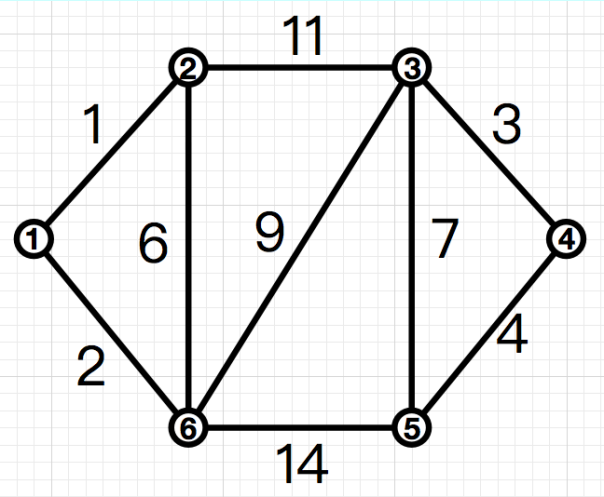
Parallel MST

软件学院《数据结构》讲义
内部使用

引例：社交网络中的图



最小生成树MST的三个性质：

- 最小生成树不能有回路。
- 最小生成树可能是一个，也可能是多个。
- 最小生成树边的个数等于顶点的个数减一。

- 一个社交关系网用连通的无向图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）。

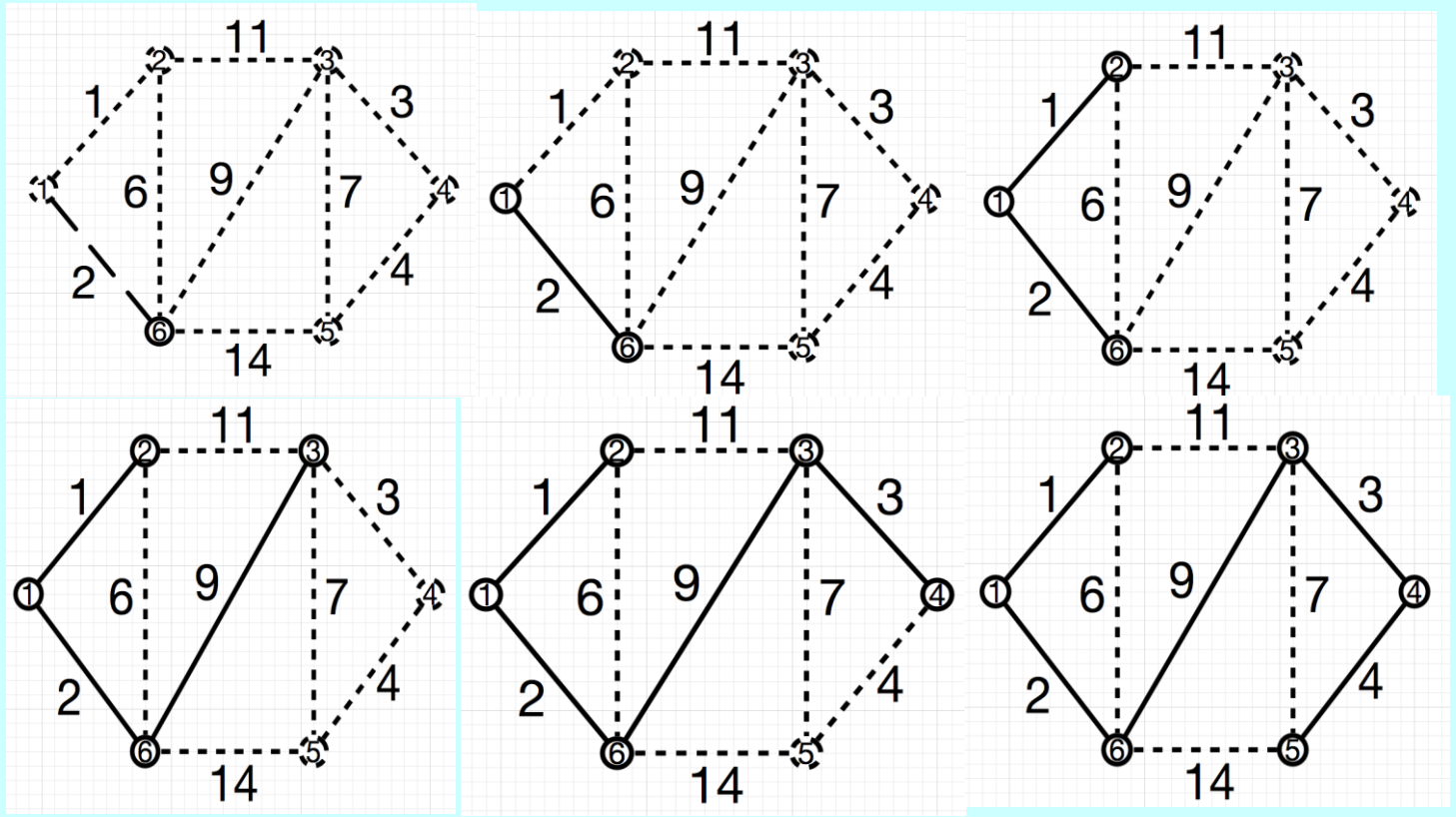
最小生成树

- 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
- 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。

