢弃这条边。向 T 中加完 n-1 条边后算法结束，此时(V,T)即为图 G 的最小生成树。该算法的时间复杂度为 O(m log m)。

此算法可概括如下：

1. 将 G 的边按权重以非降序排列；
2. 对于排序表中每条边，如果将其加入 T 后不会形成环，则加入到 T，否则丢弃。

同样以图 13.1 为例，首先将权重最小的边（1，2）加入 T 中，随后是边（1，6），（3，

4)，(4，5)，接下来由于边（2，6）和（3，5）将会在 T 中构成环，故将其丢弃。最后将边（3，6）加入 T 中，由此得到该图的最小生成树（如图 13.7 所示）。



(a)加入边（1，2）　(b)加入边（1，6）(3，4)(4，5)

(c) 边（2，6）(3，5）将引入环　(d)加入边（3，6）

图 13.7.

下面给出 Kruskal 算法的代码实现。

```
1  //创建并查集
2  void MakeSet(vector<int>& uset,int n)
3  {
4      uset.assign(n,0);
5      for (int i = 0; i < n; i++)
6          uset[i] = i;
7  }
8  //查找当前元素所在集合的代表元
9  int FindSet(vector<int>& uset,int u)
```

```
10  {
11      int i = u;
12      while (uset[i] != i) i = uset[i]; (进行邻接表的节点的跳转，找到对应节点)
13      return i;
14  }
15  void Kruskal(const vector<Edge>& edges,int n)
16  {
17      vector<int> uset;
18      vector<Edge> SpanTree;
19      int Cost = 0,e1,e2;
20      MakeSet(uset,n);
21      for (int i = 0; i < edges.size(); i++)
22      {
23      e1 = FindSet(uset,edges[i].u);
24      e2 = FindSet(uset,edges[i].v);
25      if (e1 != e2)  (不考虑自环的情况，同时e1与e2是否相等的判断实际上就是在看两条边的起点与终点是否重合，
                       即是否成环，若成环则直接跳过)
26          {                                          这段是直接省略了对edges进行排序的算法
27              SpanTree.push_back(edges[i]);
28              Cost += edges[i].w;
29              uset[e1] = e2;
30          }
31      }
32      cout << "Result:\n";
33      cout << "Cost: " << Cost << endl;
34      cout << "Edges:\n";
35      for (int j = 0; j < SpanTree.size(); j++)
36          cout << SpanTree[j].u << " " << SpanTree[j].v << " " <<
37  SpanTree[j].w << endl;
38      cout << endl;
39  }
```

代码 13.4. Kruskal 算法

最小生成树结果的动态维护

现实生活中的社交关系网通常具有动态特征，如两人建立或解除好友关系对应着边的增减、两人之间的通信成本变化对应着边权重的改变等，因此维护动态图的最小生成树结果成为了需要考虑的问题。

一般情况下需要使用例如 B 树等高级索引结构对图中的边和点信息建立索引，进而完成最小生成树结果的动态维护算法。对于具有 n 个顶点的图，现有的研究成果表明基于索引的维护算法可在$O(n^{1/3}\log n)$的时间内完成维护工作。

13.3 并行最小生成树算法

在互联网时代，真实的社交关系组成的网络往往具有较大的规模，与此同时对计算的时效性却又有更高的要求。在求解最小生成树问题时，传统的算法已无法应付如此庞大的数据量，基于多（核）处理器的并行最小生成树算法成为提升效率的关键。

