数据结构与算法

DATA STRUCTURE

第二十五讲 最小生成树

胡浩栋

信息管理与工程学院

2018 - 2019 第一学期

课堂内容

- 最小生成树

最小生成树

Minimum Spanning Tree

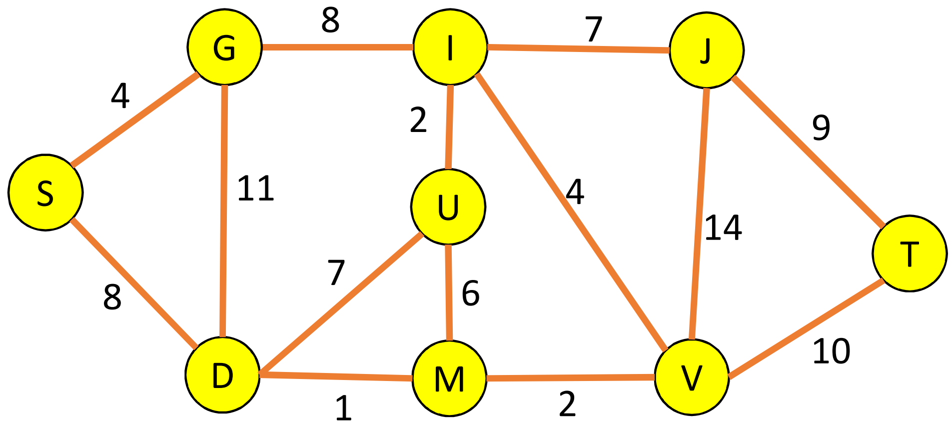
生成树

- 树是一个无向，无环的连通图
- 一个连通无向图上的生成树，是这个图的子图
  - 子图是树结构
  - 包含全部顶点集

生成树

- 定义生成树的cost就是树的所有边的权重和

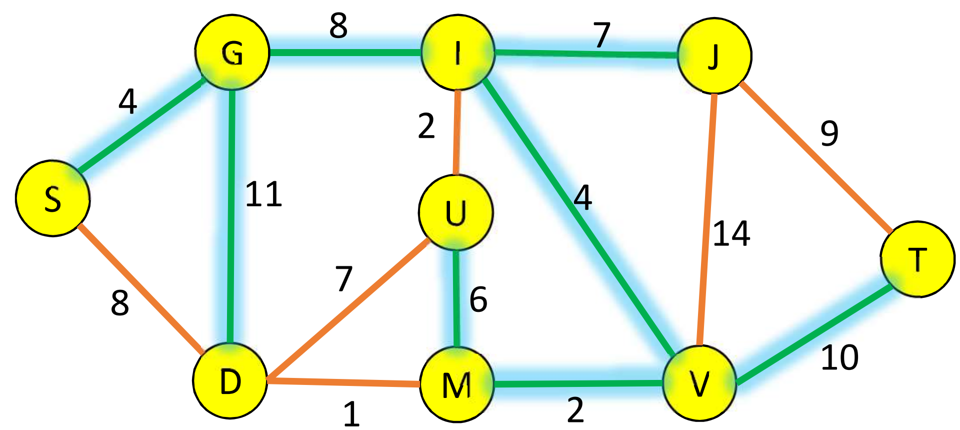
- 我们要找一个cost最小的生成树，也就是最小生成树MST



生成树

- Cost是

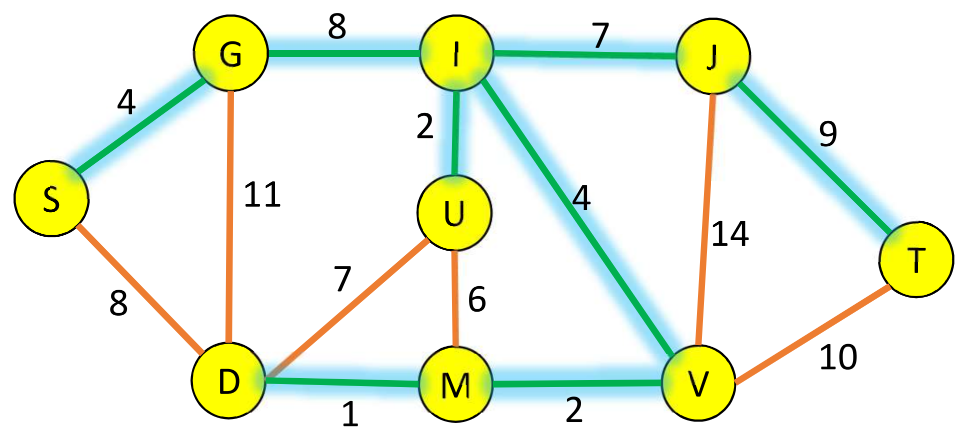
4+8+7+4+10+2+6+11=52



- Cost是

4+8+7+4+9+2+2+1=37

这个是MST

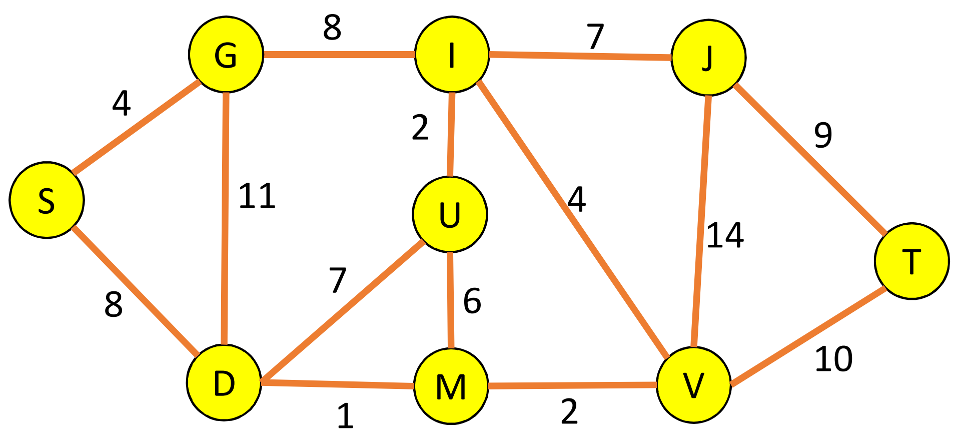


为什么要找MST

- 城市道路，公共设施设计
- 物流算法，TSP路径问题
- 别的图形算法

如何在算法里找MST

- Prim算法和Kruskal算法
- 贪婪算法
  - 如何每次选择一条最“好”的边
  - 使得之前的选择都包含在某个MST里面



Prim算法

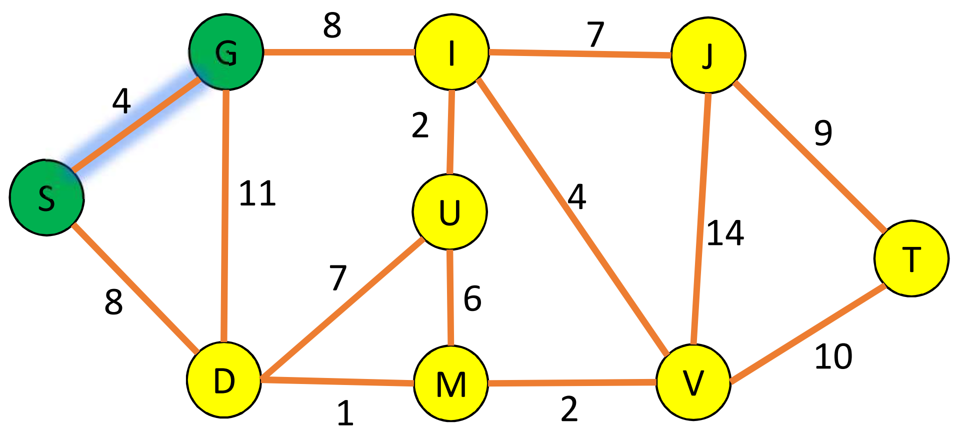
一个贪婪算法

Prim算法思路

- 第一步选一条最短的边
- 从这条边出发，维护一棵树，
  - 逐个添加边使得它成长为生长树
  - 添加的边必须是一个顶点在树里，一个顶点不在
  - 把符合条件的边里选一条最短的

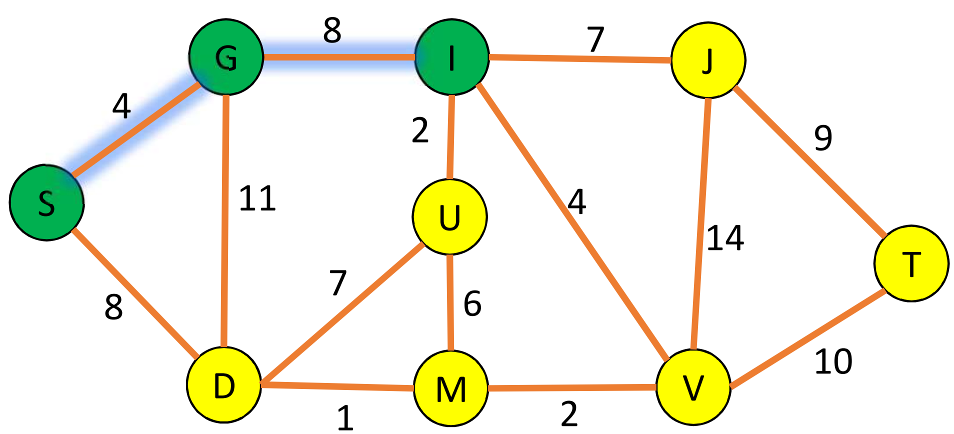
Prim算法例子

- 第一步选一条最短的边
- 从这条边出发，维护一棵树，
  - 逐个添加边使得它成长为生长树
  - 添加的边必须是一个顶点在树里，一个顶点不在
  - 把符合条件的边里选一条最短的



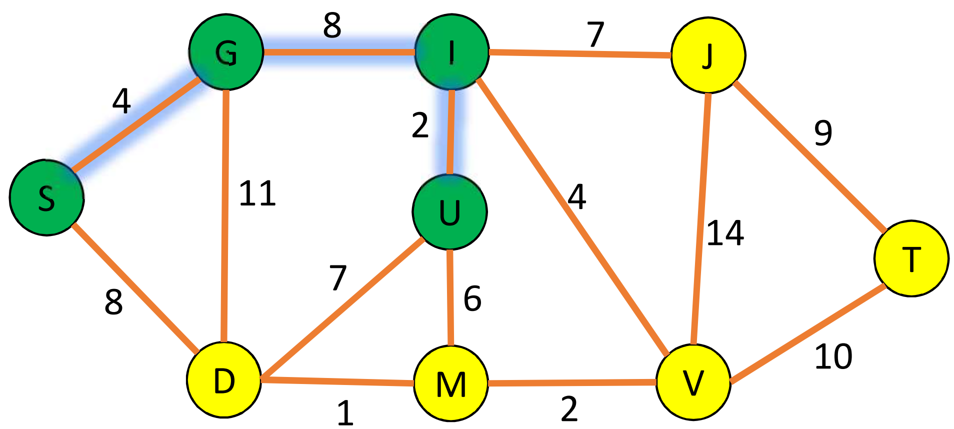
Prim算法例子

- 第一步选一条最短的边
- 从这条边出发，维护一棵树，
  - 逐个添加边使得它成长为生长树
  - 添加的边必须是一个顶点在树里，一个顶点不在
  - 把符合条件的边里选一条最短的



