程式設計研討專題 4

110403518 林晉宇

一、題意

UVA 10480 Sabotage，給 n 個點(n<=50)，m 條邊(m<=500)，求將 1 跟 2 分開的最小割，並輸出割包含的邊集合。這是一題解 minimum-cut (maximum-flow) 的問題，第一種作法會先使用一般的 flow 方法去解題，第二種方法實作隨機合併的做法，最後比較兩者的時間及第二種 Randomize 的準確率。

二、使用 Flow 解題

- **Max-flow / Min-cut 演算法:**

  用來解最大流最小割(max-flow min-cut)的演算法主要有三種:

  1. **Ford-Fulkerson** 為三者中最早提出， 時間複雜度為 **O(fm)** (f 為 max-flow, m 為邊數)
  2. **Edmonds-Karp**，可以視為改進版的 Ford-Fulkerson，時間複雜度為 **O(m^2 n)**，不會受到 max-flow 值所影響
  3. **Dinic**，複雜度為 **O(m n^2)**，一般邊的數量都大於點的數量，所以通常這比 Edmonds-Karp 快。

  這邊我使用 **Dinic** 做法，可以直接求出 max-flow(min-cut)值，而要判斷哪些邊為割的集合，就看邊的兩個端點是不是在同一群即可，如果不是(即原邊的一點跟 1 同一群，另一點跟 2 同一群)，則屬於割的邊集合，輸出該邊。

- **Dinic 步驟:**

  1. 一開始 residual graph(殘量網路)即為原圖。
  2. 循環:
     I. 用 bfs 去找當前 residual graph 的 level graph(分層網路)。
     II. 用 dfs 去找當前 level graph 上的 augmenting path(擴增路

徑)，即一條可以從源點走向匯點的路徑。[**如果找不到擴增路徑，就終止循環**]。

III. 更新 residual graph，包含更新邊權、刪除飽和(邊權為 0)的邊、建反向流量。回到第一步驟。

- Pseudo code:

1. Dinic

```
int dinic()
    ans <- 0
    while(bfs()) //有增廣路徑
        while(d <- dfs(s,inf)) //去找增廣路徑
            ans+=d;
    return ans;
```

2. bfs 找分層圖(level graph)

```
ll bfs() //find level graph
    dis[] <- {-1}
    dis[1]=0
    queue q
    q.push(1)
    while(!q.empty())
        ll u=q.front(); q.pop();
        for(遍歷 u 點的邊集合)
            ll v=edge[i].v
            if(dis[v]==-1&&edge[i].flow)
                dis[v]=dis[u]+1; //dis 紀錄該點為分層圖的第幾層
                q.push(v);
    return dis[t]!=-1; //如果等於-1 代表 就沒有 augmenting path 可以直接終止循環
```

3. dfs 找擴增路徑(augmentint path)

```
ll dfs(ll u,ll flow) //find augmenting path
    if(u==t)    return flow; //如果當前該點為匯點 直接 return
    for(遍歷 u 的邊集合)
        ll v=edge[i].v;
        if(dis[v]==dis[u]+1&&edge[i].flow)
            ll d=dfs(v,min(edge[i].flow,flow));
```

```
            if(d>0)
                edge[i].flow-=d;
                edge[i^1].flow+=d;
                return d;
    return 0;
```

三、使用 Randomized 做法解題

- **Karger algorithm:**

  Karger 是隨機算法，相比前面的 flow 演算法，優點是速度快，實現簡單，但未必找到的解一定是最小割。這裡用 karger 演算法來處理 s-t cut 的問題。

- **Karger 步驟:**

  1. 一開始 Contracted graph 即為原圖。
  2. 當還有超過 2 點在 contracted graph 上時:
     I. 隨機在 Contracted graph 挑一個邊(u, v)，但此邊不能是(s, t)。
     II. 將 u 跟 v 兩點合併成一點，並更新 contracted graph。
     III. 移除自環。
  3. 最後剩下的兩點所連的邊即為最小割的可能答案。

- **實作方式:**

  寫額外的 struct 來存圖的邊(edge)，而每當有兩點合併時，用並查集(Disjoint set union)去做處理，有加上路徑壓縮(data compression)以及按秩合併(union by rank)。