并行最小生成树算法的核心思想是首先通过对图的划分，将原图分为若干不相交的分区，交由不同的进程或线程处理。接下来各进程在分配给自己的分区图上完成算法的并行部分，将计算结果输出给一个全局进程，全局进程在此基础上进一步完成整个最小生成树算法的实现。本节中，我们依然使用邻接表作为图的存储数据结构（输入文件同上）。

```
1   class edge
2   {
3   public:
4       edge() {}
5       edge(int a,int b,int c):v1(a),v2(b),w(c),next(NULL) {}
6       int v1,v2;
7       int w,tid;
8       edge* next;
9   };
10  class vertex
11  {
12  public:
13      vertex(int v):vid(v),next(NULL) {}
14      int vid;
```

```
15      edge* next;
16  };
17  void construct()
18  {
19      int v_num,a_num;
20      cin >> v_num >> a_num;
21      for (int i = 1; i <= v_num; i++)
22          graph.push_back(vertex(i));
23      for (int i = 0; i < a_num; i++)
24      {
25          edge *temp1 = new edge(),*temp2;
26          cin >> temp1->v1 >> temp1->v2 >> temp1->w;
27          temp2 = new edge(*temp1);
28          temp1->next = graph[temp1->v1 - 1].next;
29          graph[temp1->v1 - 1].next = temp1;
30          temp2->next = graph[temp2->v2 - 1].next;
31          graph[temp2->v2 - 1].next = temp2;
32      }
33  }
```

代码 13.5. 构建邻接表

13.3.1 图的划分

对于图 G(V, E)，它的一组划分指集合$S = \{P_1, P_2, \ldots, P_n\}$，其中$\forall i \in [1, n]$，分区$P_i$均为图 G 的子图。同时，对于$\forall v \in V$，必有$v \in P_i$，且对$\forall j(1 \le j \le n \wedge j \ne i)$，$v \notin P_j$。 要做到不重不漏

划分图的策略较多，在求解最小生成树的算法中，为了尽量使各进程的负载均衡，计算量相当，我们建议采取基于哈希函数等均匀的划分方式。

```
1   void partition()
2   {
3       vector<vertex> temp[thread_num];
4       for (int i = 0; i < graph.size(); i++)
5           temp[i % thread_num].push_back(graph[i]);
```

根据thread_num进行均分，这里这种方式不错

```
    for (int i = 0; i < thread_num; i++)
        //建立子线程，实现详见下文
        subthreads.push_back(thread(subthread_func,temp[i],i));
}
```

代码 13.6. 图的划分

13.3.2 并行最小生成树算法

接下来将分别以边和顶点为视角介绍两种不同的并行最小生成树算法。

基于边的并行最小生成树算法

基于边的并行最小生成树算法也可称作并行 Kruskal 算法，其主体思想与非并行化的 Kruskal 算法大致相同。整个算法分为由各并行进程完成的“部分算法”与由一个全局进程完成的“仲裁算法”。

在“部分算法”中，当各进程收到来自全局进程的消息后选出本分区当中具有最小权重的边并发送给全局进程，直到本分区没有待处理的边或收到全局进程的结束通知(当添加的边数量达到n-1时)时结束进程。

```
void send_edge(multimap<int,edge>& m)
{
    if (!m.empty())
    {
        edge_queue.insert(*(m.begin()));
        m.erase(m.begin());
    }
}

//线程函数，执行“部分算法”
void subthread_func(vector<vertex> v,int tid)
{
    multimap<int,edge> e;

    for (int i = 0; i < v.size(); i++)
```

send_edge作用：将本子线程选择的边发给主线程(m即为子线程中记录边的数据结构)，主线程插入边，子线程删除对应的边

```
16      {
17          edge* temp = v[i].next;
18          while (temp != NULL)
19          {
20              temp->tid = tid;
21              e.insert(pair<int,edge>(temp->w,*temp));
22              temp = temp->next;
23          }
24      }
25
26      unique_lock<mutex> lk(mut);   此处加锁的作用：前面的sen_edge函数可以看到，
                                      send的时候会操作一个共享全局变量edge_queue。
27      send_edge(e);
28       这里在操作之前已经将边的权值排好序了，所以发送之后取begin的第一个就是当前最小的
29      while (true)
30      {
31          cond_v[tid].wait(lk);   当前线程的条件变量处于wait状态等待持有lock的线程的唤醒，
                                    就是唤醒之后会重新上锁，相当于当前线程即为持有lock那个thread
32          if (isfinished)
33              return;
34          send_edge(e);
35      }
36  }
```

代码 13.7. 子进程“部分算法”

