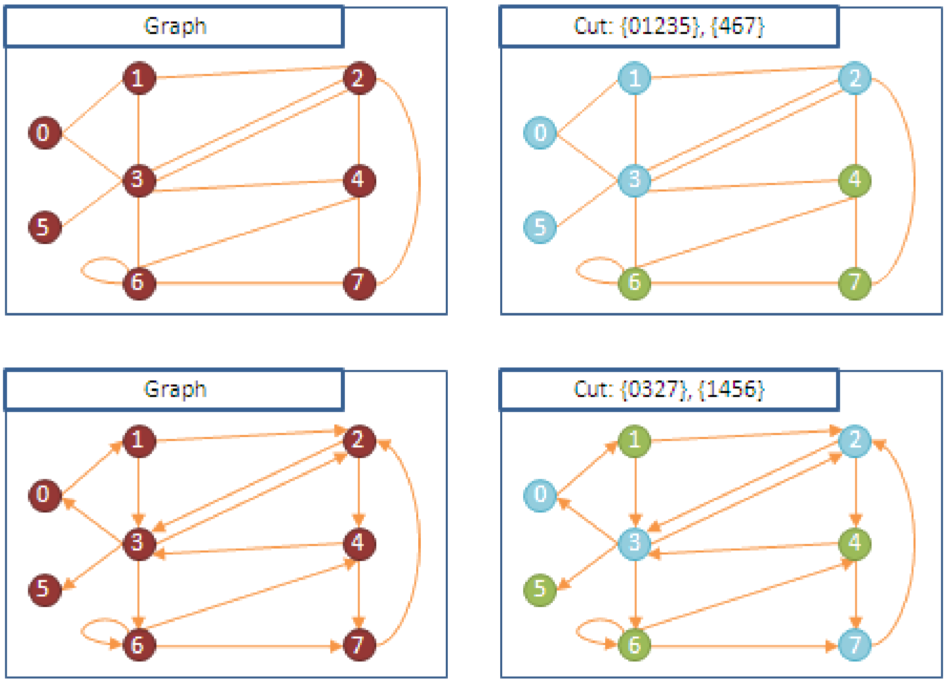
CUT Stoer-Wagner Algorithm [转]

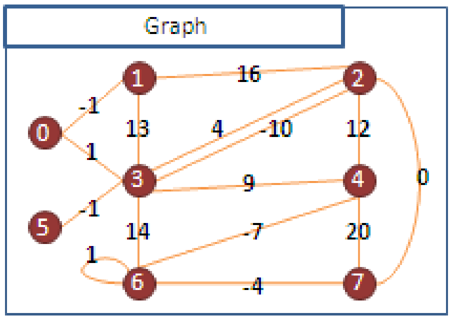
标签： algorithm search matrix 算法 tree path

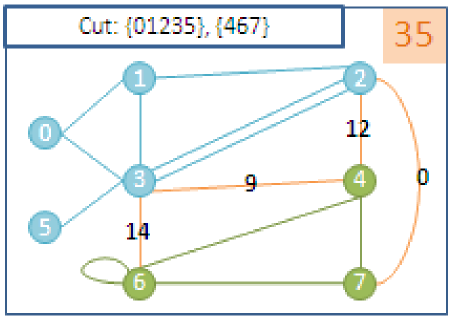
2009-10-12 18:50 1042人阅读 评论(0) 收藏 举报

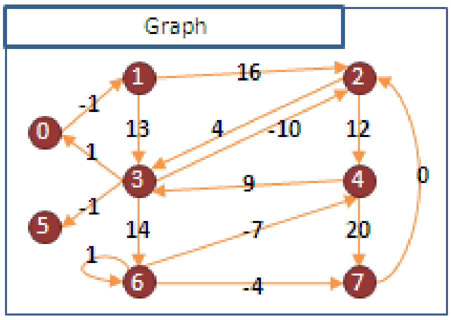
中译「割」。一个 Cut 把一张图上的点分成两群。也就是说，每一个点都必须选边站（也可以全部都站在同侧，另一侧就是空的）。以数学术语来描述：一个 Cut 把一张图上所有点所构成的集合，重新划分成两个集合。

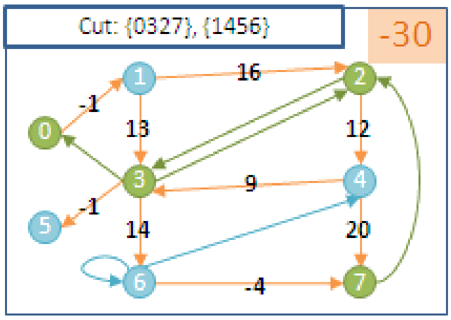


如果图上的边拥有权重，一个 Cut 也可以有权重。无向图：由第一群点横跨到第二群点的所有边的权重总和。有向图：由第一群点横跨到第二群点的所有边的权重总和，减去由第二群点横跨到第一群点的所有边的权重总和。









