第三次作业

ピカピカピ

2021 年 11 月 27 日

1 Transpose of a Directed Graph. CLRS 22.1-3

1.1 Adjacency List

见算法 1: AdjacencyListTranspose 算法

Algorithm 1 AdjacencyListTranspose

```
1: procedure AdjacencyListTranspose(Graph g)      ▷ 输入为一个链表数组, 是图的邻接链表
2:     n = |V|
3:     Graph gt                                   ▷ 初始化一个空的链表数组用来存放图的转置
4:     for i from 0 to n do
5:         for j in g[i] do                       ▷ 即遍历 g[i] 链表中的每一个元素
6:             g[j].add(i)                        ▷ 对 g[i] 中的元素 j, 将 i 添加到链表 g[j] 中
7:     return gt
```

对算法 AdjacencyListTranspose, 其遍历了链表数组中每一个链表的每一项, 换言之遍历了图的每一个结点与每一条边, 因而时间复杂度为 $T(n) = O(|E| + |V|)$

1.2 Adjacency Matrix

对有向图而言, 其邻接矩阵必是一个方阵, 因而对表示为邻接矩阵的有向图, 求其转置转化为了求其邻接矩阵的转置

见算法 2: AdjacencyMatrixTranspose 算法

对算法 AdjacencyMatrixTranspose, 其运行了一个两层的嵌套循环来遍历转置矩阵的上三角, 因而时间复杂度为 $T(n) = O(\frac{1}{2} \cdot |V|^2) = O(|V|^2)$

2 Cubic of Directed Graph. CLRS 22.1-5

2.1 Adjacency List

见算法 3: AdjacencyListCubic 算法

Algorithm 2 AdjacencyMatrixTranspose

```
1: procedure AdjacencyMatrixTranspose(Graph g)    ▷ 输入为一个二重数组, 是图的邻接矩阵
2:     n = |V|
3:     for i from 0 to n do
4:         for j from i+1 to n do
5:             tmp = g[i][j]
6:             g[i][j] = g[j][i]
7:             g[j][i] = tmp
8:     return g
```

Algorithm 3 AdjacencyListCubic

```
1: procedure AdjacencyListCubic(Graph g)          ▷ 输入为一个链表数组, 是图的邻接链表
2:     n = |V|
3:     gc = g
4:     for i from 0 to n do
5:         for j in g[i] do
6:             gc[i].add (g[j])                   ▷ 对 g[i] 中元素 j, 将 g[j] 中所有元素添加到 g[i] 链表中
7:             for k in g[j] do
8:                 gc[i].add (g[k])               ▷ 对 g[j] 中元素 k, 将 g[k] 中所有元素添加到 g[i] 链表中
9:     AdjacencyListSimple(gc)
10:    return gc
11: procedure AdjacencyListSimple(Graph g)        ▷ 输入为一个链表数组, 是图的邻接链表
12:    initialize res                             ▷ 初始化一个新的结果邻接矩阵
13:    n = |V|
14:    initialize A                               ▷ 初始化一个值全为 0, 大小为 |V| 的数组
15:    for i from 0 to n do
16:        for j in g[i] do
17:            if A[j] != i then
18:                A[j] = i
19:                res[i].add(j)
20:    return res
```

对算法 AdjacencyListSimple, 其进行多重边的去重, 其遍历图的每一个结点与每一条边, 时间复杂度为 $T(n) = O(|E| + |V|)$

对算法 AdjacencyListCubic, 其首先对每一个结点 u 的每一条边进行遍历, 在对每一条边的遍历过程中, 对该有向边指向的结点 v 执行如下操作: 对所有结点 v', 若存在从 v 到 v' 的长度不超过 2 的路径, 将其添加到结点 u 的链表中. 在每条边的遍历过程中最多可能需要遍历 $|V| \cdot |V|$ 个结点, 因而有时间复杂度 $O(|E| \cdot |V|^2$, 最后调用算法 AdjacencyListSimple 对重复边进行去重, 因而该算法时间复杂度为 $T(n) = O(|E||V|^2) + O(|E||V|^2 + |V|) = O(|E||V|^2 + |V|)$

2.2 Adjacency Matrix

首先定义, 对无权图而言, 邻接矩阵中 $g[i-1][j-1]$ 为 0 代表结点 i 到结点 j 之间不含有一条有向边, 而 $g[i-1][j-1]$ 不为 0 代表结点 i 到结点 j 之间含有一条有向边.