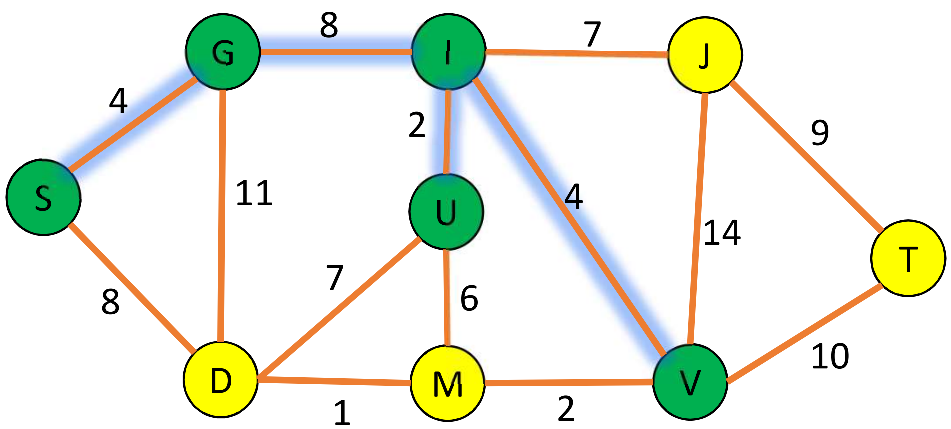
Prim算法例子

- 第一步选一条最短的边
- 从这条边出发，维护一棵树，
  - 逐个添加边使得它成长为生长树
  - 添加的边必须是一个顶点在树里，一个顶点不在
  - 把符合条件的边里选一条最短的



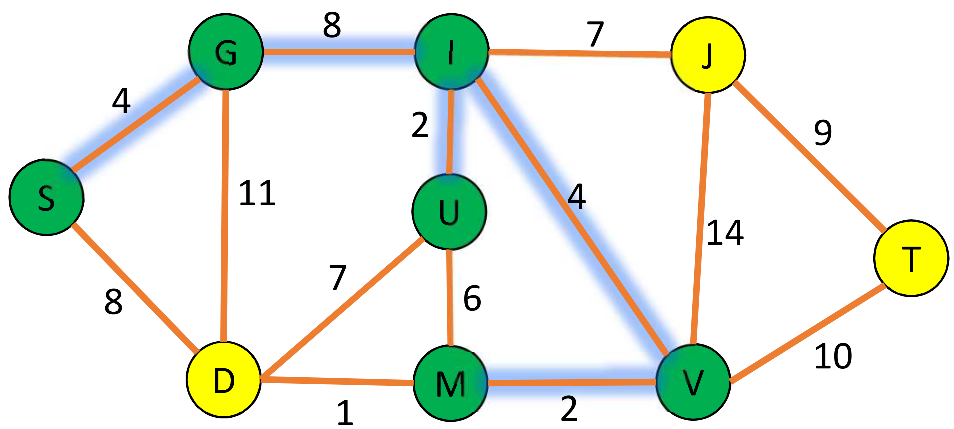
Prim算法例子

- 第一步选一条最短的边
- 从这条边出发，维护一棵树，
  - 逐个添加边使得它成长为生长树
  - 添加的边必须是一个顶点在树里，一个顶点不在
  - 把符合条件的边里选一条最短的



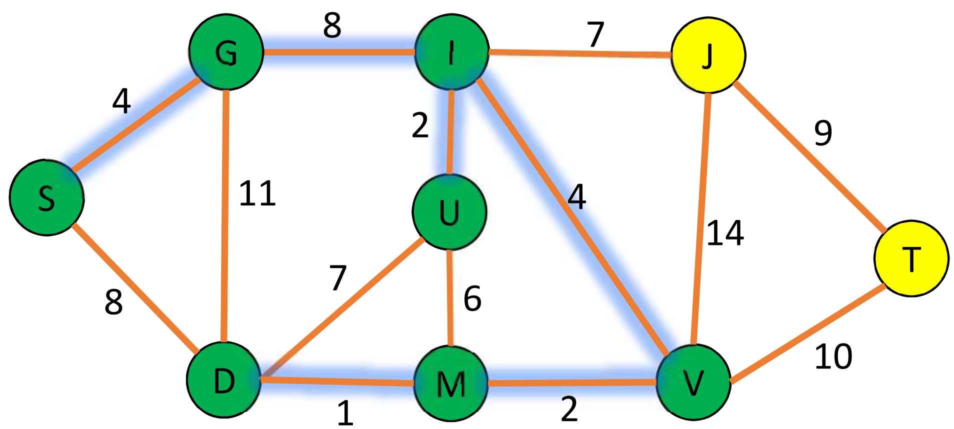
Prim算法例子

- 第一步选一条最短的边
- 从这条边出发，维护一棵树，
  - 逐个添加边使得它成长为生长树
  - 添加的边必须是一个顶点在树里，一个顶点不在
  - 把符合条件的边里选一条最短的



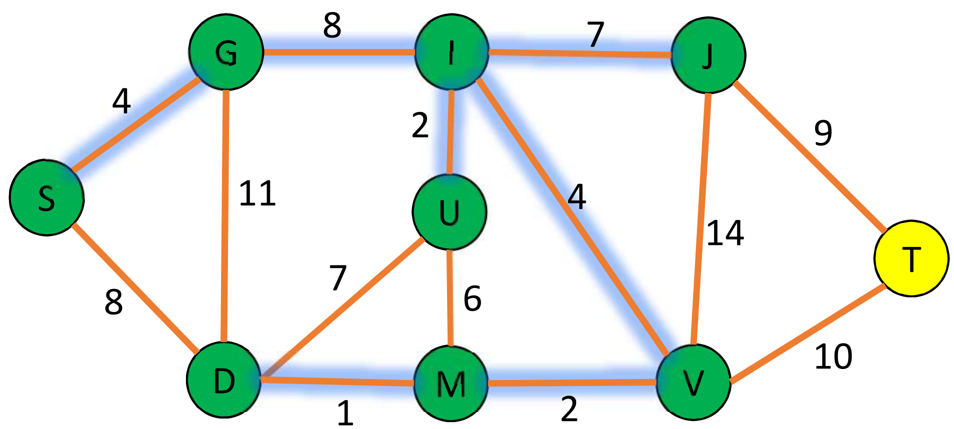
Prim算法例子

- 第一步选一条最短的边
- 从这条边出发，维护一棵树，
  - 逐个添加边使得它成长为生长树
  - 添加的边必须是一个顶点在树里，一个顶点不在
  - 把符合条件的边里选一条最短的



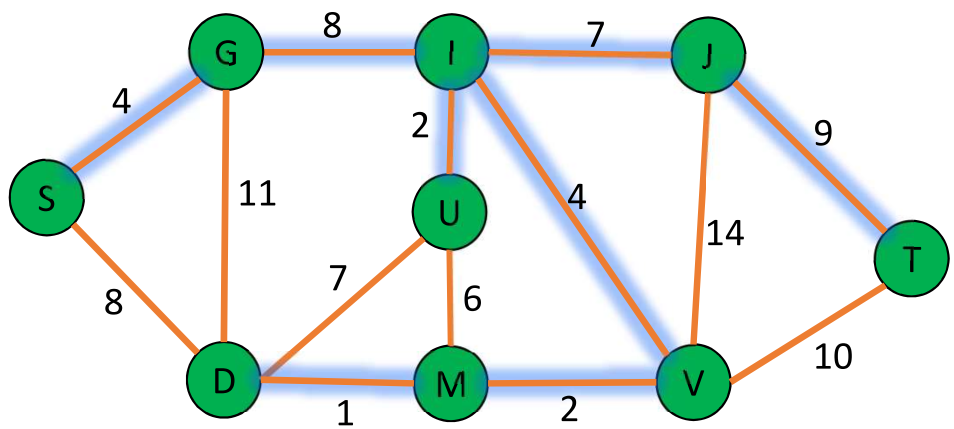
Prim算法例子

- 第一步选一条最短的边
- 从这条边出发，维护一棵树，
  - 逐个添加边使得它成长为生长树
  - 添加的边必须是一个顶点在树里，一个顶点不在
  - 把符合条件的边里选一条最短的



Prim算法例子

- 第一步选一条最短的边
- 从这条边出发，维护一棵树，
  - 逐个添加边使得它成长为生长树
  - 添加的边必须是一个顶点在树里，一个顶点不在
  - 把符合条件的边里选一条最短的



伪代码

- 复杂度是$O(nm)$，如果不优化的话
  - 每次加一个顶点，需要遍历边来找到符合条件的边

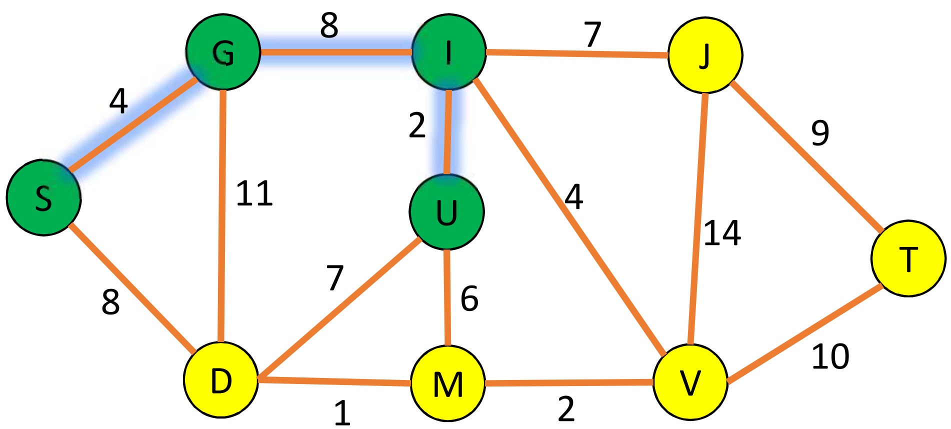
MST = { (s,u) }, （s, u）是最短的边
VisitedVertices = { s, u }
while |VisitedVertices| < |V|:
    $\boldsymbol{find\ the\ lightest\ edge}\ (\boldsymbol{x}, \boldsymbol{v})\ \boldsymbol{in\ E,}$
        $\boldsymbol{x}$ in VisitedVertices and $\boldsymbol{v}$ not
    MST $\leftarrow (\boldsymbol{x}, \boldsymbol{v})$
    VisitedVertices $\leftarrow \boldsymbol{v}$

