KVCC 大作业实验报告

涂奕腾 2020201018

一、任务 1：最小(点)割

在网络流部分，我选择使用链式前向星存图以方便网络流算法使用：

```
typedef struct{
    int tot = 1, head[N], cur[N];
    struct {int next, from, to, cap;} edge[M];
    void init(){
        tot = 1;
        memset(head, 0, sizeof(head));
    }
    void addedge(int from, int to, int cap){
        edge[++tot].next = head[from];
        edge[tot].from = from;
        edge[tot].to = to;
        edge[tot].cap = cap;
        head[from] = tot;
    }
}Graph;
```

不同于 PPT 中给出的算法，在求最小点割时我没有使用反图，而是采取了常用的“拆点”法：

将点 I 拆成入点 i 和出点 i+n，除了源点和汇点以外在 i 和 i+n 之间连一条流量为 1 的边，而对于原图中的边 U-V 则在 U 的出点 u+n 和 V 的入点 v 之间连一条流量为 inf 的边表示这条边不能被割掉，因此只有除源点和汇点外的其他点的入点和出点之间所连的边才可能会被割掉，这样一来就可以将求最小点割转化为求最小边割的问题。

而另一方面，最小边割问题则可以由最大流最小割定理使用网络流算法解决。我是用的网络流算法是当前弧优化的 Dinic 算法。找割边则是在残量网络上从源点 dfs 染色直到找到一条满流的边。dfs 完成后枚举每条边，若边的一个端点被染色而另一端点未被染色，则该边是一条割边。

该部分算法虽然代码较长，但以模板为主，不做赘述，主要代码如下，完整代码见/code/low-cut.cpp(注：此算法只支持连通图)：

```
struct LOC_CUT{
Graph g;
int d[N], vis[N], maxflow, s, t;
vector<int> cut;

LOC_CUT(int _s, int _t) : s(_s), t(_t){init();}
LOC_CUT(int _s, int _t, Graph _g) : s(_s), t(_t), g(_g){}

void init(){
```

```
    maxflow = 0;
    g.init();
}

void add(int from, int to, int cap){
    g.addedge(from, to, cap);
    g.addedge(to, from, 0);
}

bool bfs(){
    memset(d, 0, sizeof(d));
    queue<int> q; q.push(s);
    d[s] = 1; g.cur[s] = g.head[s];
    while(!q.empty()){
        int u = q.front(); q.pop();
        for(int i = g.head[u]; i; i = g.edge[i].next) {
            int v = g.edge[i].to;
            if(g.edge[i].cap && !d[v]) {
                g.cur[v] = g.head[v];
                d[v] = d[u] + 1;
                q.push(v);
            }
        }
    }
    return d[t];
}

int dfs(int u, int in){
    if(u == t || in == 0) return in;
    int out = 0;
    for(int i = g.cur[u]; i && in; i = g.edge[i].next){
        g.cur[u] = i;
        int v = g.edge[i].to;
        if(g.edge[i].cap && d[v] == d[u] + 1){
            int res = dfs(v, min(g.edge[i].cap, in));
            g.edge[i].cap -= res, in -= res, g.edge[i ^ 1].cap += res, out += res;
            if(in == 0) break;
        }
    }
    if(out == 0) d[u] = 0;
    return out;
}

void dinic(){
    while(bfs())
        maxflow += dfs(s, inf);
```

```cpp
}

void dfs_cut(int u){
    vis[u] = 1;
    for(int i = g.head[u]; i; i = g.edge[i].next){
        int v = g.edge[i].to;
        if(!vis[v] && g.edge[i].cap) dfs_cut(v);
    }
}

bool find_cut(){
    dinic();
    if(maxflow >= inf) return false;
    memset(vis, 0, sizeof(vis));
    dfs_cut(s);
    for(int i = 2; i <= g.tot; i += 2){
        int u = g.edge[i].from, v = g.edge[i].to;
        if(u + n != v) break;//根据建图时的顺序可以提前退出
        if(vis[u] && !vis[v]) cut.push_back(u);
    }
    return true;
}
};
```