Cut 切断了点与点之间的连结，将一张图一分为二。各位可以想一想 Cut 可以应用在哪些地方。

【注：一般大家在谈 Cut 的时候，经常是使用另外一种定义： Cut 的其中一侧至少要有一点，而且图上的边的权重均非负值。我想这应该是因为 Cut 的演算法，一开始是从 Flow 的演算法推演来的。】

Minimum Cut （ Min-Cut ）：

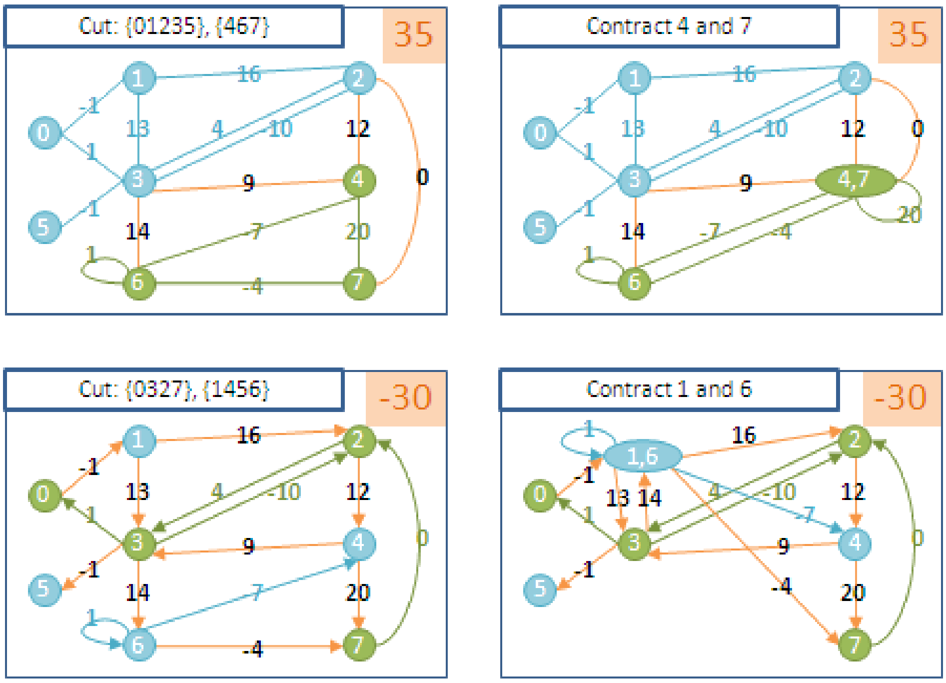
「最小割」。给定一张图，权重最小的 Cut 就是 Min-Cut 。一张图上可能会有许多个 Min-Cut 。

Maximum Cut （ Max-Cut ）：

最大割」。给定一张图，权重最大的 Cut 就是 Max-Cut 。一张图上可能会有许多个 Max-Cut 。

Contraction:

「收缩」。在一个 Cut 之中，把 Cut 的其中一侧的任两点作合并，不会影响此 Cut 的权重。合并图上的点、却不影响一个 Cut 的权重，这个行为就叫做 Contraction 。