两个问题

- 一个是怎么证明找到的生成树是MST？
  - 即存在一个MST，使得每个步骤前找到的边都在MST里
- 如何实现，使得找最“好”边的过程优化？

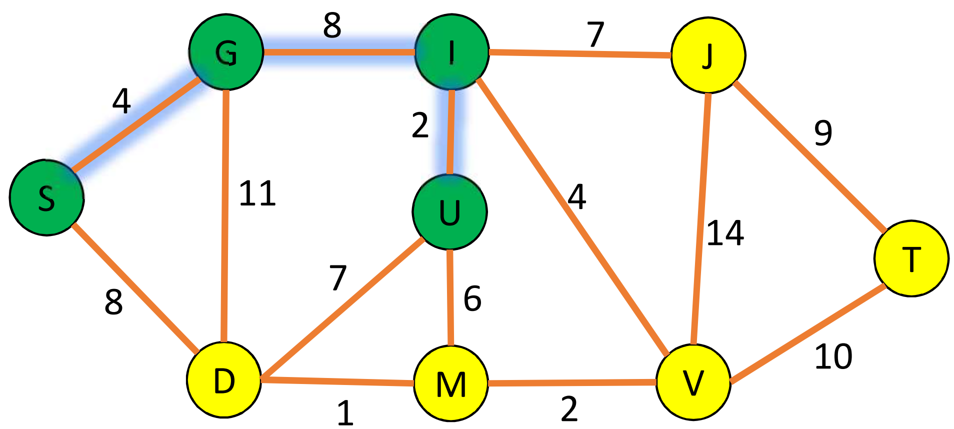
如何实现

- 假如我们已经有SG和GI，IU两条边了
- 如何选择下一条最“好”边
- 其实是选择一个黄点，使得其到绿点集合的距离最短



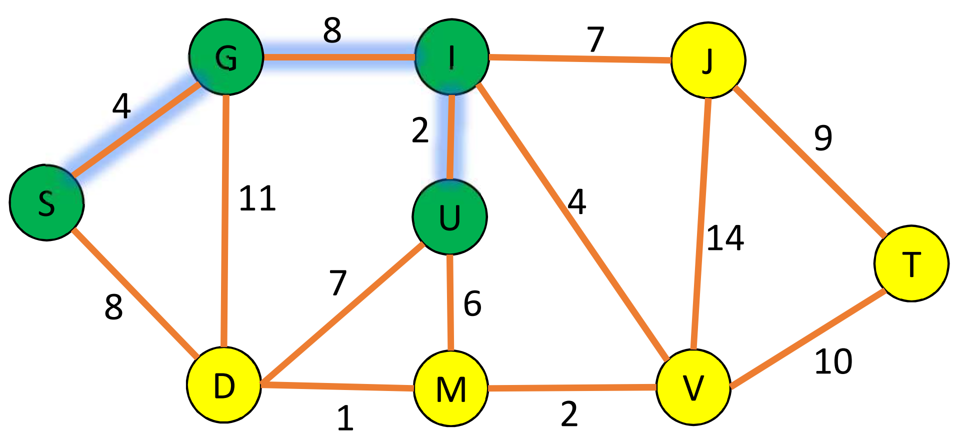
如何实现

- 其实是选择一个黄点，使得其到绿点集合的距离最短
- 回忆dijkstra算法，也是逐个添加顶点，需要找距离源点最近的黄点，通过更新预估值$\boldsymbol{d(v)} = \min\{\boldsymbol{d(v)}, \boldsymbol{d(u)} + \boldsymbol{w(u,v)}\}$
- 这里做法也类似，不过预估值改为$\boldsymbol{d(v)} = \min\{\boldsymbol{d(v)}, \boldsymbol{w(u,v)}\}$



如何实现

- 对每个顶点添加一个预估值$\boldsymbol{d(v)}$和其父节点指针
- 不过预估值改为$\boldsymbol{d(v)} = \min\{\boldsymbol{d(v)}, \boldsymbol{w(u,v)}\}$



伪代码

- 初始化
  - 未完成
  - 完成
- 最短边寻找过程

For each vertex u

$\quad \boldsymbol{d}(\boldsymbol{u}) = \infty$; $u.status =$

$\boldsymbol{d}(\boldsymbol{s}) = 0, \boldsymbol{s}$是最短边的一个顶点

For $i = 1, \ldots, n$:

$\quad$ **Find** u with status **,** such that $\boldsymbol{d}(\boldsymbol{u})$ is min

$\quad$ **For** each neighbor v with status :

$$\boldsymbol{d}(\boldsymbol{v}) \leftarrow \boldsymbol{min}\{\boldsymbol{d}(\boldsymbol{v}), \boldsymbol{w}(\boldsymbol{u}, \boldsymbol{v})\}$$

$$\boldsymbol{p}(\boldsymbol{v}) \leftarrow \boldsymbol{u} \text{ if } \boldsymbol{d}(\boldsymbol{v}) \text{ is updated}$$

$\quad u.status =$

算法复杂度

- 和Dijkstra算法一样
  - $O((n+m)\log(n))$，如果优先队列是红黑树，最小堆
  - $O(n\log(n)+m)$，如果优先队列是Fibonacci堆

为什么正确

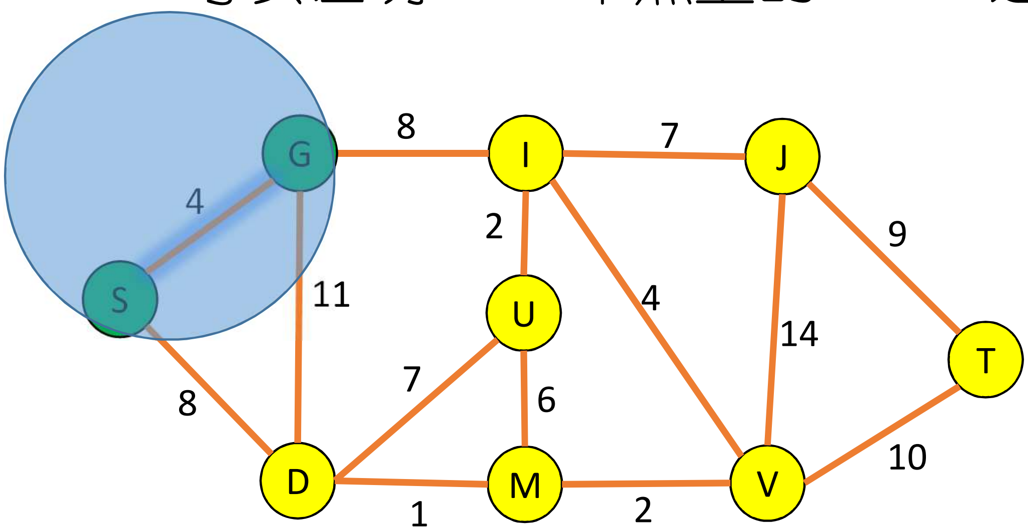
- 我们每一步根据当前状态选择一条“最好”边
- 要保证最后选出的生长树是最小（之一）
  - 需要证明子问题的最优解还原成原问题的解还是最优的
  - 或者证明每一步选择后，之前的子树都在某一个MST里

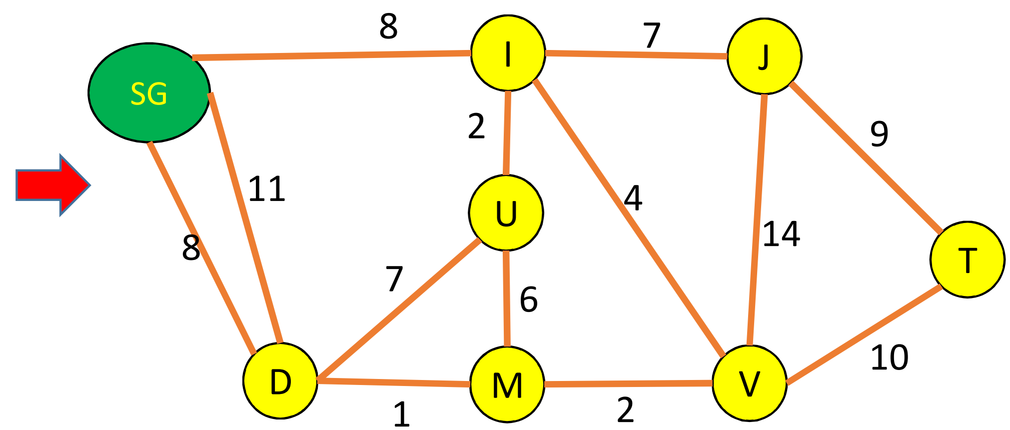
分解原问题

- 子问题：在k个顶点集上，起始点是s，找一个最小的生成树
- 怎么归纳？
  - 从起始点出发的边上找一条最短边，把这两个点塌陷成点s，
  - 就变成了k-1个顶点集中，起始点是s，找一个最小生成树
- 需要证明，从子问题的MST，再添加塌陷的那条边，结果还是k个顶点集上的ST，而且是最小的

归纳方法一

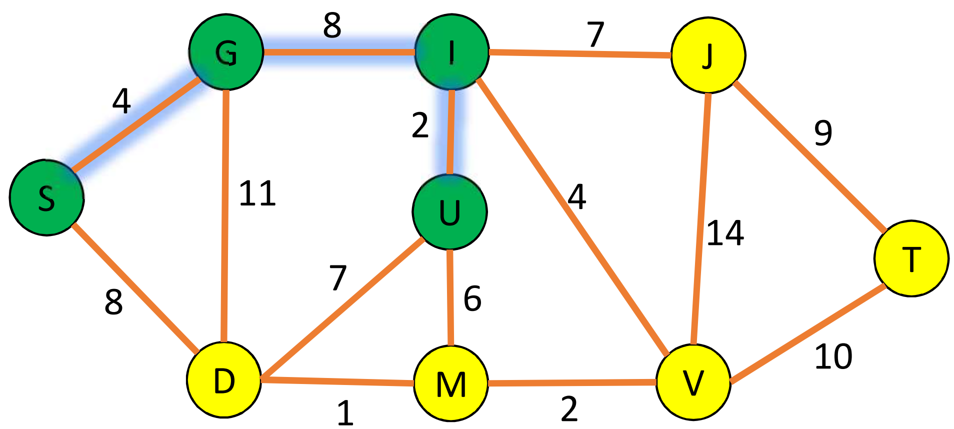
- 选了一条最短的边SG，显然存在MST使得SG是它的一条边
- 把S、G塌陷成一个大的节点，原问题就变成了$k-1$个点上的MST问题
- 可以证明$k-1$个点上的MST + 边SG就是k个点上的MST