在“仲裁算法”中，全局进程首先向所有并行进程发送消息获取各分区最小权重边构成队列 Q。接下来循环取出 Q 中权值最小的边 e，并向提供边 e 的进程发送消息请求补充新的最小权重边至 Q 中。如果取出的边 e 加入到结果集 T 中不会构成环路则保留此边，若会构成环路则将其丢弃。当 T 中的边的数量为|V|-1 或队列 Q 为空时算法结束，同时通知各进程结束算法。

```
1   void add_edge(edge e,map<int,int>& i,map<int,vector<int>>& rev_i)
2   {                                          rev_i存储的是从1开始的下标以及
                                               每个数字代表的边(的端点的)集合
3       mst.push_back(e);  这里使用了共享变量
4       int cid1 = i[e.v1],cid2 = i[e.v2];
5       if (cid1 == -1 && cid2 == -1)
                                   一条完全新边，起点和终点都不存在于index数组中
```

```
6       {
7           int cid = rev_i.size() + 1;
8           vector<int> temp;
9           temp.push_back(e.v1);
10          temp.push_back(e.v2);
11          rev_i[cid] = temp;
12          i[e.v1] = cid;
13          i[e.v2] = cid;
14      }
15      else if (cid1 == -1)
16      {
                                    添加新的起点
17          i[e.v1] = cid2;
18          rev_i[cid2].push_back(e.v1);
19      }
20      else if (cid2 == -1)
21      {
22          i[e.v2] = cid1;         添加新的终点
23          rev_i[cid1].push_back(e.v2);
24      }
25      else if (rev_i[cid1].size() <= rev_i[cid2].size())   起点终点都已经存在于邻接表中，
                                                             直接建立新的连接
26      {
27          for (int k = 0; k < rev_i[cid1].size(); k++)
28          {
29              i[rev_i[cid1][k]] = cid2;
30              rev_i[cid2].push_back(rev_i[cid1][k]);
31          }
32          rev_i.erase(cid1);
33      }
34      else
35      {
36          for (int k = 0; k < rev_i[cid2].size(); k++)
37          {
38              i[rev_i[cid2][k]] = cid1;
```

```
39              rev_i[cid1].push_back(rev_i[cid2][k]);
40          }
41          rev_i.erase(cid2);
42      }
43  }
44
45  void kruskal()
46  {
47      map<int,int> index;
48      map<int,vector<int>> rev_index;
49      for (int i = 0; i < graph.size(); i++)
50      {
51          index[i + 1] = -1;
52      }
53      while (mst.size() < graph.size() - 1)
54      {
55          unique_lock<mutex> lk(mut);
56          if (edge_queue.empty())
57              break;
58          pair<int,edge> temp = *(edge_queue.begin());
59          edge_queue.erase(edge_queue.begin());
60          lk.unlock();
61          cond_v[temp.second.tid].notify_all();
62          if (index[temp.second.v1] != index[temp.second.v2] ||
63  index[temp.second.v1] == -1)
64              add_edge(temp.second,index,rev_index);
65      }
66
67      isfinished = true;
68
69      for (int i = 0; i < thread_num; i++)
70      {
71          cond_v[i].notify_all();
```

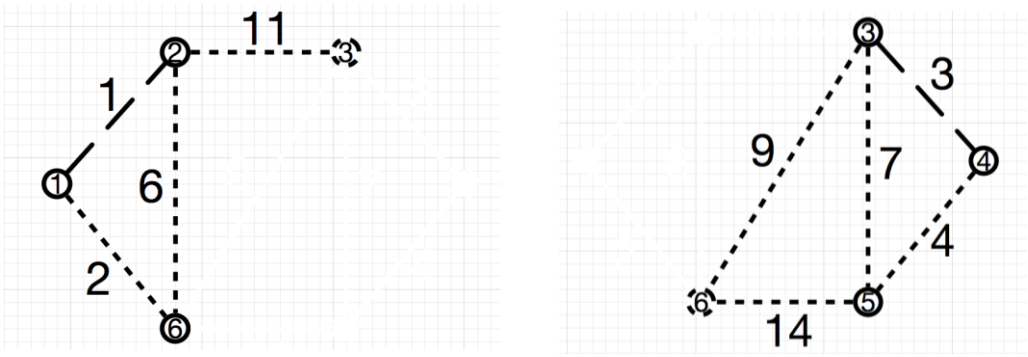
邻接表

已经取出来一条新的最短边，则通知所有子线程继续执行其各自的程序。

```
72            subthreads[i].join();
73        }
74    }
```

