算法基础 HW8

PB18111697 王章瀚

2020 年 12 月 24 日

1.

(欧拉回路) 强连通有向图 $G = (V, E)$ 中的一个欧拉回路是指一条遍历图 G 中每条边恰好一次的环路. 不过这条环路可以多次访问一个结点.

a.

证明: 图 G 中有一条欧拉回路当且仅当对于图中的每个结点 v, 有 $in - degree(v) = out - degree(v)$.

(必要性)若有一条欧拉回路, 则该回路上的任意一个点相邻的边都分别为一个入边和一个出边, 此时, 对该点来说, 必然有 $in - degree(v) = out - degree(v)$
而该性质对每个点都成立, 因此最后加起来仍然有 $in - degree(v) = out - degree(v)$.

(充分性)若对每个点 v 都有 $in - degree(v) = out - degree(v)$, 我们总是可以从一个点 u 出发, 向任意的结点前进构造路径, 然后最终回到 u.
否则, 假设构造的最长路径最终只能停在点 v, 则除了最后一次, 之前每次访问点 v 之后必然从点 v 出去, 那么 最后这次进入点 v, 其使用过的入边数量一定比使用过的出边数量多 1, 而根据 $in - degree(v) = out - degree(v)$, 我们知道我们还可以从 v 出发, 接上下一个顶点, 构成更长的路径. 这就与假设矛盾.
因此我们必然可以构造出一个欧拉回路. □

b.

给出一个复杂度为 $O(E)$ 的算法来找出图 G 的一条欧拉回路.

就如上面描述的, 只要当前结点有未使用的出边, 就走上去. 下面给出稍微严谨一点的说法:

Algorithm 1 EULER-CIRCUIT

Require: 图 $G = (V, E)$

```
1: function EULER-CIRCUIT(G)
2:     任选起点 u, (u 应至少有一出边)
3:     v = u
4:     初始化路径 path
5:     while edge = v.next_out_edge() != NULL do    ▷ 这里 next_out_edge() 应当用 yield 之类的协程方式(如python)实现, 以保证每次取到下一个
6:         path.append(edge)
7:         v = edge.j                                ▷ edge.j 表示 edge 指向的下一个顶点
8:     return path
```

由于每个边最多访问一次, 且整个过程以访问边为根据, 因此显然其时间复杂度恰为 $O(E)$.

2.

(套利交易) 套利交易指的是使用货币汇率之间的差异来将一个单位的货币转换为多于一个单位的同种货币的行为. 例如, 假定 1 美元可以购买 49 印度卢比, 1 印度卢比可以购买 2 日元, 1 日元可以购买 0.0107 美元. 那么通过在货币间进行转换, 一个交易商可以从 1 美元开始, 购买 49*2*0.0107=1.0486 美元, 从而获得 4.86% 的利润.
假定给定 n 种货币, $c_1, c_2, \cdots, c_n$ 和一个 $n \times n$ 的汇率表 R, 一个单位的 c_i 货币可以买 $R[i, j]$ 单位的 c_j 货币.

a.

给出一个有效的算法来判断是否存在一个货币序列 $\langle c_{i_1}, c_{i_2}, \cdots, c_{i_n} \rangle$, 使得

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdot \cdots \cdot R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1$$

请分析算法运行时间

不妨考虑取对数后的情景,

$$-\log R[i_1, i_2] - \log R[i_2, i_3] - \cdots - \log R[i_{k-1}, i_k] - \log R[i_k, i_1] < 0$$

也就是我们可以转换一下原表, 对原表 R 的每个元素取负对数得到表 Q.
然后以这些为该完全图的边权, 并运行一遍 Bellman-Ford 算法, 即可找到其中可能存在负环. 如果存在负环, 说明存在一个环使得上式成立, 那么也就是一个满足题意的环. 至于是否存在, 则可以直接由 Bellman-Ford 算法的返回值给出, 如果返回 true, 就存在, 返回 false 就不存在.
显然, Bellman-Ford 算法的时间复杂度是 $O(VE) = O(V^3)$(因为这里是完全图)
下面是 Bellman-Ford 算法

Algorithm 2 BELLMAN-FORD

Require: 图 $G = (V, E)$
Require: w 为取了负对数的权重.
Require: s 为起始点, 任选其一即可.

```
1: function BELLMAN-FORD(G, w, s)
2:     INITIALIZE-SINGLE-SOURCE(G, s)
3:     for i = 1 to |G.V| − 1 do                    ▷ 包括内层总共 O(VE)
4:         for each edge(u, v) ∈ G.E do
5:             RELAX(u, v, w)
6:     for each edge(u, v) ∈ G.E do
7:         if v.d > u.d + w(u, v) then
8:             return true                          ▷ 表明存在负环, 即存在对应套利方法
9:     return false                                 ▷ 表明不存在负环, 即不存在对应套利方法
```

b.