那么图的邻接矩阵 A 的 n 次方 A^n 则会对原始图作出如下改变: 对两结点 u, v, 若存在一条从 u 到 v 的长度不大于 n 的路径, 则添加一条由 u 到 v 的有向边, 换言之邻接矩阵 $g^n[u-1][v-1]$ 不为 0

见算法 4: AdjacencyMatrixCubic 算法

Algorithm 4 AdjacencyMatrixCubic

```
1:  procedure AdjacencyMatrixCubic(Graph g)          ▷ 输入为一个二重数组, 是图的邻接矩阵
2:      gc = g
3:      gc = AdjacencyMatrixMutiple(gc, g)
4:      gc = AdjacencyMatrixMutiple(gc, g)
5:      return gc
6:  procedure AdjacencyMatrixMutiple(Matrix a, Matrix b)          ▷ 输入为两个方阵
7:      n = sqrt(a.length())                          ▷ n 为方阵的长/宽长度
8:      initialize res                                ▷ 初始化结果方阵 res, 值全为 0
9:      for i from 0 to n do
10:         for j from 0 to n do
11:             for k from 0 to n do
12:                 res[i][j] += a[i][k] * b[k][j]
        return res
```

对算法 AdjacencyMatrixMutiple, 其对矩阵执行一个三层嵌套循环来实现方阵乘法, 因而时间复杂度 $T(n) = O(n^3)$

对算法 AdjacencyMatrixCubic, 其两次调用算法 AdjacencyMatrixMutiple, 因而时间复杂度 $T(n) = 2 \cdot O(|V|^3) = O(|V|^3)$

3 Universal Sink. CLRS 22.1-6

见算法 5: 算法 FindUniversalSink

正确性证明如下:

Algorithm 5 FindUniversalSink

```
1: procedure FindUniversalSink(Graph g)          ▷ 输入为一个二重数组, 是图的邻接矩阵
2:     n = |V|
3:     for i from 0 to n do
4:         if g[i][i] == 1 then
5:             return false
6:     i = 0
7:     j = 1
8:     while i <= N and j <= N do
9:         if g[i][j] == 1 then
10:            i = maxi+1, j
11:            j++
12:        else
13:            j = maxi+1, j+1
14:        if i <= N then
15:            if CheckAll(g, i) then
16:                return true
17:            else
18:                return false
19:        else
20:            return false
21: procedure CheckAll(Graph g, v)
22:    n = |V|
23:    for i from 0 to n do
24:        if g[v][i] == 1 then
25:            return false
26:        if i != v then
27:            if g[i][v] == 0 then
28:                return false
29:    return true
```

- 如果 $g[i][j] = 0$, 则 j 不是通用汇点
  这是因为 $g[i][j] = 0$ 说明了没有从 i 到 j 的有向边, 而通用汇点的定义是一定要每个点都要有一条指向他的边, 因此 j 不是通用汇点.

- 接着证明如果 A[i][j]=1, 则 i 不是通用汇点
  这是因为若 $g[i][j] = 1$, 说明有从 i 到 j 的有向边, 所以 i 不是通用汇点

- 最后证明: 如果 A[i][j]=0, 则 i 之前与 j 之前的点都不是通用汇点

  1) 循环不变式: 每次迭代前, i 前和 j 前且不包括 i 的点都不是通用汇点

  2) $i = 0, j = 1$ 时, i 前和 j 前且不包括 i 的点为空, 因此循环不变式成立

  3) 倘若在迭代开始时, 已知 i 前和 j 前且不包括 i 的点都不是通用汇点

  4) 那么当进入循环体后, 若 $g[i][j]$ 为 1, 则说明 i 不是通用汇点. 同时已知 i 前和 j 前且不包括 i 的点都不是通用汇点, 因而执行 $i = max\{i + 1, j\}, j++$ 后仍然保持不变式;
  若 $g[i][j]$ 为 0, 则说明 j 不是通用汇点. 同时已知 i 前和 j 前且不包括 i 的点都不是通用汇点, 若 j 原本小于 i, 那么其需要变化到 $i + 1$, 因而仍保持循环不变式