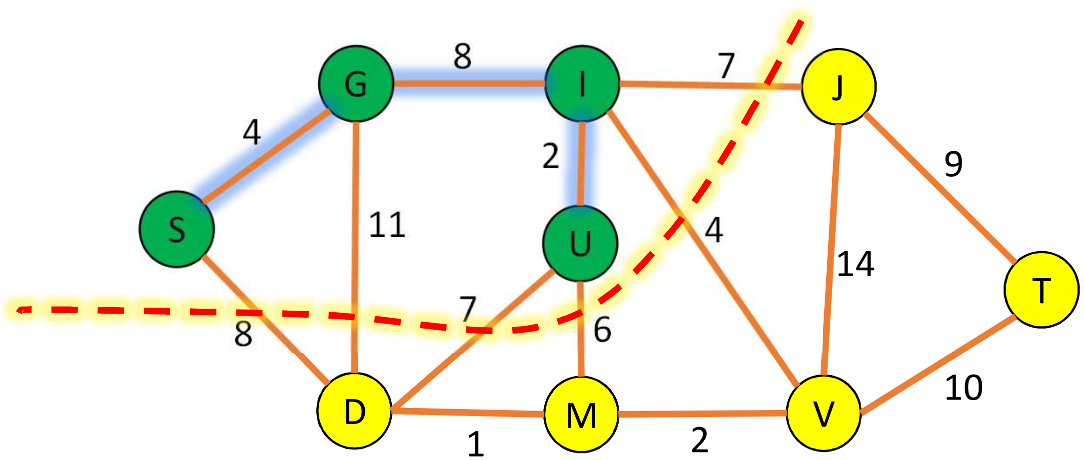
归纳方法二

- 假设中间状态，SG，GI，IU已经选择
- 证明下一步选择后，之前的子树都在某一个MST里



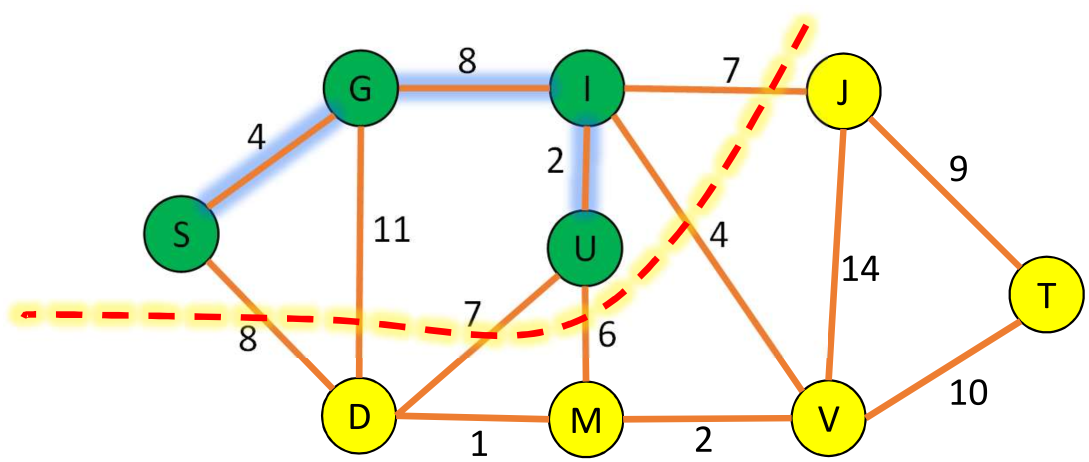
一般情形

- 实际上，我们可以用一条曲线切割图，一边是绿点，一边是黄点。
- 为什么可以这么做，因为SGIU是树结构，是可以塌陷的
- 如果绿点集合有环，比如GIJVMD，把U包在里面，ST在外面，这样是不存在这样的cut，使得一边一种颜色，像bipartite



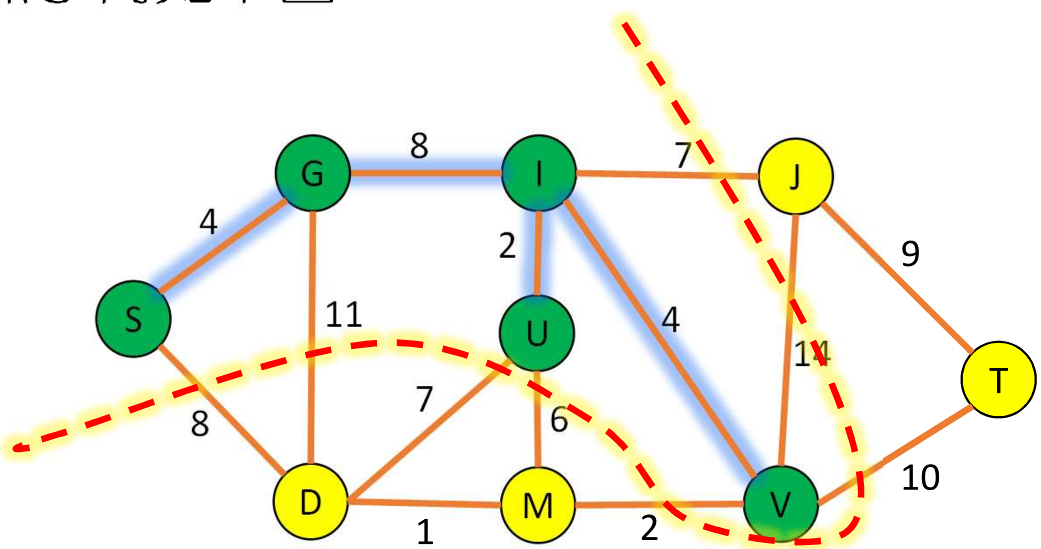
一般情形

- 归纳过程：假设之前选的子树都是某一MST的一部分
- 下一步：就是在虚线相交的边中选一条最短的边
- 注意这些边中任意一条选中都满足子树得条件



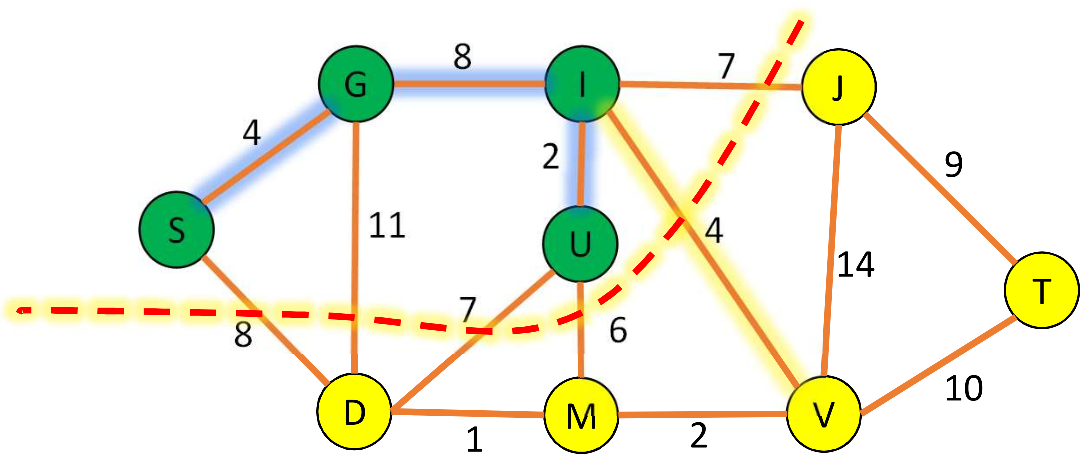
一般情形

- 归纳过程：选一条最短的边后的新子树还是某一MST得一部分
- 算法是在虚线相交的边中选一条最短的边，然后把邻接顶点V变成绿点
- 新的绿点子树如下图



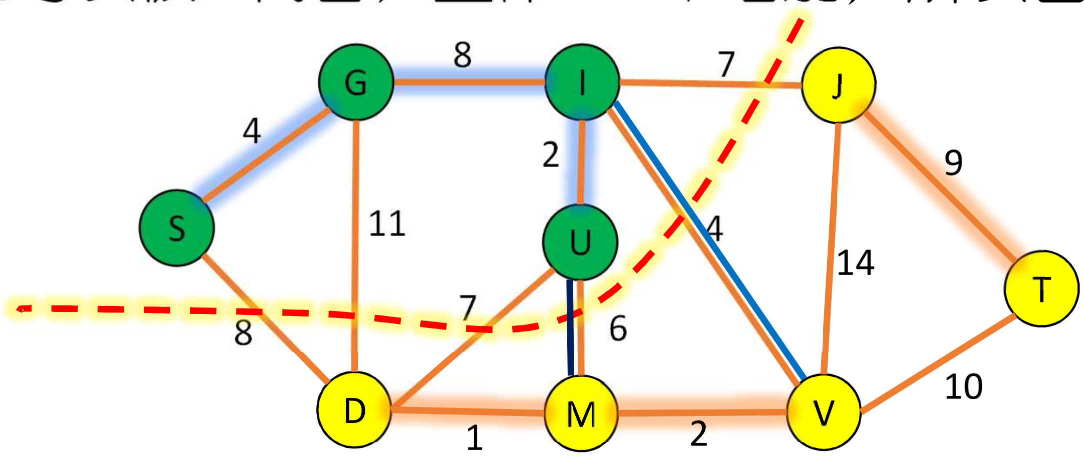
证明

- 要证明新的绿点子树就是某一MST的一部分
- 数学归纳法：假设之前的绿点子树是某一MST，τ的一部分
- 证明下一步选择后，存在某一MST，新绿点子树是τ'的一部分
- 注意下一步算法选择IV



为什么正确

- 因为τ里至少有一条边连接绿点集合和黄点
- 反证法：如果这条边不是最短的IV
- 那么我们可以添加最短的IV，这样肯定有一个环，而且IV是环的一条边，那么一定有另外一条边也是一端是绿色，一端是黄色。
- 这条边可以被IV代替，整体cost不增加，所以也是MST