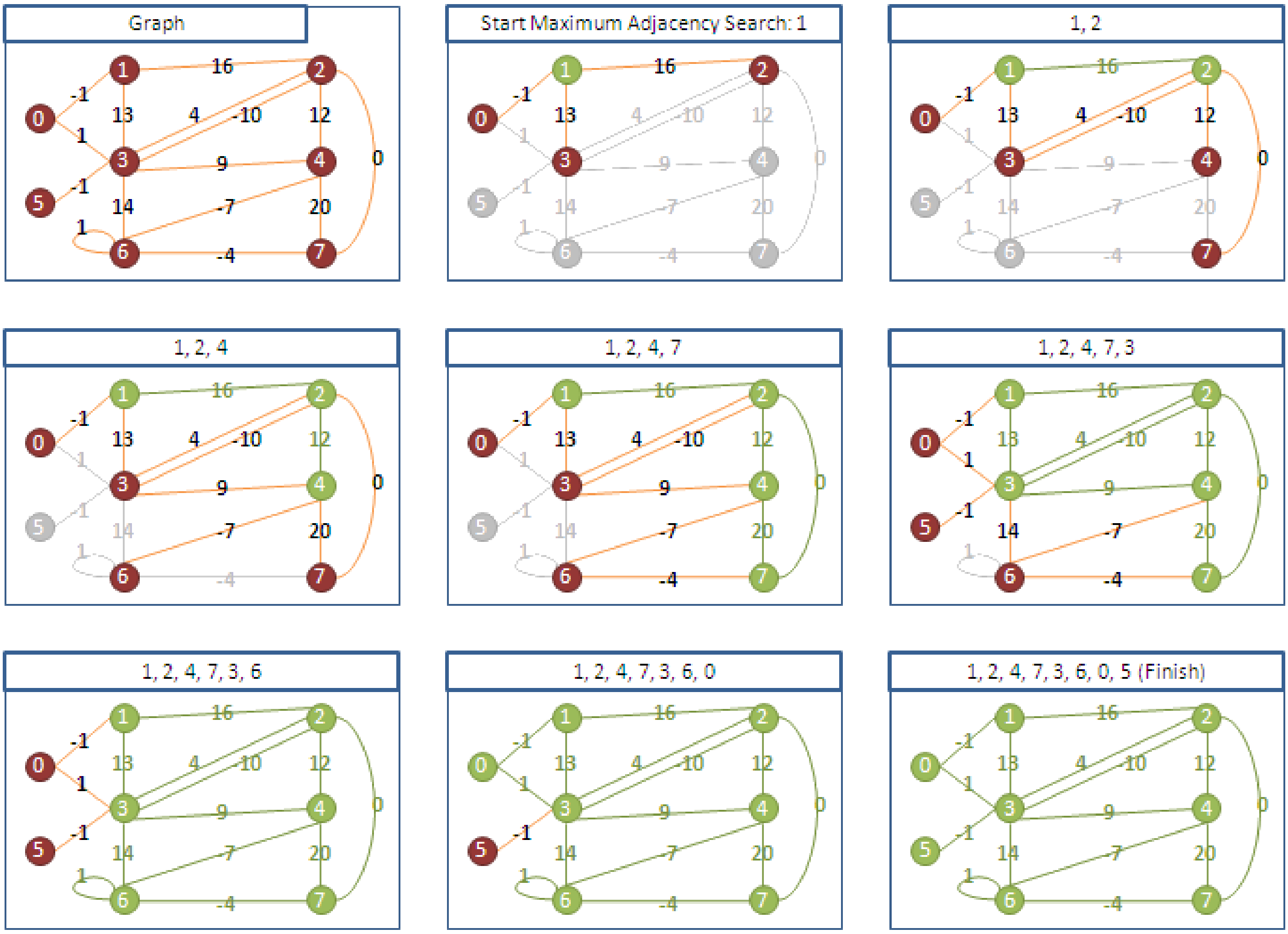


进行 Contraction 之后，甚至可以把两点之间多重的边的权重相加合并，也不会改变 Cut 的权重。

Maximum Adjacency Search:

在讲述 Min-Cut 的演算法之前，这里先介绍一个与 Min-Cut 极有关系的 Maximum Adjacency Search：

1. 建立一个空的A集合。
2. 首先随便在图上找一个点，加入到A集合当中。
3. 令w(A, x)是「目前的A集合的各个点」与「x点」之间所有的边的权重总和。
   逐次加入一个尚未加入A当中、且w(A, x)最大的x点到A集合中。
4. 图上所有点都加入到A集合之后，各个点加入的顺序即为



Maximum Adjacency Search （ adjacency matrix ）：利用 Dynamic Programming 来实做程式码，时间复杂度是 O(V^2) 。如同 Dijkstra's Algorithm 一样，可以另外再配合 Priority Queue ，成为 O(V+ElogE) 。

```
int map [9][9]; // adjacency matrix
int w[9];            // 纪录各个点到目前的A集合的距离
bool visit[9]; // 纪录各个点是不是已找过

void maximum_adjacency_search()
{
       for (int i=0; i<9; i++) visit[i] = false; // initialize
       for (int i=0; i<9; i++) w[i] = 0;
```