  5) 因而若 $i < |V| + 1, j = |V| + 1$, 那么 j 前不包括 i 的点都不是通用汇点, 因而只需要检查 i 即可; 若 $i = |V| + 1$, 则没有通用汇点

由此算法正确性得到了证明

4 Counting Simple Paths. CLRS 22.4-2

见算法 6: 算法 FindPath

Algorithm 6 FindPath

```
1: procedure FindPath(Graph g, u, v)          ▷ 输入图的邻接链表以及搜索的两个结点
2:     if u == v then
3:         return 1
4:     else
5:         pathnum = 0
6:         for i in g[u] do                    ▷ 遍历结点 u 的链表
7:             pathnum += FindPath(g, w, v)
8:         return pathnum
```

5 Euler Tour. CLRS 22-3

5.1 a

- 首先证明充分性
  利用反证法. 假设图中存在欧拉回路, 且假设图中有一些定点 v, 其入度和出度不相等. 不妨假设入度大于出度, 那么在从 v 出发并回到 v 多次后, 必然将存在仍未使用的进入边, 这是因为每当经过 v 一次, v 的一条入边和一条出边都被使用, 假设 v 入度为 a, 出度为 b, 那么当经过 v a 次以后, 将存在 $b-a$ 条从 v 出发的边, 然而此时从 v 进入的边都已经被使用, 因而不可能存在欧拉回路, 与前提矛盾, 因而可以证明充分性

- 随后证明必要性
  利用归纳法,

  1) 对于仅有一个结点 v 的图, v 的入度和出度都相等, 那么说明其要么无环, 要么仅有自环, 显然存在欧拉回路

  2) 假设对所有 k 个结点的图, 其 k 个结点的入度和出度相等, 都存在欧拉回路

  3) 那么对存在 $k+1$ 个结点的图, 其 $k+1$ 个结点的入度和出度相等, 取其中一个结点 u, 若其有自环, 显然欧拉回路可以由该结点通过自环来回到该结点, 所有自环都能被使用, 因而可以删去该结点的自环, 此时仍能保证该结点入度和出度相等. 对于该结点所连接的其余的有向边, 为其进行两两配对, 对每一对有向边 e_1, e_2, 假设其分别连接结点 v_1, v_2, 可以将 e_1, e_2 删去, 同时在 v_1, v_2 间建立新的一条有向边, 这样可以保证 v_1, v_2 的入度和出度仍然相等, 同时 u 的入度和出度也相等. 对 u 的每一对有向边都执行相同的操作, 直到 u 的入度和出度都为 0, 此时该含 $k+1$ 个结点的图可以转化为含 k 个结点的图, 因而含有欧拉回路

  4) 因而对所有的图, 若其所有结点的入度和出度都相等, 那么改图存在欧拉回路

5.2 b

见算法 7: 算法 FindEulerCircuit

6 Arbitrage. CLRS 24-3

6.1 a

首先构建图, 将货币作为图中的结点, 每个结点之间的有向边权重取 $-In(R[i,j])$, 这样 $R[i_1,i_2]\cdot R[i_2,i_3]\cdot\cdots R[i_{k-1},i_k]\cdot R[i_k,i_1]>1$ 可以被转化为 $In(R[i_1,i_2]\cdot R[i_2,i_3]\cdots R[i_{k-1},i_k]\cdot R[i_k,i_1])=In(R[i_1,i_2])+In(R[i_2,i_3])+\cdots In(R[i_{k-1},i_k])+In(R[i_k,i_1])<0$, 换言之即转化为检测负权环问题, 因而运用 Bellman-Ford 算法即可解决
见算法 8: 算法 DetermineArbitrage