二、任务 2：全局最小(点)割

这里主要参照 PPT 中给出的的方法，先找到图上最小度数的点为源点(记为 s)，先计算 s 是否与其他所有点之间有最小割，再找 s 的邻居节点之间是否有最小割，关键代码如下，完整代码见/code/global_cut.cpp(注：此算法只支持连通图)：

```cpp
vector<int> find_global_cut(const set<int>& V){
    int min_deg = inf, s = 0, t = 0;
    for(auto u : V){
        int deg = 0;
        for(auto v : Adj[u]) if(V.find(v) != V.end()) ++deg;
        if(deg < min_deg)
            min_deg = deg, s = u;//找一个最小度点
    }
    for(auto x : V){
        if(s == x) continue;
        vector<int> cut = getcut(s, x, V);
        if(cut.size()) return cut;
    }
    for(int i = 0; i < Adj[s].size(); ++i){
        if(V.find(Adj[s][i]) == V.end()) continue;
        for(int j = i + 1; j < Adj[s].size(); ++j){
```

```
            if(V.find(Adj[s][j]) == V.end()) continue;
            vector<int> cut = getcut(Adj[s][i], Adj[s][j], V);
            if(cut.size()) return cut;
        }
    }
    return {};
}
```

三、任务 3：KVCC 基础解法

首先是 k-core 的求法。这里我没有使用 ppt 上的算法而是采取了一种类似于拓扑排序的思想，维护一个队列，首先将所有度数小于 k 的点入队并标记；对于队列中的点枚举其出边，所连的点度数-1，若所连的点未标记且度数小于 k 则将其标记并入队，最终所有未被标记的点即为 k-core 中的点

```
//提取 k-core 中的点和边
    vector<int> deg;
    vector<bool> vist(n + 1, false);
    for(int i = 0; i <= n; ++i) deg.push_back(E[i].size());
    queue<int> kcore_q;
    for(int i = 0; i <= n; ++i) {
        if(!deg[i]) vist[i] = true;
        else if(deg[i] < k) kcore_q.push(i), vist[i] = true;
    }
    while(!kcore_q.empty()){
        int u = kcore_q.front(); kcore_q.pop();
        for(auto v : E[u])
            if(!vist[v] && --deg[v] < k)
                kcore_q.push(v), vist[v]=true;
    }
```

随后对提取出来的 k-core 重新建图，对节点重新标号，重新连边：

```
//重新标号建立新图
    set<int> V;
    int cnt = 0;
    for(int i = 0; i <= n; ++i)
        if(!vist[i])
            f1[i] = ++cnt, f2[cnt] = i, V.insert(cnt);
    vector<int> *Adj = new vector<int>[cnt + 1];
    for(int u = 0; u <= n; ++u){
        if(vist[u]) continue;
        for(auto v : E[u]){
            if(vist[v] || v > u) continue;
            int u1 = f1[u], v1 = f1[v];
            Adj[u1].push_back(v1);
            Adj[v1].push_back(u1);
```

```
        }
    }
    n = cnt;
```

接下来则是用 dfs 求出不同的 k-core 连通块，对于每个连通块求全局最小割，若无最小割则该连通块就是 k-core；反之则进行 overlap-partition，同样是通过 dfs 找出不同的分割块进行递归：

```
void dfs_connected_subgraph (int u, set<int> &tmp, vector<int>* &Adj, vector<bool>&
vis){//找极大联通子图
    vis[u]=1; tmp.insert(u);
    for(auto v : Adj[u])
        if(!vis[v])
            dfs_connected_subgraph(v, tmp, Adj, vis);
}

void dfs_overlap_partition(int u, const set<int>& sV, vector<int>* &Adj, const set<int>&
cut, set<int>& tmp, vector<bool>& vis){
    vis[u] = 1; tmp.insert(u);
    for(auto v : Adj[u])
        if(!vis[v] && sV.find(v) != sV.end() && cut.find(v) == cut.end())
            dfs_overlap_partition(v, sV, Adj, cut, tmp, vis);
}

    vector<bool> vis1(n+1, false);
    for(int i = 1; i <= n; ++i){
        if(vis1[i]) continue;
        set<int> sV;
        dfs_connected_subgraph(i, sV, Adj, vis1);//找 k-core 连通块
        if(sV.size() <= k) continue;
        GLOBAL_CUT gc(n, sV, Adj);
        set<int> cut = gc.find_global_cut();//全局最小割
        if(cut.empty()){
            set<int>tmp;
            for(auto x : sV) tmp.insert(f2[x]);
            ans.push_back(tmp);//转化
        }
        else{//overlap partition
            vector<bool> vis2(n+1, false);
            for(auto x : sV){
                if(cut.find(x) != cut.end() || vis2[x]) continue;
                set<int> pV;
                dfs_overlap_partition(x, sV, Adj, cut, pV, vis2);
                for(auto y : cut) pV.insert(y);//建立 overlap partition 后的新子图
                vector<int>* pAdj = new vector<int>[n + 1];
                for(auto x : pV)
                    for(auto y : Adj[x])
```

```
                        if(pV.find(y) != pV.end())
                            pAdj[x].push_back(y);
                    vector<set<int> > res = kvcc(n, pAdj);
                    for(auto v : res){
                        set<int> tmp;
                        for(auto x : v) tmp.insert(f2[x]);
                        ans.push_back(tmp);
                    }
                }
            }
        }
```