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)。

最小生成树



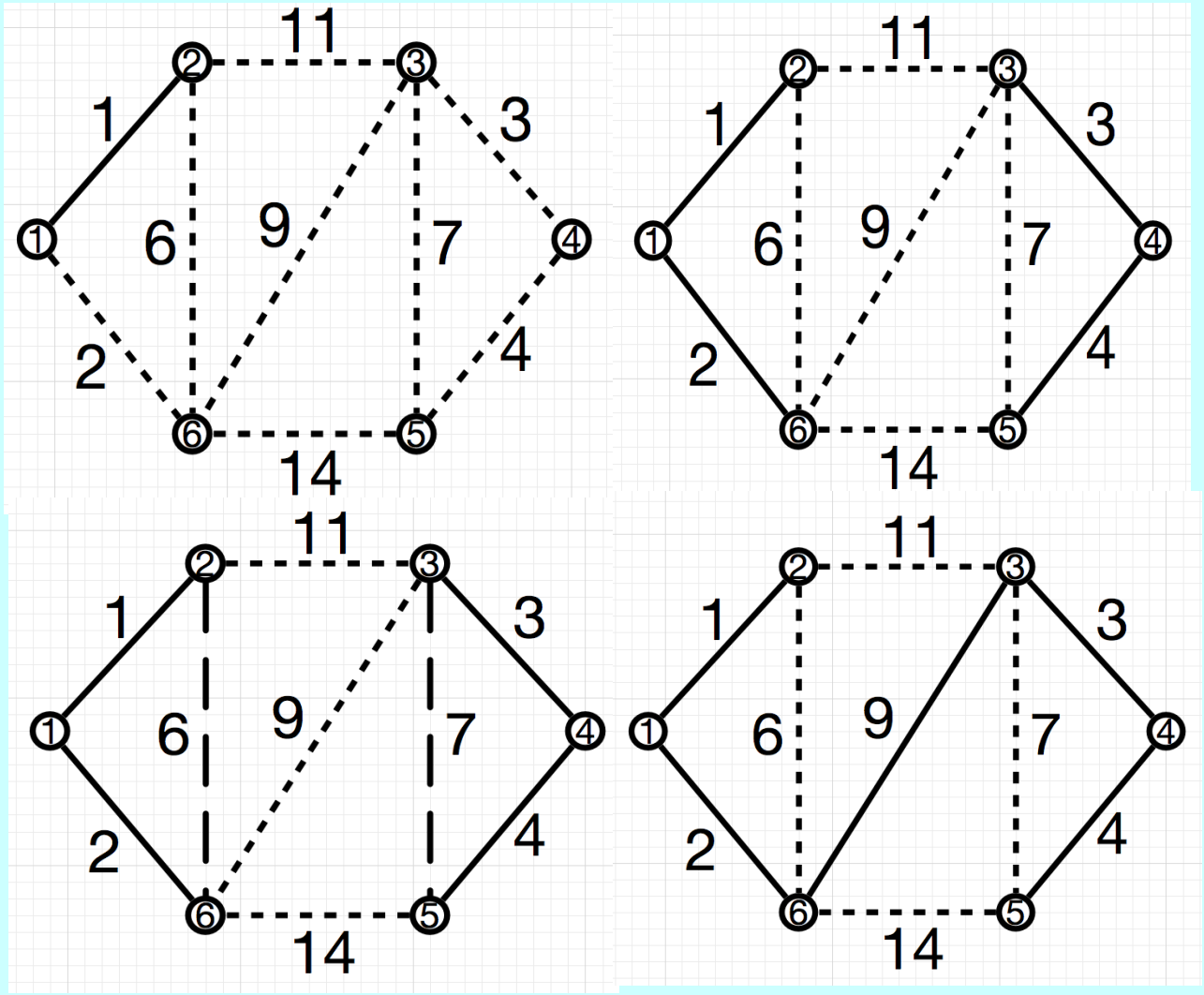
最小生成树-Kruskal算法

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。

最小生成树-Kruskal算法



最小生成树-Kruskal算法

```
//创建并查集
void MakeSet(vector<int>& uset,int n)
{
        uset.assign(n,0);
        for (int i = 0; i < n; i++)
                uset[i] = i;
}
//查找当前元素所在集合的代表元
int FindSet(vector<int>& uset,int u)
{
        int i = u;
        while (uset[i] != i) i = uset[i];
        return i;
}
```

uset为记录点的集合，意思是每个点有一个index，并将这个index存放在uset的对应的index上面

这里是在模拟邻接表，while做的就是进行邻接表的跳转，直到跳到对应的点为之。

```
void Kruskal(const vector<Edge>& edges,int n)
{
        vector<int> uset;
        vector<Edge> SpanTree;
        int Cost = 0,e1,e2;
        MakeSet(uset,n);
        for (int i = 0; i < edges.size(); i++)
        {
        e1 = FindSet(uset,edges[i].u);
        e2 = FindSet(uset,edges[i].v);
        if (e1 != e2)
                {
                        SpanTree.push_back(edges[i]);
                        Cost += edges[i].w;
                        uset[e1] = e2;
                }
        }
                }
```

条件变量

- std::mutex互斥量是多线程间同时访问某一共享变量时，保证变量可被安全访问的手段。
- 线程同步是指线程间需要按照预定的先后次序顺序进行的行为。
- 条件变量提供了两类操作：wait和notify。这两类操作构成了多线程同步的基础
- notify_one 唤醒等待的一个线程，注意只唤醒一个。
- notify_all 唤醒所有等待的线程。

```
std::mutex mutex;

std::condition_variable cv;


// 条件变量与临界区有关，用来获取和释放一个锁，因此通常会和mutex联用。

std::unique_lock lock(mutex);

// 此处会释放lock，然后在cv上等待，直到其它线程通过cv.notify_xxx来唤醒当前线程，cv被唤醒后会再次对lock进行上锁，然后wait函数才会返回。

// wait返回后可以安全的使用mutex保护的临界区内的数据。此时mutex仍为上锁状态

cv.wait(lock)


// 所有等待在cv变量上的线程都会被唤醒。但直到lock释放了mutex，被唤醒的线程才会从wait返回。

cv.notify_all(lock)
```

并行最小生成树算法

- 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。
- 使用邻接表作为图的存储数据结构

```
struct edge
{
        edge() {}
        edge(int a,int b,int c)：v1(a),v2(b),w(c),next(NULL) {}
        int v1,v2;
        int w,tid;
        edge* next;
};
struct vertex
{
        vertex(int v)：vid(v),next(NULL) {}
        int vid;
        edge* next;
};
```

并行最小生成树算法

```
void construct()
{
        int v_num,a_num;
        cin >> v_num >> a_num;
        for (int i = 1; i <= v_num; i++)
                graph.push_back(vertex(i));
        for (int i = 0; i < a_num; i++)
        {
                edge *temp1 = new edge(),*temp2;
                cin >> temp1->v1 >> temp1->v2 >> temp1->w;
                temp2 = new edge(*temp1);
                temp1->next = graph[temp1->v1 - 1].next;
                graph[temp1->v1 - 1].next = temp1;
                temp2->next = graph[temp2->v2 - 1].next;
                graph[temp2->v2 - 1].next = temp2;
        }
```

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$\forall j(1 \leq j \leq n \wedge j \neq i)$，$v \notin P_j$。