给出一个有效算法来打印出这样一个序列(如果存在这样一种序列), 分析算法的运行时间.

刚才 (a) 中的算法已经能确认是否有这样的序列了. 现在我们只要找出这样的序列. 根据前面的结果, 在运行完 Bellman-Ford 算法的第一步后, 如果有可以继续 RELAX 的边, 那么就说明图中存在有负环, 这时候, 必然还可以

进行 Relax.
如果我们能够对所有可以 Relax 的边进行再一个 Relax 操作, 那么就可以更新负环上的边相应的顶点的 d 值为更小的值. 此时, 倘若紧挨着的点也可以因为其前驱的更新而接着更新.

- 这句话听起来有点抽象, 举个例子: 比如一个环 $u \to v \to w \to u$, 那么这时候如果我因为 (u, v) 满足了 Relax 的条件 $v.d > u.d + w(u, v)$, 我就可以把 $v.d$ 更新为更小的 $u.d + w(u, v)$, 此时, 只需要再进行一轮更新, 就可以类似地把 $w.d$ 更新为更小的 $v.d + w(v, w)$, 再一轮, 可以把 $u.d$ 更新为更小的 $w.d + w(w, u)$.

因此, 最多不超过 $|V| - 1$ 轮的更新内, 我们可以遍历到某个环的所有边集.
于是, 我们只需要把边集构造为一个图并找出一个环即可(DFS之类的, 不超过 $O(E + V)$), 乘上前面的 $O(EV) = O(V^3)$ 就会得到 $O(V^4)$, 总的时间复杂度为 $O(V^4)$(因为这里是完全图, $|E| = |V|^2$)
下面是算法:

Algorithm 3 Bellman-Ford-Find-Neg-Circle

Require: 图 $G = (V, E)$
Require: w 为取了负对数的权重.
Require: s 为起始点, 任选其一即可.

function Bellman-Ford-Find-Neg-Circle(G, w, s)
    Initialize-Single-Source(G, s)
    for $i = 1$ to $|G.V| - 1$ **do** ▷ 包括内层总共 $O(VE)$
        for each edge$(u, v) \in G.E$ **do**
            Relax(u, v, w)
    for each edge$(u, v) \in G.E$ **do**
        if $v.d > u.d + w(u, v)$ **then**
            goto Find-Neg-Circle ▷ 表明存在负环, 即存在对应套利方法
    return false ▷ 表明不存在负环, 即不存在对应套利方法
    Find-Neg-Circle:
    初始化可能的负环边集构成的图 $negG$ ▷ 其实维护过程中它一直是树, 因为一旦找到了环就退出
    for $i = 1$ to $|G.V| - 1$ **do** ▷ 这个外层循环一定不会超过 $|E|$ 次
        for each edge$(u, v) \in G.E$ **do** ▷ 而内层循环是 $O(E)$ 的
            Relax(u, v, w)
            $negG.add\_edge((u, v))$
            if 存在环 **then** ▷ $O(V + E)$
                return 返回这个环

3.

假定在一个权重函数为 ω 的有向图上运行 $Johnson$ 算法. 证明: 如果图 G 包含一条权重为 0 的环路 c, 那么对于环路 c 上的每条边 (u,v), $\hat{w}(u,v)=0$

根据定义 $\hat{w}(u,v)=w(u,v)+h(u)-h(v)\geq 0$ 对任意的 u,v 都成立.
如果环路 $u_1,u_2,\cdots,u_k,u_{k+1}=u_1$ 权重为0, 那么 $\sum\limits_{i=1}^{k} w(u_i,u_{i+1})=0$
因此

$$\begin{aligned}\sum_{i=1,i\neq j}^{k}\hat{w}(u_i,u_{i+1}) &= \sum_{i=1,i\neq j}^{k}(w(u_i,u_{i+1})+h(u_i)-h(u_{i+1}))\\ &=\sum_{i=1}^{k}w(u_i,u_{i+1})+\sum_{i=1}^{k}(h(u_i)-h(u_{i+1}))-(w(u_j,u_{j+1})+h(u_j)-h(u_{j+1}))\\ &=-(w(u_j,u_{j+1})+h(u_j)-h(u_{j+1}))\end{aligned}$$