最终基础解法的完整代码见/code/kvcc-basic

四、任务 4、5：KVCC 优化解法

这一部分的优化主要是针对求全局最小割部分的算法。对于优化 2(Neighbor Sweep using Vertex Deposit)这一部分直接参考 ppt 上的代码即可：

```
void sweep(int u){
    pru[u] = true;
    for(auto v : Adj[u]){
        if(pru[v]) continue;
        ++deposit[v];
        if(check_ssv(u) || deposit[v] >= k) sweep(v);
    }
}
```

更多的问题主要是针对优化 1(Neighbor Sweep using Side-Vertex)

首先是论文所提供的方法：根据文中的 theorem4 和 definition10 我们可以以 O(deg(u)^2)的时间复杂度判断一个点是否是 strong side vertex

THEOREM 4. *A vertex u is a side-vertex if $\forall v, v' \in N(u)$, either $(v, v') \in E$ or $|N(v) \cap N(v')| \geq k$.*

DEFINITION 10. (STRONG SIDE-VERTEX) *A vertex u is called a strong side-vertex if it satisfies the conditions in Theorem 4.*

```
bool check_ssv(int u, vector<int>* &Adj){
    for(int i = 0; i < Adj[u].size(); ++i){
        for(int j = i + 1; j < Adj[u].size(); ++j){
            int v1 = Adj[u][i], v2 = Adj[u][j];
            set<int> s1, s2, s3;
            for(auto x : Adj[v1]) s1.insert(x);
            for(auto x : Adj[v2]) if(s1.find(x) != s1.end()) s2.insert(x);
            if(s1.find(v2) == s1.end() && s2.size() < k) return false;
        }
```

```
    }
    return true;
}
```

同时根据论文，进行 overlap partition 后我们不需要逐个判断划分出来的子图中的每个点，只需要判断满足如下条件的点即可：

> Based on Lemma 11 and Lemma 12, in a graph G_i partitioned from graph G by vertex cut $\mathcal{S}$, we can reduce the scope of strong side-vertex checks from the vertices in the whole graph G_i to the vertices u satisfying following two conditions simultaneously:
> - u is a strong side-vertex in G; and
> - $N(u) \cap \mathcal{S} \neq \emptyset$.

只需对 overlap partition 的部分进行插入如下判断即可：

```
bool check_neighbor_in_cut(const vector<int>& Adj, const set<int>& cut){
    for(auto x : Adj)
        if(cut.find(x) != cut.end())
            return true;
    return false;
}


for(auto u : pV){
    if(SSV.find(u) != SSV.end() && check_neighbor_in_cut(pAdj[u], cut))
        if(check_ssv(u, pAdj))
            pssv.insert(u);
}
```

但是在该算法下需要逐个判断全图 k-core 中的每一个节点，每个节点的判断都要有 O(deg(u)^2)的复杂度，特别是点数较少的稠密图会造成比较大的时间消耗。因此我又注意到了论文中的 lemma8：

> LEMMA 8. *Given a graph G, a vertex u is a side-vertex if and only if $\forall v, v' \in N(u)$, $v \equiv^k v'$.*

Lemma8 是一个充要条件，其中每个 side vertex 的判断仅为 O(deg(u))的复杂度。在图中点数较小时，我们就不选择一开始计算出全图的 strong side vertex，而只是在需要的时候对某个点是否为 side vertex 进行判断，因此我将其定义成了 global_cut 的类成员函数。在该方法下判断 side vertex 的代码如下：

```
bool check_ssv(int u){
    for(int i = 0; i < Adj[u].size(); ++i){
        for(int j = i + 1; j < Adj[u].size(); ++j){
            int v1 = Adj[u][i], v2 = Adj[u][j];
            if(!check_neighbor(v1, v2)) return false;
        }
```

```
    }
    return true;
}
```