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。

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

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时结束进程。

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

基于边的并行最小生成树算法

```
//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++)
        {
                edge* temp = v[i].next;
                while (temp != NULL)
                {
                        temp->tid = tid;
                        e.insert(pair<int,edge>(temp->w,*temp));
                        temp = temp->next;
                }
        }

        unique_lock<mutex> lk(mut);//lock_guard只能保证在析构的时候执行解锁操作
        send_edge(e);
while (true)
        {
                cond_v[tid].wait(lk);
                //等待条件变量
                if (isfinished)
                        return;
                send_edge(e);
        }
}
```

“仲裁算法”

1. 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。
2. 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。
3. 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。
4. 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。

“仲裁算法”

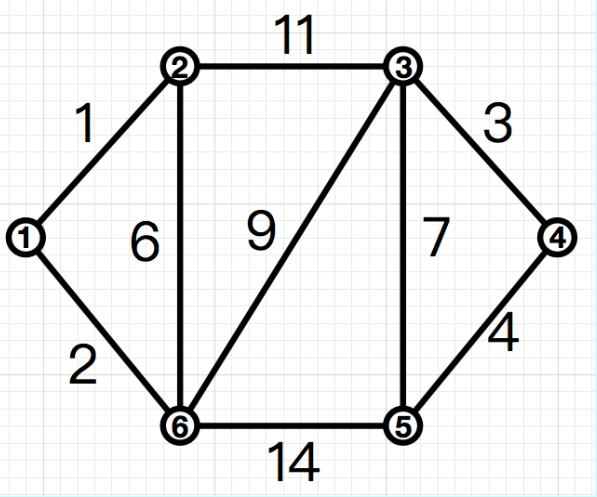
```
void add_edge(edge e,map<int,int>& i,map<int,vector<int>>& rev_i)
{
        mst.push_back(e);
        int cid1 = i[e.v1],cid2 = i[e.v2];
        if (cid1 == -1 && cid2 == -1)
        {
                int cid = rev_i.size() + 1;
                vector<int> temp;
                temp.push_back(e.v1);
                temp.push_back(e.v2);
                rev_i[cid] = temp;
                i[e.v1] = cid;
                i[e.v2] = cid;
        }
        else if (cid1 == -1)
        {
                i[e.v1] = cid2;
                rev_i[cid2].push_back(e.v1);
        }
        else if (cid2 == -1)
        {
                i[e.v2] = cid1;
                rev_i[cid1].push_back(e.v2);
        }
        else if (rev_i[cid1].size() <= rev_i[cid2].size())
        {
                for (int k = 0; k < rev_i[cid1].size(); k++)
                {
                        i[rev_i[cid1][k]] = cid2;
                        rev_i[cid2].push_back(rev_i[cid1][k]);
                }
                rev_i.erase(cid1);
        }
```

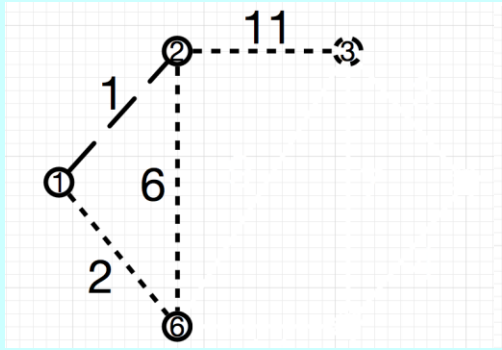
“仲裁算法”

```
        else
        {
                for (int k = 0; k < rev_i[cid2].size(); k++)
                {
                        i[rev_i[cid2][k]] = cid1;
                        rev_i[cid1].push_back(rev_i[cid2][k]);
                }
                rev_i.erase(cid2);
        }
}
```