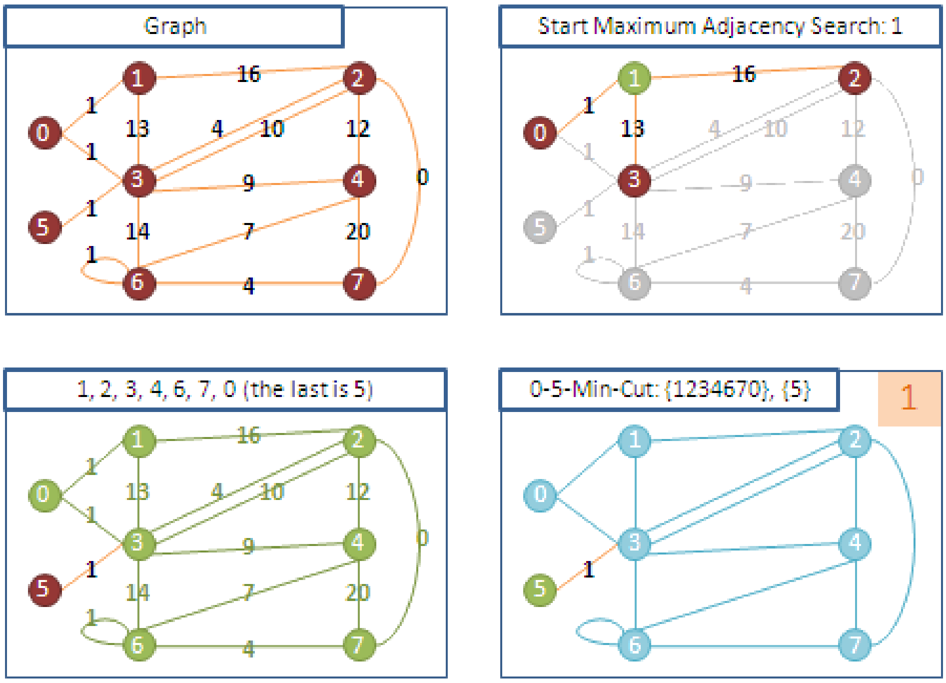
```
        for (int i=0; i<V; ++i)
        {
                // 找出一个尚未加入A当中、且w(A, x)最大的x点。
                int s = 0, max = -1e9;
                for (int j=0; j<V; ++j)
                        if (!visit[i] && w[i] > max)
                                max = w[i], s = i;

                visit[s] = true;       // 加入s点到A集合
                cout << "这次读到第" << s << "点" << endl;

                // 加入s点到A集合后，更新w(A, x)的值。
                for (int t=0; t<V; ++t)
                        if (!visit[t])
                                w[t] += map[s][t];
        }
}
```

Maximum Adjacency Search v.s. Min-Cut:

在一张无向图当中，边的权重皆非负值，令这张图经过 Maximum Adjacency Search 所得到的顺序是 x1-x2-… -xV。在这张图中， {x1 … xV-1} 与 {xV} 这一个 Cut，必会是这张图「限制 xV-1 和 xV 在 Cut 不同侧」的 Min-Cut。



笔者曾努力想过证明，但能力不足，无法证得这个性质。如果有人懂得证明，欢迎告知。

Min-Cut: Stoer-Wagner Algorithm:

用来求出一张无向图的其中一个 Min-Cut 。此处提及的 Cut 其中一侧至少要有一点，而且图上的边的权重均非负值。

任取图上两点，这两点要嘛就是在 Min-Cut 同侧，要嘛就是在 Min-Cut 不同侧。这个想法等同于本站文件「Divide and Conquer — Combination 」分割问题的想法。

如果这两点将在 Min-Cut 同侧，那么我们可以进行 Contraction ，合并这两点为一点，制作出一张比原图还要少一点的图，缩小问题范畴。这么做不会影响此两点在同侧的 Cut 们的权重，当然也就不会影响 Min-Cut 的权重。

如果这两点会在 Min-Cut 不同侧，则需要想办法找出这两点在不同侧的 Min-Cut 。

演算法:

Stoer-Wagner Algorithm 巧妙地运用了 Maximum Adjacency Search 的性质，求出一个限制某两点在不同侧的Min-Cut 。

令map[a][b]是a点到b点的距离（即是边的权重）。

1. 重复下面这件事V-1次，以求出其中一侧至少要有一点的Min-Cut：

甲、Maximum Adjacency Search：对图上所有点使用Maximum Adjacency Search，

最后两点依序为s点和t点，求出s点和t点在不同侧的Min-Cut。

乙、Contraction：合并s点和t点，继续求出s点和t点在同侧的Min-Cut。

对于图上每一点x，map[s][x]=map[x][s]=map[s][x]+map[t][x]（把t点并至s点）

时间复杂度：

约是 V 次的 Maximum Adjacency Search ，总共是 O((V^2) * V) = O(V^3)