- **Pseudo code:**

  1. Karger

```
int Karger(struct Graph* graph)
{
    並查集初始化(將每個點的祖先設成自己)
    v <- 點數
    while(v>2)
    {
```

```
        隨機選一邊
        if(邊的兩點屬於同一集合)     continue;
        else if(邊的兩點包含1跟2) continue;
        else
        {
            v--;
            union();將兩點集合合併
        }
    }
    cutedge <- 0;
    for 遍歷所有邊
    {
        subset1 <- 當前邊所屬集合1
        subset2 <- 當前邊所屬集合2
        if(兩集合不同)     cutedge+=邊權;
    }
    return cutedge;
}
```

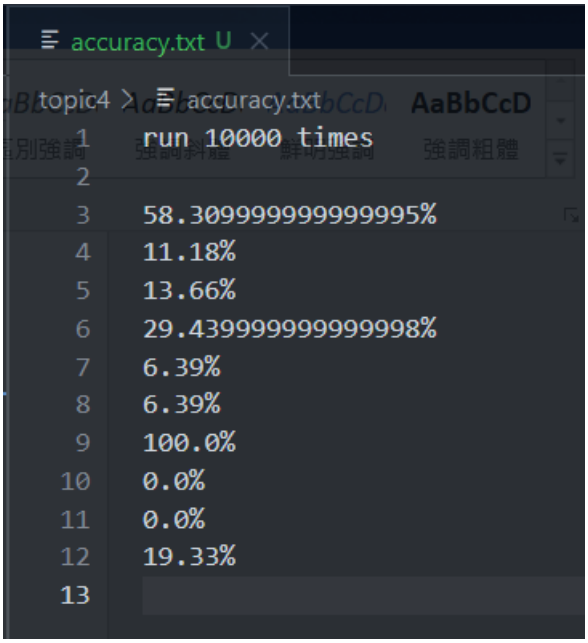
四、比較

- **Randomized 的準確率:**

  測資使用 UVA 10480 原題的測資，使用網路上教的方法(連結)，寫 python 把隱藏測資偷過來(測資連結)，UVA 這題總共有十筆資料，以下對比不同資料的準確率。(以 dinic 算出的答案作為標準答案，拿 karger 算出的結果來比對)。

資料	點數/邊數	邊權範圍	準確率(karger 跑 10000 次)
第一筆	5/10	100 以內	58.3%
第二筆	5/6	100 以內	11.18%
第三筆	5/6	100 以內	13.66%
第四筆	5/4	100 以內	29.44%
第五筆	5/8	100 以內	6.39%
第六筆	5/8	100 以內	6.39%

第七筆	2/1	100 以內	100%
第八筆	50/497	非常極端大部分 100 以內，但少數大至 40000000	0%
第九筆	50/497	非常極端大部分 100 以內，但少數大至 40000000	0%
第十筆	50/500	五位數以內	19.33%



可以看出 karger 隨機算法在面對點數及邊權小的資料時，可能還有機會算出最小割，但基本上只要少數邊權非常極端或點數及邊數增加，準確率就會慘不忍睹。

- **時間比較:**

  使用 python 產生隨機測資(連通圖)，以下為實驗的結果:

資料筆數	Flow (Dinic)	Randomized(Karger)
200 筆	0.035 秒	0.021 秒
2000 筆	0.18 秒	0.144 秒

20000 筆	1.795 秒	1.225 秒

可以看出 Karger 整體比 flow 快大約 30%左右。

五、結論

這次報告讓我對 flow 中算最大流最小割的三種演算法有更深入的了解，也實際時做了 dinic 做法，而透過與隨機演算法 karger 的比較，可以發現雖然 karger 可能比較容易實作及速度較快，但準確率蠻低的，除非點數、邊數及邊權範圍較小及相近，準確率才有可能比較高。

六、資料來源

1. Youtube - 13-1: 网络流问题基础 Network Flow Problems
2. Youtube - 13-2: Ford-Fulkerson Algorithm 寻找网络最大流
3. Youtube - 13-3: Edmonds-Karp Algorithm 寻找网络最大流
4. Youtube - 13-4: Dinic's Algorithm 寻找网络最大流
5. Geeksforgeeks - Karger's algorithm for Minimum Cut