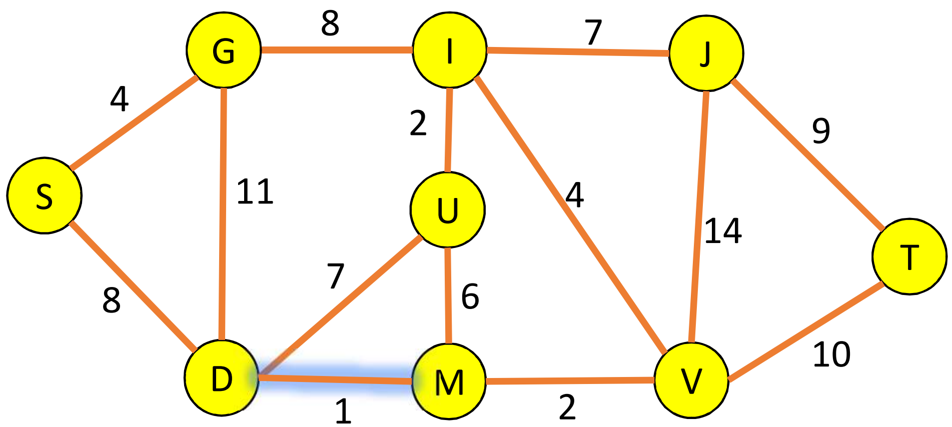


Kruskal算法

另一个贪婪算法

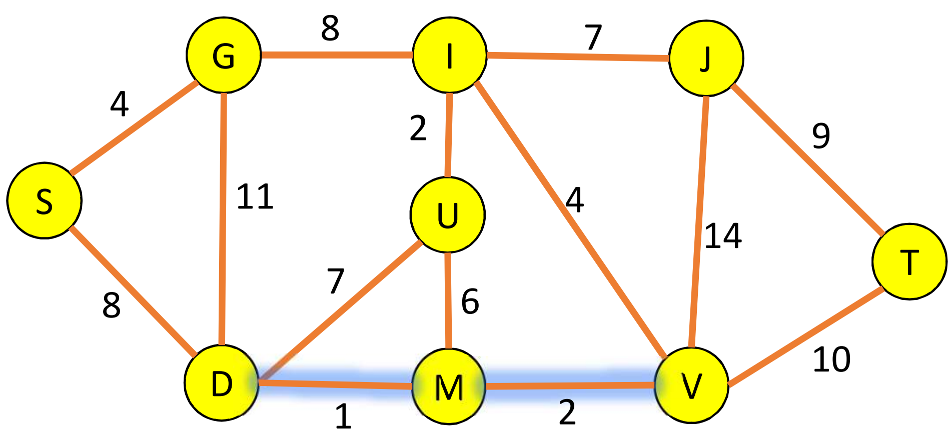
Kruskal

- 每一步都选一条最短的边
- 不管是不是和之前的连通
- 只要新的边不造成环路



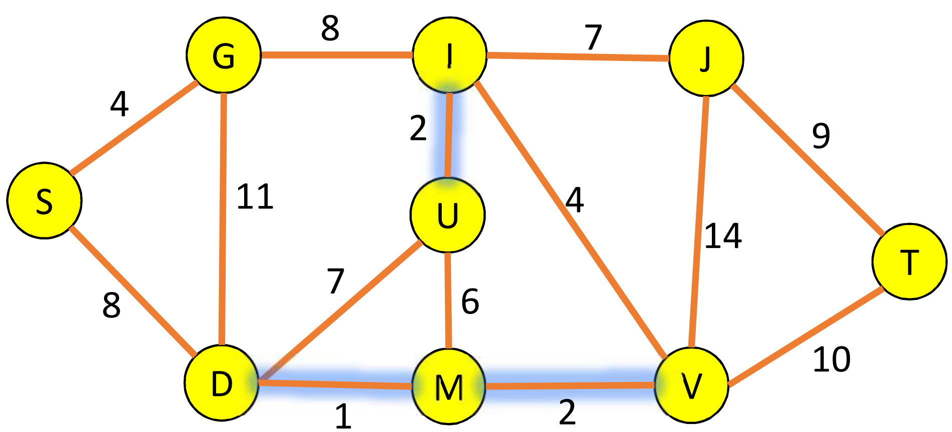
Kruskal

- 每一步都选一条最短的边
- 不管是不是和之前的连通
- 只要新的边不造成环路



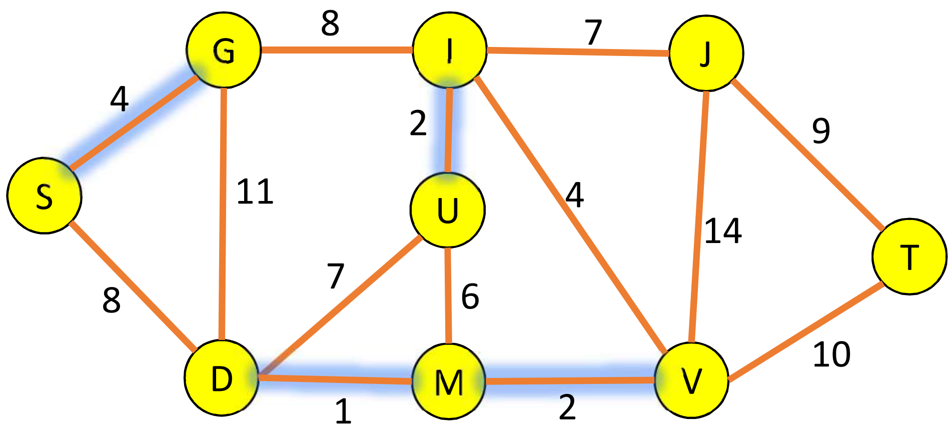
Kruskal

- 每一步都选一条最短的边
- 不管是不是和之前的连通
- 只要新的边不造成环路



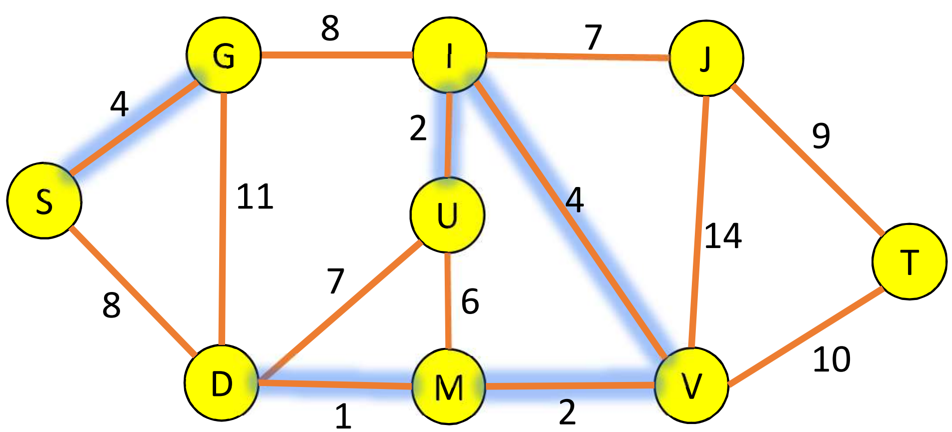
Kruskal

- 每一步都选一条最短的边
- 不管是不是和之前的连通
- 只要新的边不造成环路



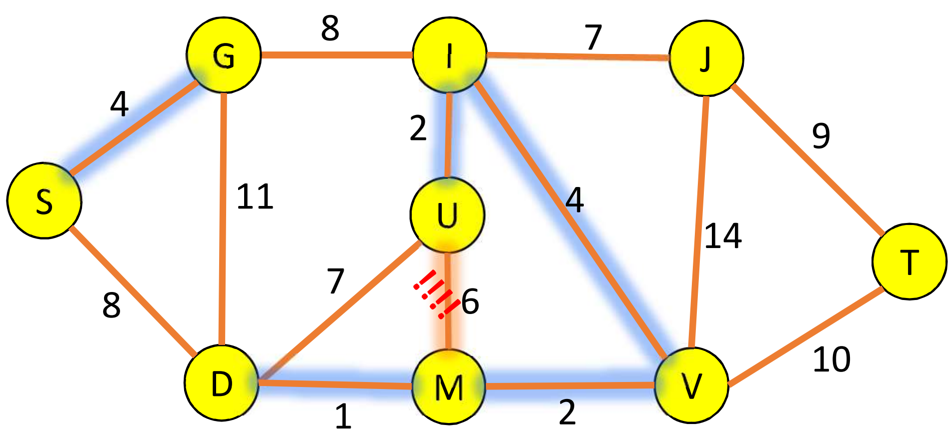
Kruskal

- 每一步都选一条最短的边
- 不管是不是和之前的连通
- 只要新的边不造成环路



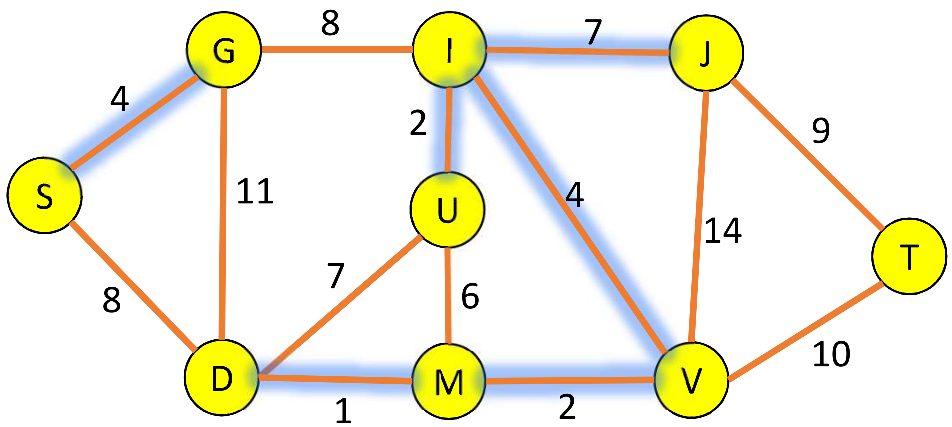
Kruskal

- 每一步都选一条最短的边
- 不管是不是和之前的连通
- 只要新的边不造成环路



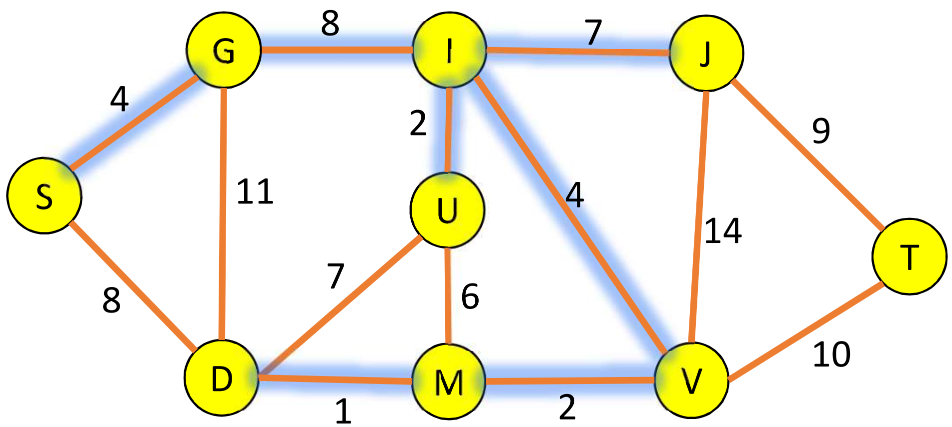
Kruskal

- 每一步都选一条最短的边
- 不管是不是和之前的连通
- 只要新的边不造成环路



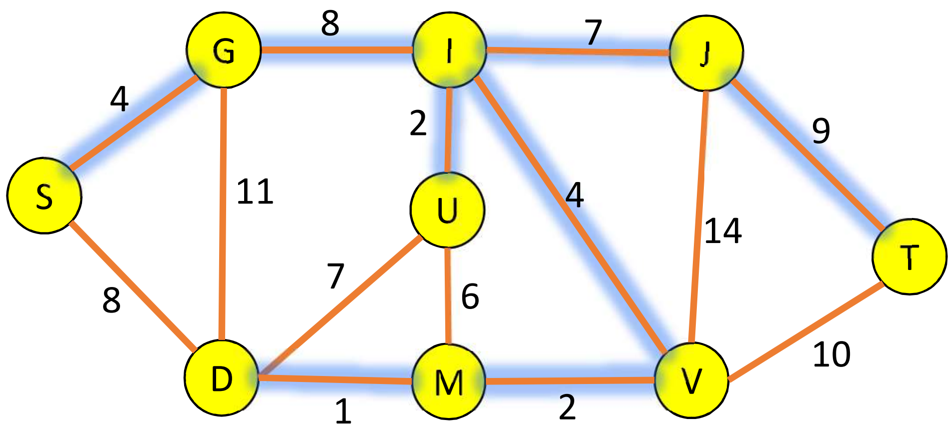
Kruskal

- 每一步都选一条最短的边
- 不管是不是和之前的连通
- 只要新的边不造成环路



Kruskal

- 每一步都选一条最短的边
- 不管是不是和之前的连通
- 只要新的边不造成环路



伪代码

- 初始化

Sort all edges by weight
$\boldsymbol{MST} = \{\}$

- 最短边寻找过程

For each $edge$ **e** $in\ sorted\ order$:
    If add **e** to $\boldsymbol{MST}$ won’t cause a cycle
        add **e** to $\boldsymbol{MST}$