计算 Min-Cut 的权重（ adjacency matrix ）：

实做方式可参考「 Shortest Path: Dijkstra's **Algorithm** 」。

```
int map[9][9]; // adjacency matrix
int w[9];              // 纪录各个点到目前的A集合的距离
bool visit[9]; // 纪录各个点是不是已找过
bool combine[9];        // 纪录各个点被合并过了没

void maximum_adjacency_search(int& s, int& t, int& s_t_min_cut_cost)
{
       for (int i=0; i<9; i++) visit[i] = false; // initialize
       for (int i=0; i<9; i++) w[i] = 0;

       for (int i=0; i<V; ++i)
       {
               // 找出一个尚未加入A当中、且w(A, x)最大的x点。
               int a = 0, max = -1e9;
               for (int j=0; j<V; ++j)
                       if (!combine[i] && !visit[i] && w[i] > max)
                               max = w[i], a = i;
```

```
            visit[a] = true;       // 加入a点到A集合

            s = t; t = a;     // 不断纪录目前的Cut位置
            s_t_min_cut_cost = w[t];       // 不断纪录目前的Cut权重

            // 加入a点到A集合后，更新w(A, x)的值。
            for (int b=0; b<V; ++b)
                if (!combine[i] && !visit[b])
                    w[b] += map[a][b];
        }
}

void stoer_wagner()
{
        for (int i=0; i<9; ++i) combine[i] = false;

        int ans = 0;
        for (int k=0; k<V-1; ++k)
        {
            // s点和t点在Cut不同侧
            int s, t, s_t_min_cut_cost;
            maximum_adjacency_search(s, t, s_t_min_cut_cost);
            if (s_t_min_cut_cost < ans) ans = s_t_min_cut_cost;

            // s点和t点在Cut同侧
            combine[t] = true;           // 把t点标记为被合并过了，变成不存在。
            for (int i=0; i<V; ++i) // 把t点合并至s点后，更新边的权重值。
                if (!combine[i])
                {
                    map[i][s] += map[i][t];
                    map[s][i] += map[t][i];
                }
        }
        cout << "Min-Cut的权重为" << ans << endl;
}
```

找出一个 Min-Cut （ adjacency matrix ）：

纪录最小值是出现在哪里。【待补文字】

找出所有的 Min-Cut：

欢迎提供想法。【待补文字】

这里提供两题练习题。

Uva 10480 10989

Min-Cut: Karger's Algorithm：

用来求出一张图的其中一个 Min-Cut （或 Max-Cut ）。特别的是，这是个随机演算法，属于 Monte Carlo**Algorithm** ，也就是不保证答案百分之百正确。

演算法：

任取图上两点，这两点要嘛就是在 Min-Cut 同侧，要嘛就是在 Min-Cut 不同侧。如果这两点在 Min-Cut 同侧，可以使用 contraction 来减少一个点，缩小问题范畴，不致影响 Min-Cut 的权重。

1. 重复下面这件事V-1次，直到图上剩下两个点，刚好可以作出一个Cut：

甲、随机取图上一条边：猜测这条边不在Min-Cut上、这条边的两个端点在Min-Cut同侧。

乙、Contraction：合并这条边的两个端点。

这个演算法近似于「 Minimum Spanning Tree: Kruskal's **Algorithm** 」：每次都选择一条边，让这条边的两个端点相连在一起。唯一的差异是 Karger's **Algorithm** 是随机地选择一条边，而 Kruskal's **Algorithm** 是选择权重最小（或最大）的边。

正确率：

既然是随机演算法，就得计算一下正确率了！下面这段正确率的证明，是我从论文中读到的。我觉得这个证明非常吊诡，不知道我是否有理解错误：

假设一张图的 Min-Cut 至少有 c 条边。然后，令图上每一个点至少都连着 c 条边，才能使这张图的任一个 Cut 至少都有 c 条边、任一个 Cut 都可能成为 Min-Cut 。

根据此设定，可推导出这张图上至少共有 c*V/2 条边。 V 是点的总数。

基于方才的假设，随机从图上选择一条边，这条边在 Min-Cut 上的机率至多是 c / (c*V/2) = 2/V ，不在 Min-Cut 上的机率至少是 1-2/V 。

Karger's Algorithm 每个步骤所选到的边，都必须不是 Min-Cut 上的边，结果才会正确。每个步骤都会减少图上的一个点，推得 Karger's **Algorithm** 的正确率至少是 [1-2/V] * [1-2/(V-1)] * … * [1-2/4] * [1-2/3] = 1/C{V,2} = Ω(1/V^2) = Ω(V^-2) 。

根据这个正确率，只要进行 V^2 次以上的 Karger's **Algorithm** ，结果就会相当准确了！

Min-Cut: Karger-Stein Algorithm：

Karger-Stein Algorithm 是 Karger's **Algorithm** 的加强版。