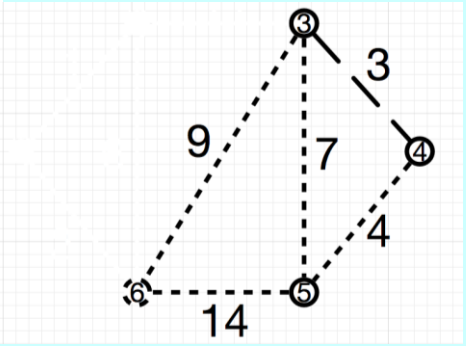
```
void kruskal()
{
        map<int,int> index; map<int,vector<int>> rev_index;
        for (int i = 0; i < graph.size(); i++)
                index[i + 1] = -1;


        while (mst.size() < graph.size() - 1)
        {
                unique_lock<mutex> lk(mut);
                if (edge_queue.empty())
                        break;
                pair<int,edge> temp = *(edge_queue.begin());
                edge_queue.erase(edge_queue.begin());
                lk.unlock();  //调用解锁接口，理由?
                cond_v[temp.second.tid].notify_all();//唤醒条件变量
                if (index[temp.second.v1] != index[temp.second.v2] || index[temp.second.v1] == -1)
                        add_edge(temp.second,index,rev_index);
        }
        isfinished = true;
        for (int i = 0; i < thread_num; i++)
        {
                cond_v[i].notify_all();//唤醒条件变量
                subthreads[i].join();
        }
}
```

```
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;
}
```