又$\hat{w}(x,y)\geq 0$, 故

$$0\leq\sum_{i=1,i\neq j}^{k}\hat{w}(u_i,u_{i+1})=-(w(u_j,u_{j+1})+h(u_j)-h(u_{j+1}))$$

即

$$0\leq-(w(u_j,u_{j+1})+h(u_j)-h(u_{j+1}))=-\hat{w}(u_j,u_{j+1})\leq 0$$

故

$$\hat{w}(u_j,u_{j+1})=0$$

这对所有的 j 都成立, 因此对于环路 c 上的每条边 (u,v), $\hat{w}(u,v)=0$ □

4.

(最大流更新) 设 $G = (V, E)$ 是一个源结点为 s, 汇结点为 t 的流网络, 其容量全部为整数值. 假定我们已经给定 G 的一个最大流.

a.

如果将单条边 $(u, v) \in E$ 的容量增加一个单位, 请给出一个 $O(V + E)$ 时间的算法来对最大流进行更新.

实际上只要走一步 Ford-Fulkerson 算法即可, 其时间复杂度恰为 $O(E + V)$. 下面细说之:

- 如果 (u, v) 在容量增加之前, 对应在残存网络里的同向边剩余流量至少为 1, 那么这时 (u, v) 多增加 1 个容量单位也不会改变找不到增广路径的事实. 否则, 原本的残存网络中也能找到增广路径, 只不过流量小了 1.
- 如果 (u, v) 在容量增加之前, 对应在残存网络里的同向边剩余流量至少为 0, 那么这时我们就做一遍 Ford-Fulkerson 算法, 从而尝试更新含有 (u, v) 的增广路径.(必然含有, 否则原残存网络中仍存在增广路径). 而更新完毕后, 必然已经是最大流, 这是因为:
  - 如果刚才的更新有让流量增加(最多加了1), 我们可以在原图中选出一个最小割, 其将 (u, v) 割开(这是因为 (u, v) 满了), 这时, 在容量增加之后, 最小割对应的容量多了 1, 这时由于刚才所述过程做了更新使得总流量 +1, 因此此时总流量已经达到该割容量, 故达到了最大流.
  - 如果没有, 则说明没有增广路径, 已经是最大流

具体算法可以这样写出:

Algorithm 4 UPDATE-MAXFLOW-WHEN-INCREASE

```
DFS 找一条增广路径
if 找不到 then
    结束
else
    按 Ford-Folkerson 算法更新最大流
```

结合上述分析该算法显然是 $O(V + E)$ 的

b.

如果将单条边 $(u, v) \in E$ 的容量减少一个单位, 请给出一个 $O(V + E)$ 时间的算法来对最大流进行更新.

与前面的类似.

- 如果 (u, v) 在容量减少之前, 原该边上流量不足容量-1(直观上, 该边没用满), 那么这时 (u, v) 减小 1 个容量单位后, 仍然无法在原残存网络中找到一个增广路径. 否则, 原本的残存网络中将也能找到增广路径, 只不过流量多 1.
- 但如果 (u, v) 在容量减少之前, 原该边上流量恰等于其容量-1(直观上, 该边用满了), 那么这时减少其容量就需要将某条沿该边的 path 上所有流量减 1. 这涉及一个 $O(V + E)$ 的 BFS. 而该操作后, 割开 (u, v) 的一个割. 此后, 还要进行一次 Ford-Fulkerson 算法的迭代, 因为刚才的 decrease 可能使得有新的增广路径出现, 但又最多出现一条(以使总流量+1), 否则原流不是最大流:

– 如果有多条, 那么选它们而不选包含 (u, v) 的这条能够使得原本的流量更大

具体算法可以这样写出:

Algorithm 5 UPDATE-MAXFLOW-WHEN-DECREASE

BFS 找到包含 (u, v) 的路径, 令该路径上所有边的流量 -1
DFS 找一条增广路径
if 找不到 **then**
    结束
else
    按 Ford-Folkerson 算法更新最大流

结合上述分析该算法显然是 $O(V + E)$ 的