代码 13.8. 主进程“仲裁算法”

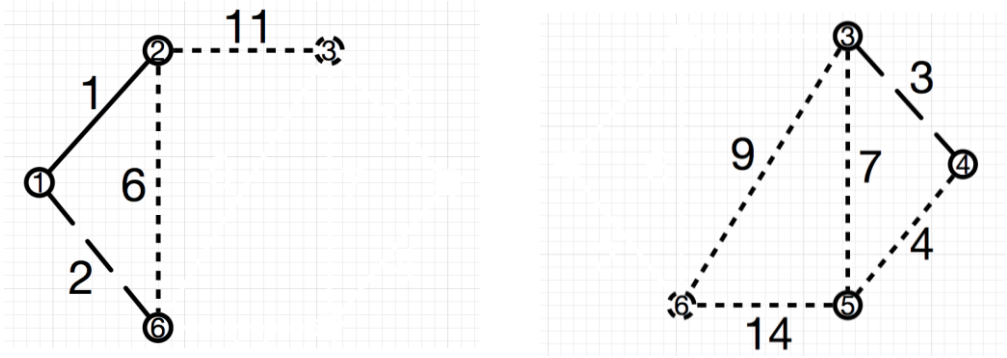
在本例中，我们使用 2 个并发线程执行“部分算法”，分别处理图 13.1 划分出的两个分区，并将各自的最小权重边提交至队列 Q，如图 13.8 所示。



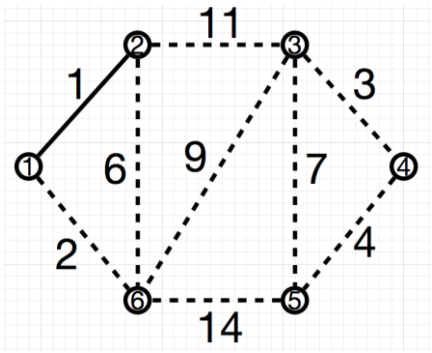
(a) 分区 1　　(b) 分区 2

图 13.8.