两个问题

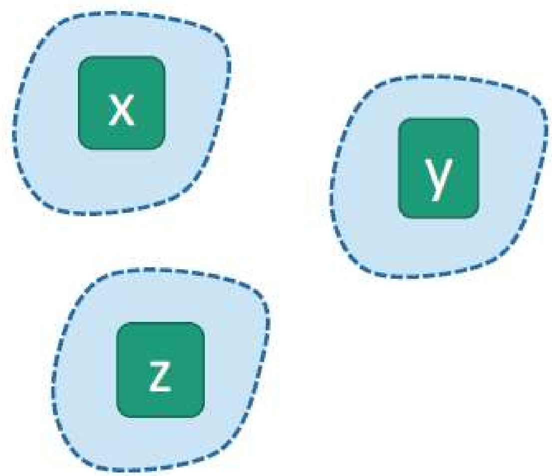
- 一个是怎么证明找到的生成树是MST？
  - 即存在一个MST，使得每个步骤中找到的边都在这个MST里
- 如何实现？
  - Dfs算法可以检查cycle
  - 使用Union-Find数据结构检查cycle更好（logn复杂度）

Union-find结构

- **makeSet(u):** 建立集合{u}
- **find(u):** 返回u所在集合的代表元素（根节点）
- **union(u,v):** 合并u所在集合和v所在集合

```
makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)
```

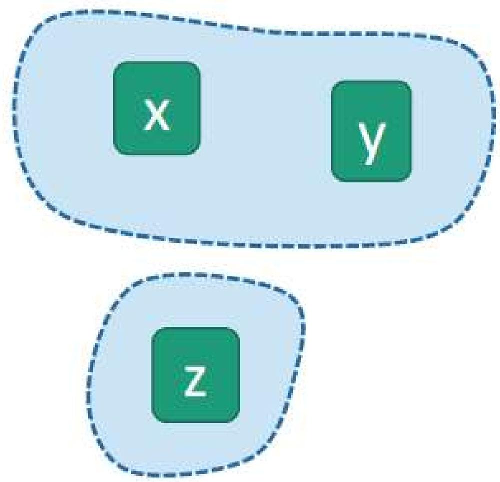


Union-find结构

- **makeSet(u):** 建立集合{u}
- **find(u):** 返回u所在集合的代表元素（根节点）
- **union(u,v):** 合并u所在集合和v所在集合

```
makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)
```



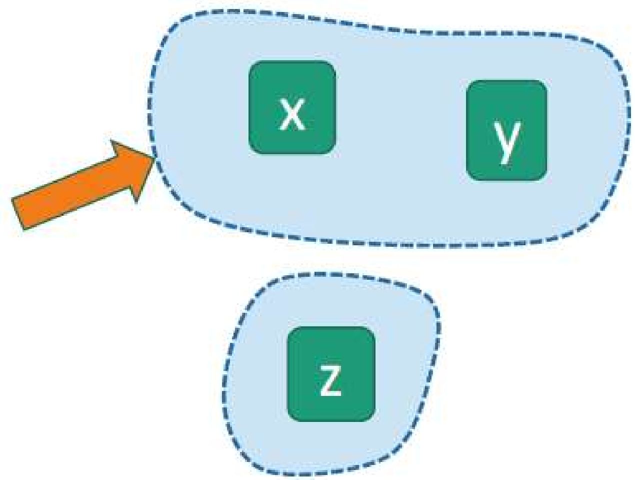
Union-find结构

- **makeSet(u):** 建立集合{u}
- **find(u):** 返回u所在集合的代表元素（根节点）
- **union(u,v):** 合并u所在集合和v所在集合

```
makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)

find(x)
```

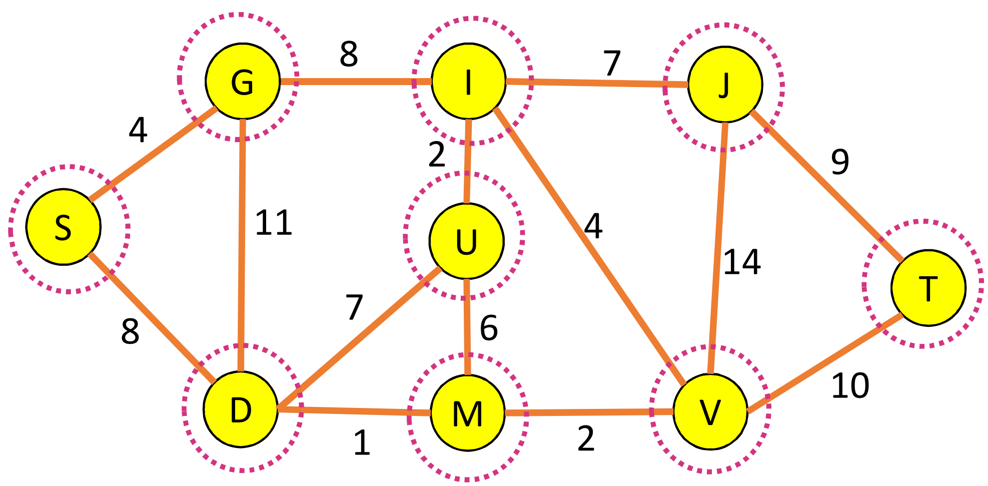


伪代码

```
• kruskal(G = (V,E)):
    • Sort E by weight in non-decreasing order
    • MST = {}                           // initialize an empty tree
    • for v in V:
        • makeSet(v)                     // put each vertex in its own tree in the forest
    • for (u,v) in E:                    // go through the edges in sorted order
        • if find(u) != find(v):         // if u and v are not in the same tree
            • add (u,v) to MST
            • union(u,v)                 // merge u's tree with v's tree
    • return MST
```