在最终提交的代码中，后面一种针对点数较小时的方法为/code/kvcc-sweep1，而原文的方法为/code/kvcc-sweep2

最后回到求全局最小割的函数主体部分，以 kvcc-sweep1 为例。按照论文的要求，第一步时根据到源点的距离由远及近地枚举当前图上的所有点，这里每个点到源点距离的计算我使用的是双端队列优化的 spfa：

```
void spfa(int s){
    deque<int> q;
    for(int i = 0; i <= n; ++i) dist[i] = inf, pru[i] = false;
    q.push_back(s); dist[s] = 0, pru[s] = true;
    while(!q.empty()){
        int u = q.front(); q.pop_front();  pru[u] = false;
        for(auto v : Adj[u])
            if(dist[v] > dist[u] + 1){
                dist[v] = dist[u] + 1;
                if(pru[v]) continue;
                pru[v] = true;
                if(!q.empty() && dist[v] < dist[q.front()]) q.push_front(v);
                else q.push_back(v);
            }
    }
    fill(pru.begin(), pru.end(), false);
}
```

随后我按照距离将所有点放到一个优先队列中。接着就是先对源点 sweep 操作，然后对图上所有点按照距离的降序进行枚举，满足 Neighbor Sweep using Vertex Deposit 优化条件的点可以直接跳过，否则将其设为汇点求最小割并进行 sweep 操作。若未求出最小割，则按照 Neighbor Sweep using Side-Vertex 优化的要求，如果源点是 side vertex 则直接返回无全局最小割，否则同朴素方法一样检验原来源点所有邻居点对之间的最小割：

```
set<int> find_global_cut(){
    int min_deg = inf, s = 0, t = 0;
    for(auto u : V){
        int deg = Adj[u].size();
        if(deg < min_deg)  min_deg = deg, s = u;
    }
    priority_queue<pair<int, int> >pq;
    spfa(s);
    for(auto x : V)
        if(dist[x] > 1 && dist[x] != inf)
            pq.push(make_pair(dist[x], x));
    sweep(s);
```

```
    while(!pq.empty()){
        int v = pq.top().second; pq.pop();
        if(pru[v]) continue;
        set<int> cut = getcut(s, v, V);
        if(!cut.empty()) return cut;
        sweep(v);
    }
    if(!check_ssv(s)){
        for(int i = 0; i < Adj[s].size(); ++i)
            for(int j = i + 1; j < Adj[s].size(); ++j){
                int w1 = Adj[s][i], w2 = Adj[s][j];
                set<int> cut = getcut(w1, w2, V);
                if(!cut.empty()) return cut;
            }
    }
    return {};
}
```

五、实验结果

由于 stanford 和 google 数据集本身是有向图且使用 c++读入时较为麻烦，我将其重新规范并删去了多于的边。所有的测试数据集为 code 文件夹下的 dblp.txt, _google.txt, _stanford.txt。

由于内存空间有限，本地无法跑出所有的结果。经过实验，在以下数据集和 k 值时 kvcc-sweep1.cpp 和 kvcc-sweep2 可以在较短时间内得出正确结果，运行时间如下，完整结果见 result 文件夹：

数据集及参数	kvcc-sweep1	kvcc-sweep2
dblp, k = 40	1.23s	29.06s
dblp, k = 35	1.406s	30.24s
dblp, k = 30	1.821s	30.84s
dblp, k = 25	3.226s	37.18s
dblp, k = 20	7.884s	45.33s
google, k = 40	77.44s	4.561s
google, k = 35	59.98s	4.499s
google, k = 30	86.63s	47.39s
google, k = 25	142.5s	122.4s
stanford, k = 40	181.6s	532.1s

(在 dblp 中 N=1e5,M=4e6；在 google 和 stanford 中 N=1e6,M=2e7)

由此可见，在点数较少时(dblp, stanford)使用 lemma8 的计算复杂度更低的充要条件判断 side vertex 进行优化速度更快；点数较多时(google)预处理 strong side vertex 会明显更快。