随后全局进程从队列 Q 中取出权值最小的边（1，2）加入 T 中，并通知边（1，2）的发送者进程 1 补充新的最小权重边到队列 Q 中，如图 13.9 所示。



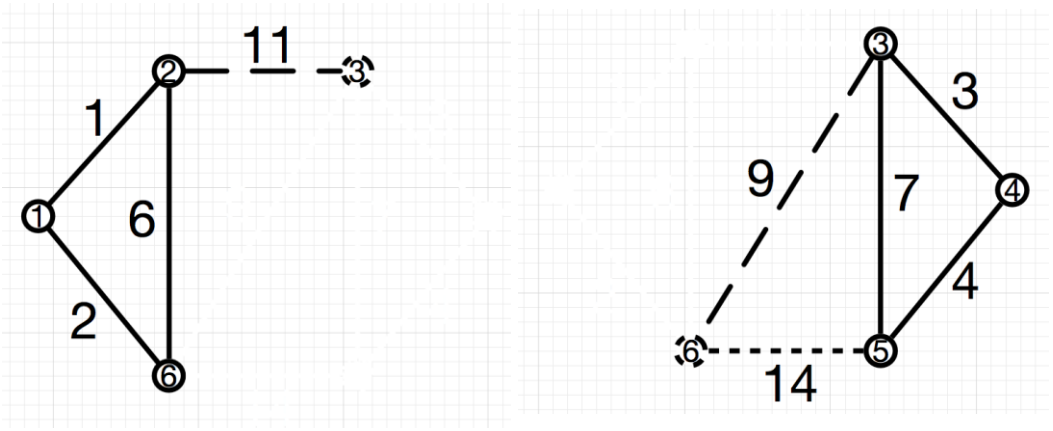
(a) 进程 1　　(b) 进程 2



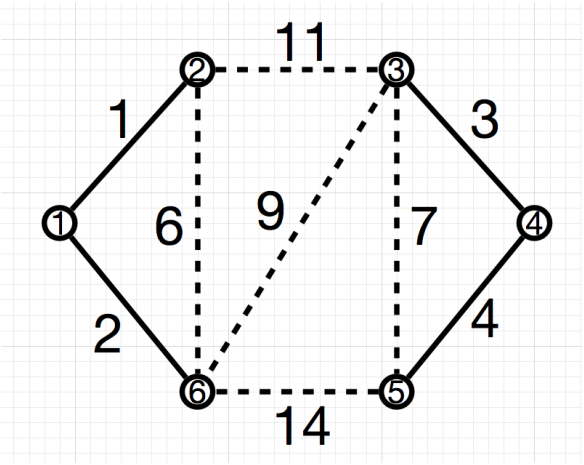
(c) 全局进程

图 13.9.

然后分别是边（1，6），（3，4），（4，5），接下来由于边（2，6）和（3，5）将会在 T 中构成环，故将其丢弃，如图 13.10。



(a) 进程 1　　(b) 进程 2



(c) 全局进程

图 13.10.

最后将边（3，6）加入 T 中，得到该图的最小生成树，如图 13.11 所示。

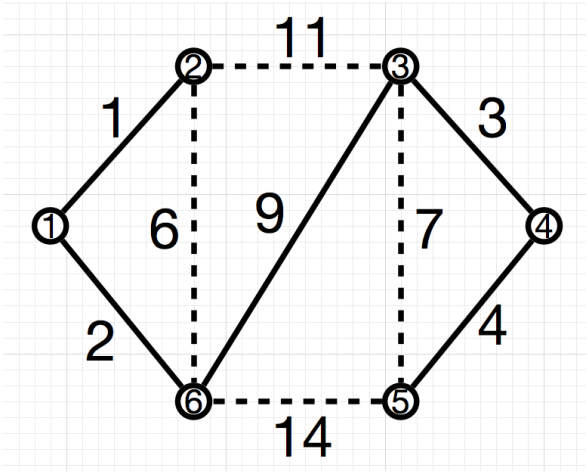


图 13.11.

以下是使用 3 个并发线程的 main 函数代码。

```cpp
#include<iostream>
#include<vector>
#include<thread>
#include<map>
#include<mutex>
#include<condition_variable>
using namespace std;

const int thread_num = 3;//线程数量
vector<vertex> graph;//图邻接表
vector<edge> mst;//结果集
vector<thread> subthreads;//线程向量
multimap<int,edge> edge_queue;//全局队列
mutex mut;//互斥锁
condition_variable cond_v[thread_num];//条件变量
bool isfinished = false;

void output()
{
    for (int i = 0; i < mst.size(); i++)
    cout << mst[i].v1 << " -> " << mst[i].v2 << endl;
}

int main()
{
    construct();//构造图的邻接表
    partition();//图的划分，创建并发进程，执行"部分算法"
    kruskal();//全局进程执行"仲裁算法"
    output();//结果输出
    return 0;
}
```

代码 13.9. main 函数代码

基于顶点的并行最小生成树算法

首先介绍一个非并行化的最小生成树算法，Boruvka 算法。

算法首先构造一个结果集 T=(V,S)，其中 S 为空集，即 T 是一个仅包含 G 中所有顶点不包含任何边的图，显然，此时图 T 中每一个顶点都属于一个独立的连通分量；然后对于 T 中每个连通分量 C，找出具有最小权重的边(x, y)，其中 x∈C，y∉C，此边即为 C 的最小邻接权重边，将所有连通分量的最小邻接权重边并入集合 S 中；接下来重复此过程直到图 T 中只存在一个连通分量为止，此时的结果集 T 即为图 G 的最小生成树。