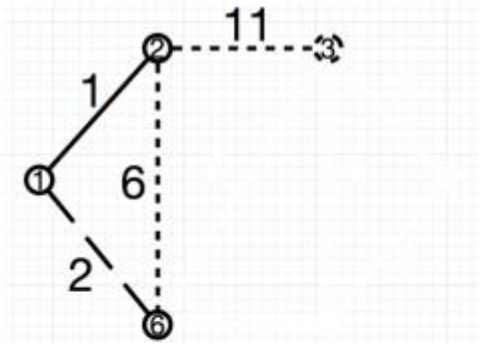




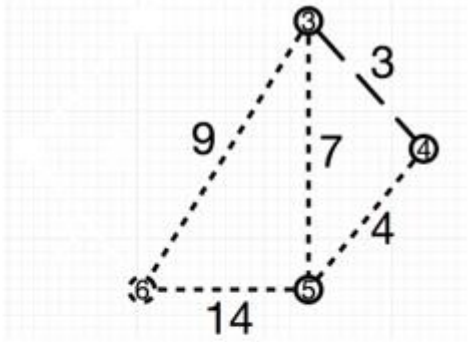
(a) 分区1



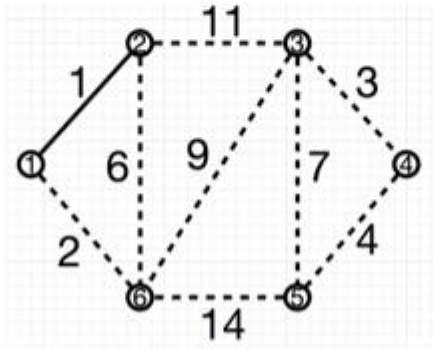
(b) 分区2



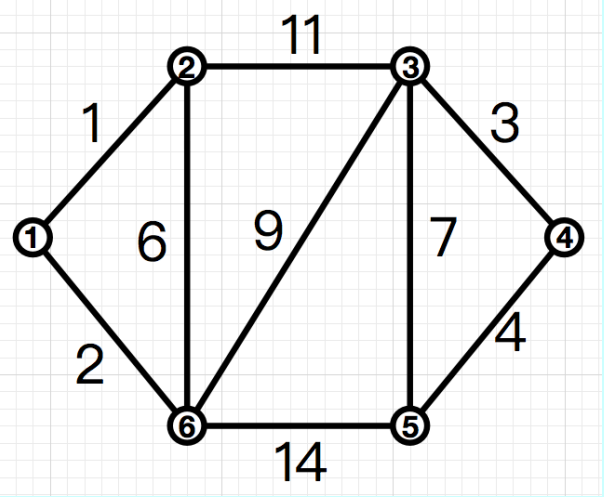
(a) 进程 1

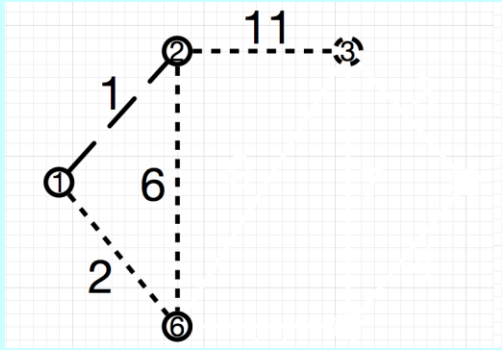


(b) 进程 2

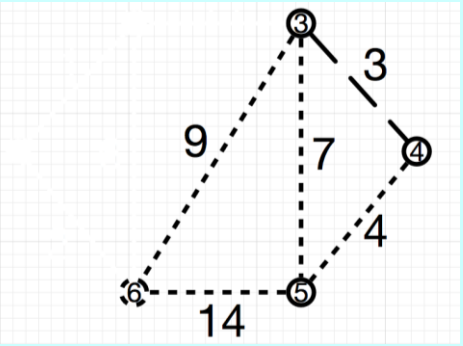


(c) 全局进程

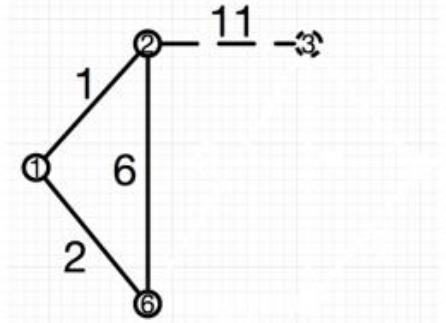




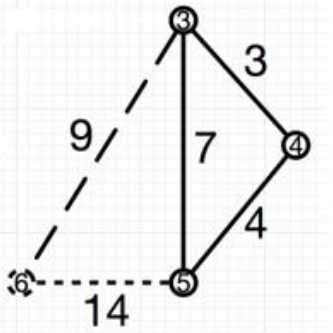
(a) 分区1



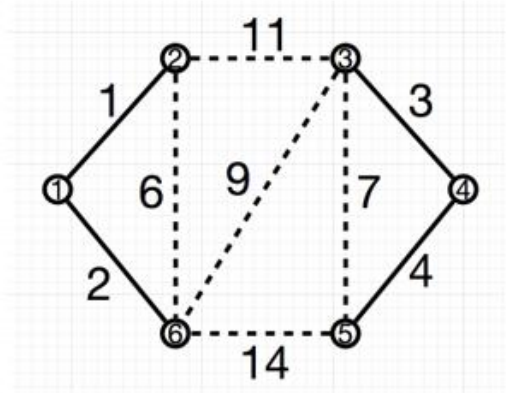
(b) 分区2



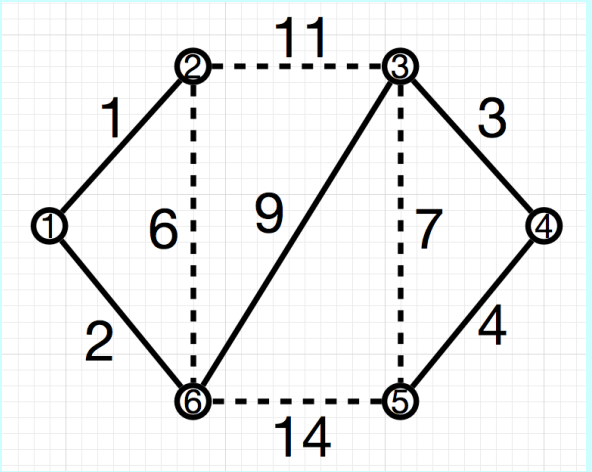
(a) 进程 1



(b) 进程 2



(c) 全局进程



基于顶点的并行最小生成树算法

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$x \in C$，$y \notin C$，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

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”。

“分区Prim算法”

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$x \in C_V$，$u \notin C_V$；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。

基于顶点的并行最小生成树算法

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”。

“分区Prim算法”

输入：P=(V`,E`)
输出：S’
$V_0 = V', S' = \emptyset$
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* $x \in C_v, u \notin C_v$
    while u ∈ P
        $S' = S' + (x,u)$
        if u is in another component C_u
            $C_v = C_v \cup C_u$
            break
        else
            $C_v = C_v + u$
            $V_0 = V_0 - \{u\}$
        run Prim on C_v *to find new edge* (x,u), *where* $x \in C_v, u \notin C_v$
return S′

基于顶点的并行最小生成树算法

“全局Boruvka算法”

- 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。

Next

- Pagerank 算法
- 数据结构讲义