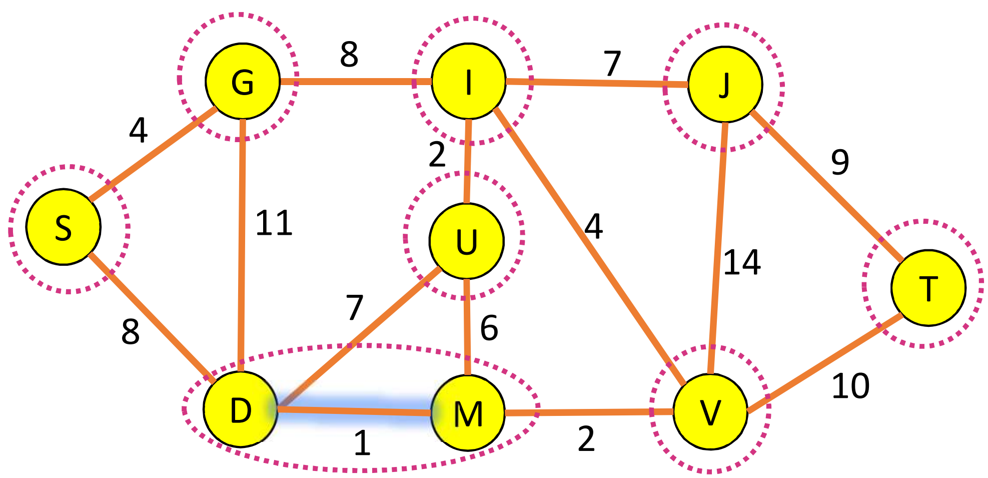
实现

- 起始状态



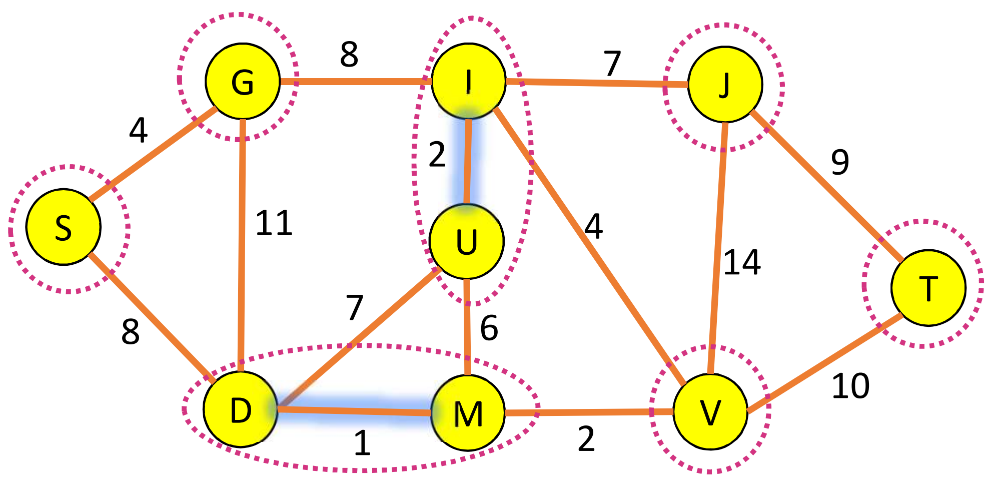
实现

- 下一步骤



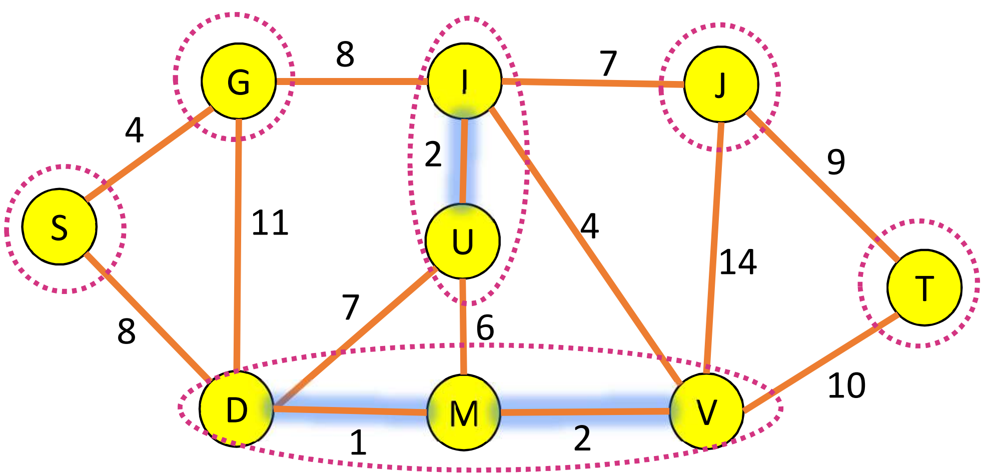
实现

- 下一步骤



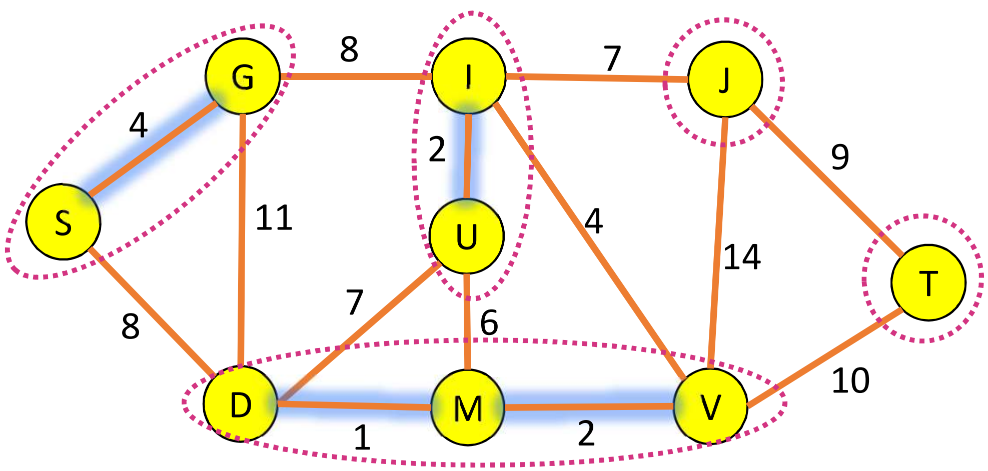
实现

- 下一步骤



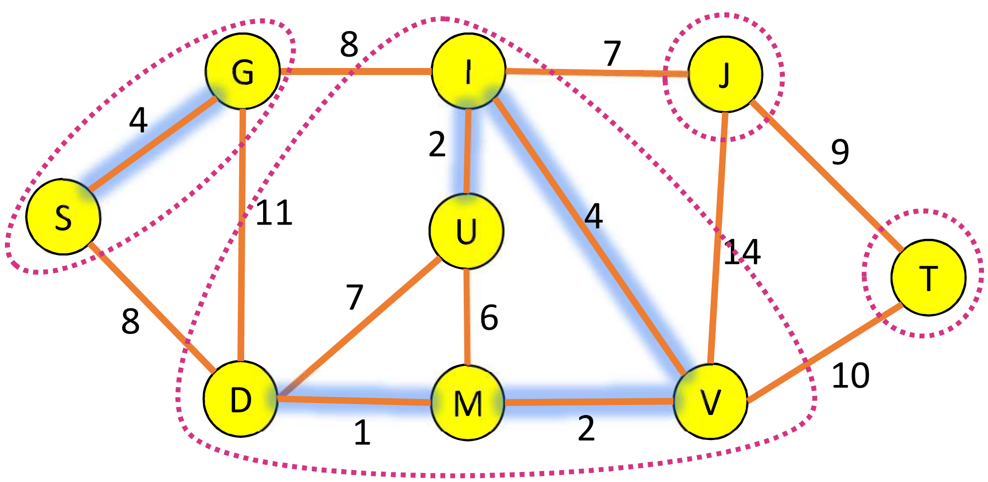
实现

- 下一步骤



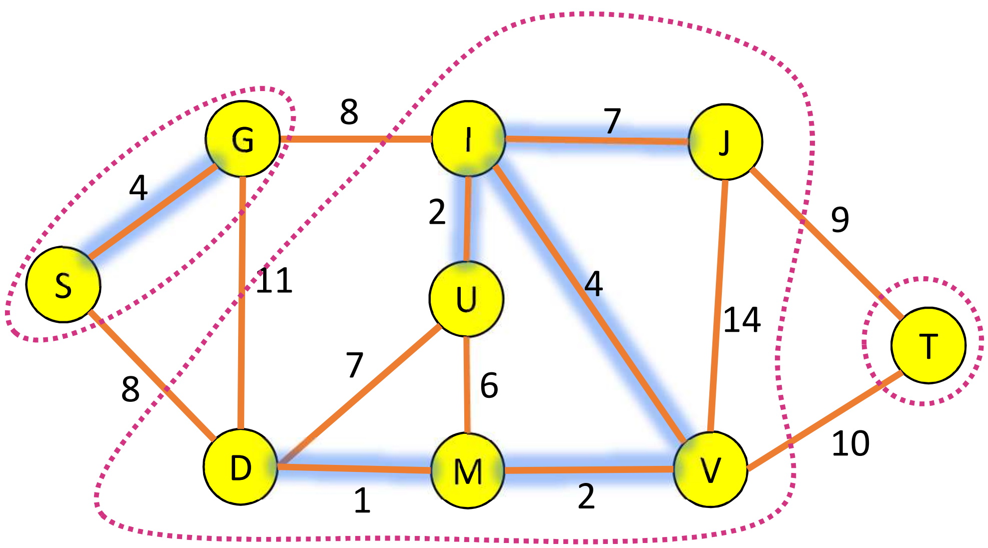
实现

- 下一步骤



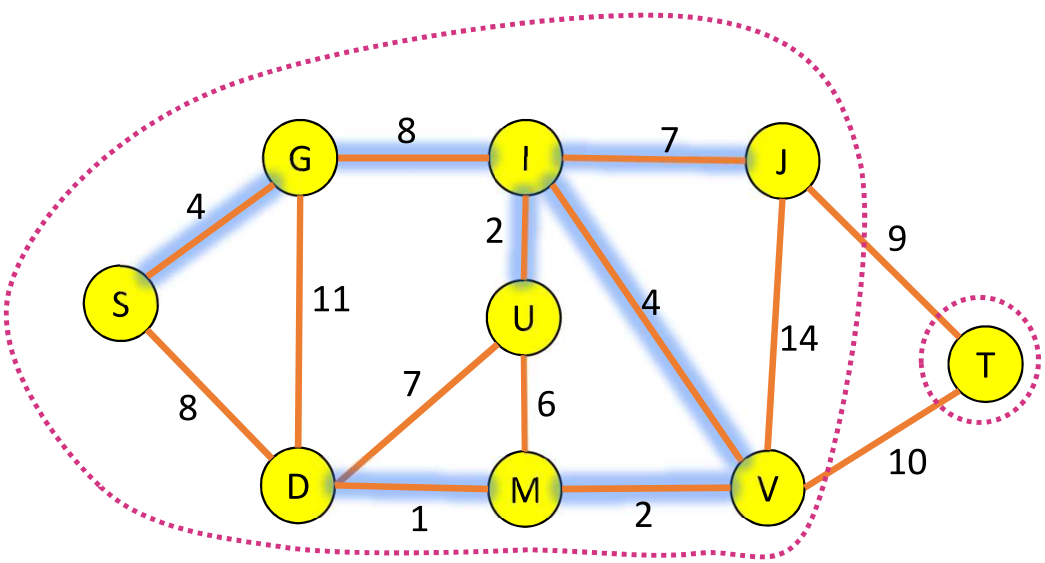
实现

- 下一步骤



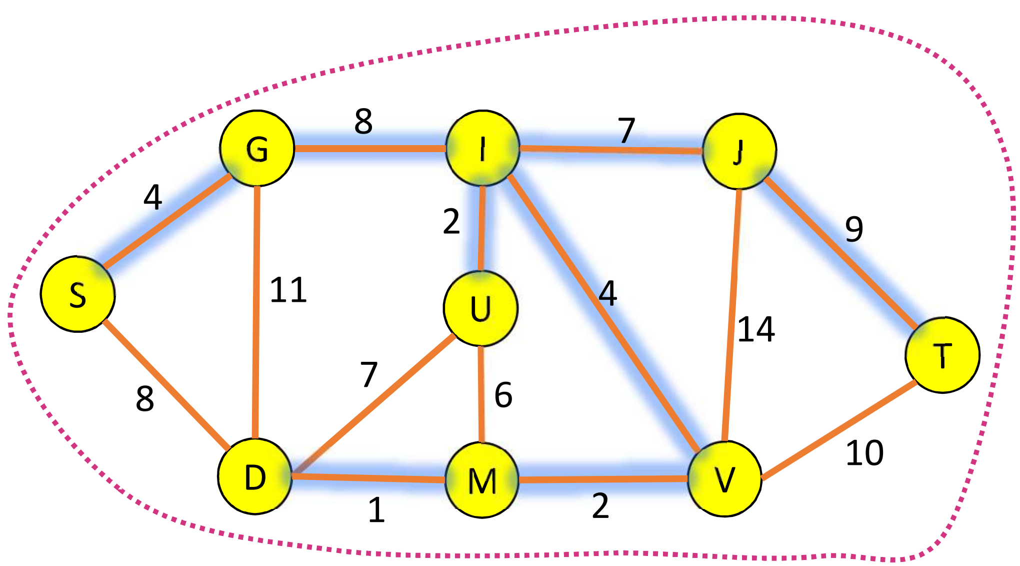
实现

- 下一步骤



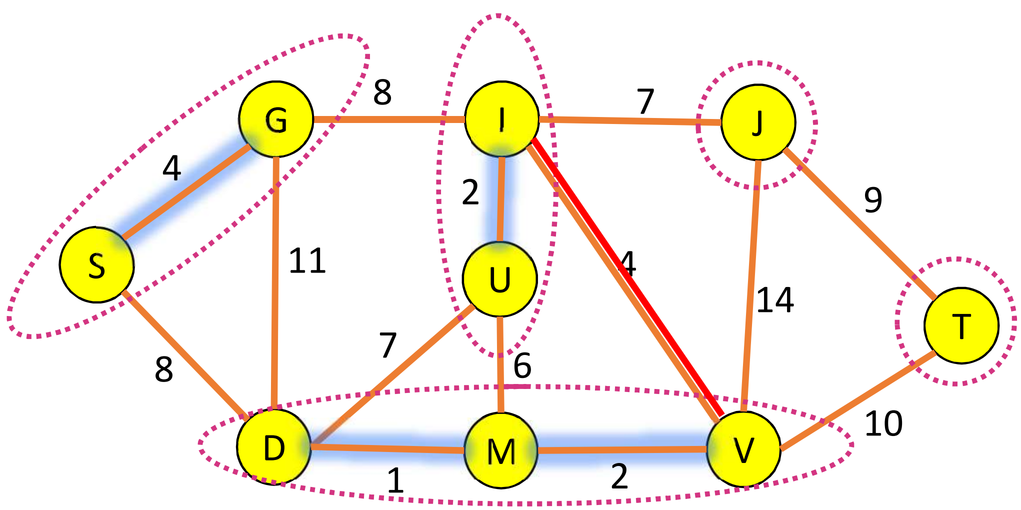
实现

- 下一步骤



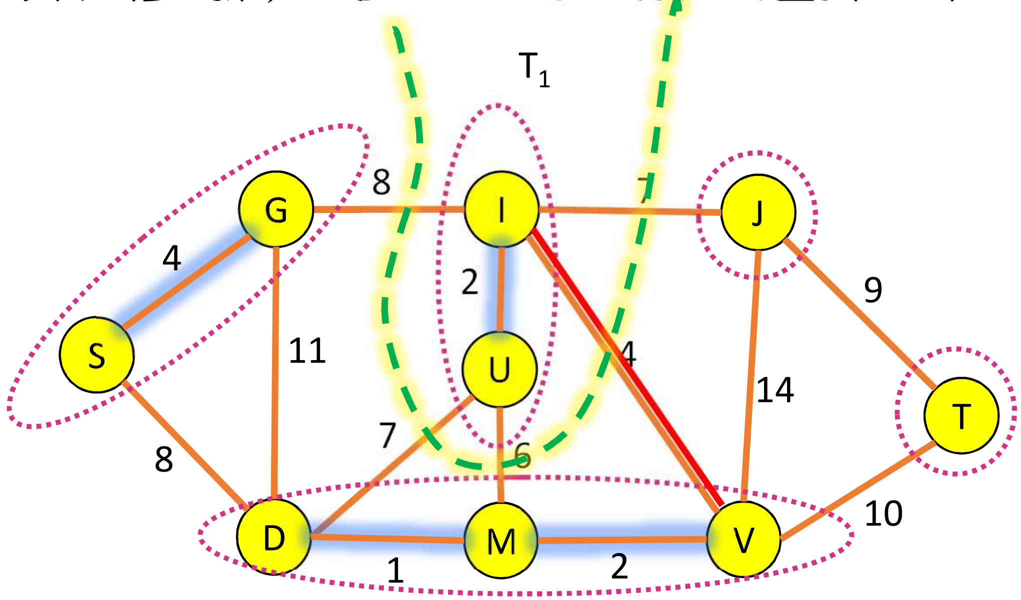
归纳

- 假设第k步时，之前选的边都在某个MST里
- 下一步，按算法，会选择IV
- 需要证明，选择IV后，还在某个MST里



归纳

- 需要证明，选择“IV”后，还在某个MST里
- 考虑$cut\{T_1, V - T_1\}$
- 类似prim算法步骤，可以证明之前的选择还在某个MST里面



对比

- Prim's
  - 维护一颗树，慢慢成长到MST
  - 时间复杂度$O(m + nlog(n))$，用Fibonacci堆实现
- Kruskal's
  - 维护森林，成长到MST
  - 时间复杂度$O(mlog(n))$，用union-find结构实现

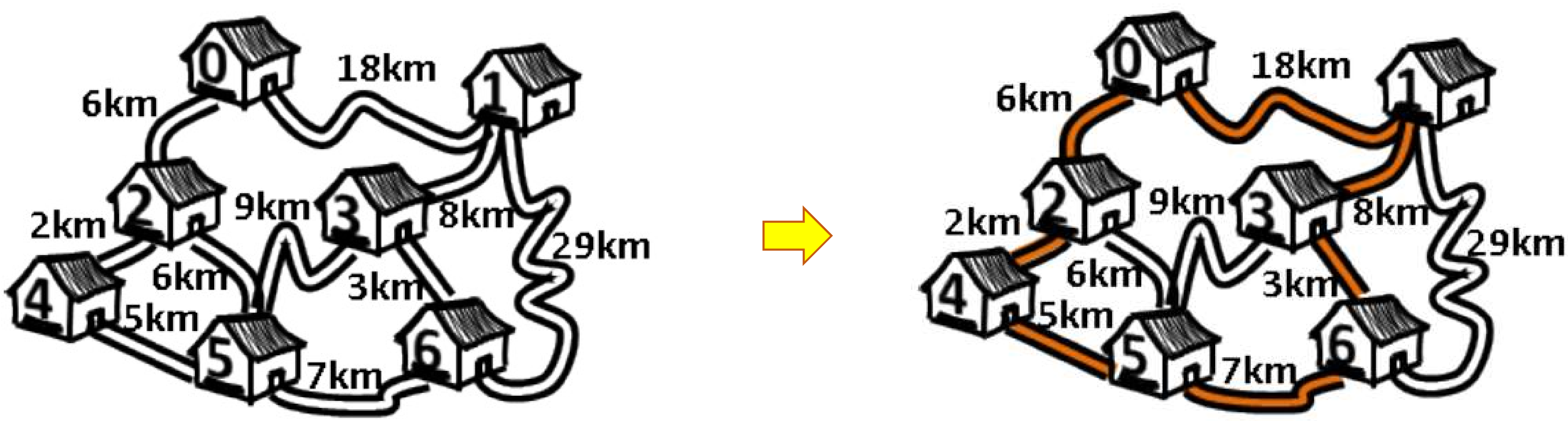
	Description	Runtime
Prim's	Grows a tree	O(|E|log(|V|)) **with red-black tree** O(|E|+|V|log(|V|)) **with Fibonacci heap**
Kruskal's	Grows a forest	O(|E|log(|V|)) **with union-find** O(|E|) **with union-find and radix sort**