基于顶点的并行最小生成树算法由 Boruvka 算法及前文介绍的 Prim 算法拓展而来。算法分为由各并行进程完成的“分区 Prim 算法”以及由一个全局进程完成的“全局 Boruvka 算法”。

在“分区 Prim 算法”中，假设为当前进程分配的分区为 P=(V`,E`)，初始情况下认为 V`中所有顶点均不属于任何连通分量，并构建空结果集 S`。首先，随机选取 V`中不属于任何连通分量的顶点 v，为其创建连通分量 $C_v$，并以 $C_v$ 为起点在全图 G 上执行一步 Prim 算法，找出 G 中 $C_v$ 的最小邻接权重边（x, u），即 x∈$C_v$，u∉$C_v$；如果 u 在本分区 P 内且 u 尚不属于任何连通分量，则将边(x, u)并入结果集 S`中，将 u 加入 C_v 中，并在 C_v 上循环执行下一步 Prim 算法重复以上过程；如果 u 在本分区 P 内但 u 已属于其他连通分量 C_u，则将边(x, u)并入结果集 S`中，将 $C_v$ 与 $C_u$ 合并，并停止 Prim 算法的执行；如果 u 不在本分区 P 内，则直接停止 Prim 算法的执行。接下来，再次在 V`中随机选取不属于任何连通分量的顶点重复上述过程，直到 V`中所有顶点都属于某一连通分量为止。最后，得到的结果集 S`即为各个进程“分区 Prim 算法”的输出结果。

在“全局 Boruvka 算法”中，首先等待并行进程的“分区 Prim 算法”执行完毕，并将所有进程输出的结果集 S`合并得到集合 S，从而构造图 T=(V,S)，其中 V 为图 G 的所有顶点；然后以 T 作为上述非并行化 Boruvka 算法的初始结果集，执行 Boruvka 算法，直到算法结束；则算法最终产生的结果集 T 即为图 G 的最小生成树。

下面给出“分区 Prim 算法”的伪代码。

```
输入：P=(V`,E`)
输出：S'
V_0 = V', S' = ∅
while |V_0| > 0
    choose vertex v from V_0, construct component C_v = {v}
    V_0 = V_0 − {v}
    run Prim on C_v to find edge (x,u), where x ∈ C_v, u ∉ C_v
    while u ∈ P
        S' = S' + (x,u)
        if u is in another component C_u
            C_v = C_v ∪ C_u
            break
        else
            C_v = C_v + u
            V_0 = V_0 − {u}
        run Prim on C_v to find new edge (x,u), where x ∈ C_v, u ∉ C_v
return S'
```

对于主线程的boruvka算法，其进行算法时的每一个"点"，实际上本身就是子线程产生的连通分量，由于其相互独立性，故不需要像并行kruskal算法一样进行额外的对于边的合并操作。

13.4 练习题

利用本章所学知识，设计并构建一个网络通信线路维护系统。针对给出的网络节点坐标，规划出一个最优的通信线路铺设方案，要求能够覆盖网络中的所有节点，且总成本最低（所有线路长度总和最小）；还应能够对通信网络实现维护，即支持动态插入或删除网络节点后，快速更新原有的线路铺设方案；此外，当网络节点数量过多时，可以考虑使用并行化的实现方案，以获得更高的效率。

13.5 文献阅读

最小生成树算法是图论中的一个非常重要的基本算法，在许多有关图计算的理论中都有很重要的应用价值。由 Thomas H.Cormen 等编撰的经典著作《算法导论》，对这部分内容有详细的讲解[Thomas H.Cormen 2013]。

不少基于图的复杂场景算法，都是在本文提到的 Prim 或 Kruskal 算法上进行的改进。如 Ghaffari M 等提出的一种分布式的最小生成树算法[Ghaffari M 2017]，以及 Cheng C 等提出的针对动态拓扑图的最小生成树维护算法[Cheng C 1988]。