总结

- MST的两个算法
  - Prim's
  - Kruskal's

- 都是用贪婪算法
  - 每次都做出一个选择
  - 选择后，只依赖于一个子问题
  - 每次选择都没有淘汰最优解
  - 即存在某个最优解使得之前的选择是其一部分

The Travelling Salesman Problem

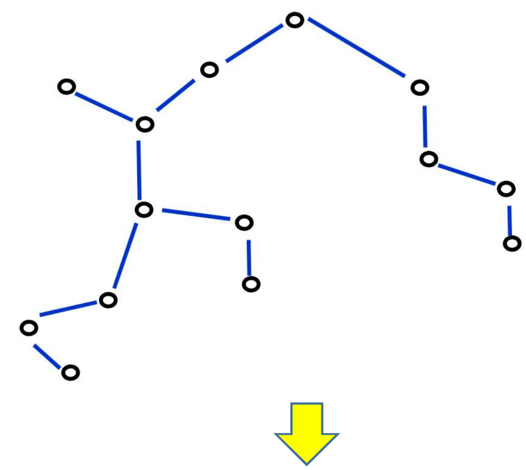


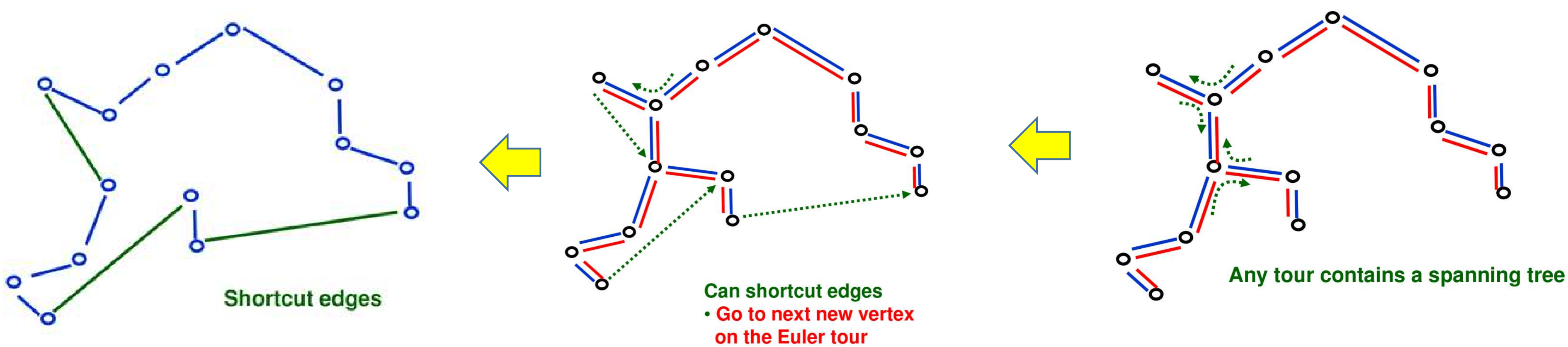
TSP问题

- 旅行商问题，货担郎问题
- 如何找一条最短的路过所选城市一次的闭路径（tour）
  - 每个城市过且仅过一次
  - 最后回到起点
- 给定一个完全图$G(V,E)$以及相应得距离$d(v_i,v_j)$，找一条cost最小得闭路径
- 优化问题，问题很简单，求解却非常难

2-approximation

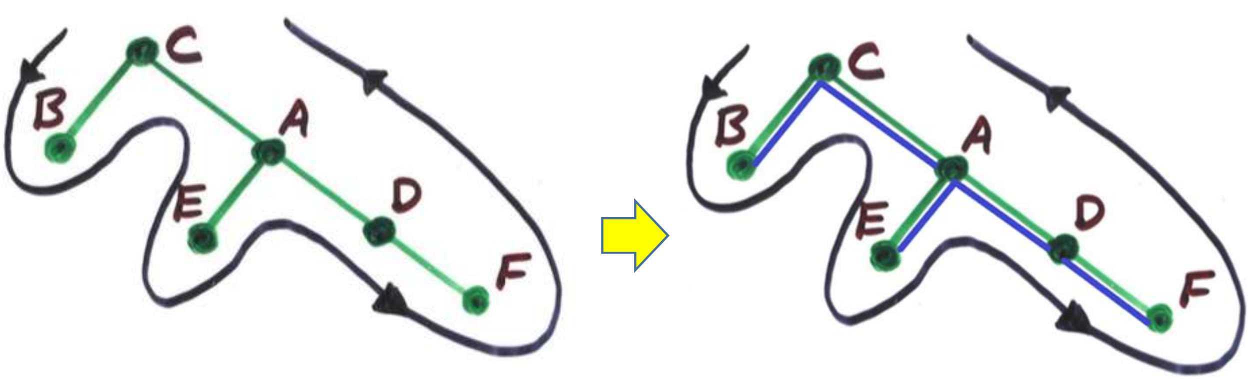
- 找MST（Prim算法）
- 任选一个顶点作为根节点
- 用DFS遍历，并记录访问顺序（参看下页）
- 根据三角不等式转换成TSP回路（shortcutting）





$\mathrm{MST}(G) \leq \mathrm{TOUR}_{\mathrm{OPT}}(G) \leq 2\,\mathrm{MST}(G) \leq 2\,\mathrm{TOUR}_{\mathrm{OPT}}(G)$

树的欧拉回路（DFS）



Euler tour E

1	2	3	4	5	6	7	8	9	10	11
C	B	C	A	E	A	D	F	D	A	C

```
void EulerTour(const Node *root)
{
    if(root == nullptr)
    {
        return;
    }

    cout<< root->key << " ";

    EulerTour(root->left);
    if (root->left)
    {
        cout<< root->key << " ";
    }

    EulerTour(root->right);
    if (root->right)
    {
        cout<< root->key << " ";
    }
}
```

Q&A

Thanks!