由于 Bellman-Ford 算法时间复杂度为 $O(m\cdot n)$, 因而算法 DetermineArbitrage 中调用 Bellman-Ford

Algorithm 7 FindEulerCircuit

```
1: procedure FindEulerCircuit(Graph g)                    ▷ 输入图的邻接表
2:     initialize v, res                                  ▷ 选择起点 v
3:     FindEuler(v, res)
4: procedure FindEuler(v, res)                            ▷ 输入结点
5:     for w in g[v] do
6:         e = (v, w)
7:         if !e.visited then
8:             e.visited = 1
9:             FindEuler(w, res)
10:    res.add(v)
```

Algorithm 8 DetermineArbitrage

```
1: procedure DetermineArbitrage(clist[], R[][])
2:     for entry in R do
3:         entry = -In(entry)
4:     res = BellmanFord(R, c[0])
5:     if res == false then
6:         return true
7:     return false
8: procedure BellmanFord(g, u)                            ▷ g 以邻接矩阵形式输入
9:     for k from 0 to n-1 do
10:        d^(k) = []
11:    for v in V, except u do
12:        d^(0)[v] = Infinity
13:    d^(0)[u] = 0
14:    for k from 1 to n do
15:        for w in V do
16:            d^(k)[w] = min{d^(k-1)[w], min_a{d^(k-1)[a] + weight(a,w)}}
17:    if d^(n-1) != d^n then
18:        return false
19:    return d^(n-1)
```

算法部分时间复杂度为 $O(n^2 \cdot n) = O(n^3)$, 为有向边赋值的两层嵌套循环复杂度为 $O(n^2)$, 因而整体时间复杂度为 $O(n^2 \cdot n) = O(n^3)$

6.2 b

根据上问, 此问可转变为输出负权环问题见算法 9: 算法 FindArbitrage

对更改后的 Bellman-Ford' 算法, 其增加了一个循环用于寻找负权环, 循环的数量级取决于负权环的大小, 换言之其时间复杂度最多不超过 $O(|E|)$. 因而此算法时间复杂度与 Bellman-Ford 算法时间复杂度一直, 为 $O(n^3)$

7 Strongly Connected Components Maintaining.

7.1 a

考虑到在添加边的过程中, 可能会出现两种情况:

- 添加的有向边的起点和终点同属于未添加边时的有向图的同一个强连通分量, 那么在添加这条有向边后, 新的有向图的强连通分量并不会发生改变
- 添加的有向边的起点和终点不同属于未添加边时的有向图的同一个强连通分量, 那么则需要对整个图重新执行查找强连通分量的操作, 以确保当新的强连通分量生成时可以被检测到

因而可以维护一个存储现有的强连通分量的存储类型 (例如栈)
见算法 10: 算法 UpdateScc

对算法 FindScc 而言, 其对图中每一个结点与每一条边都进行一次访问, 因而时间复杂度为 $O(|V|+|E|)$
对算法 UpdateScc 而言, 其第一层循环遍历了当前图中的全部强连通分量, 因而最多遍历 $|V|$ 个结点, 时间复杂度为 $O(|V|)$, 因而整个算法的时间复杂度为 $O(|V| + |E|)$

算法 10 是利用 Tarjan 算法来进行检测, 我还构想了另一种算法:
在每一次插入边后, 只需要考虑在进行这次插入后, 这条边所连接的两个点是否构成强连通, 这是因为:

- 倘如添加的有向边的起点和终点同属于未添加边时的有向图的同一个强连通分量, 那么在添加这条有向边后, 新的有向图的强连通分量并不会发生改变
- 倘若添加的有向边的起点和终点不同属于未添加边时的有向图的同一个强连通分量, 不妨假定端点 u 属于强连通分量 S_u, v 属于强连通分量 S_v, 根据定义显然有:
  对任意一点 $u_{w_i} \in S_u$, u_{w_i} 与 u 强连通, 那么假如在插入边后 u 与 v 构成强连通, 那么 u_{w_i} 必然将与 v 构成强连通 (通过新添加的边), 根据对称性可知 v_{w_i} 也将与 u 构成强连通, 换言之即 S_u 与 S_v 构成了一个新的强连通分量
- 接下来证明, 倘若添加的有向边的起点和终点不同属于未添加边时的有向图的同一个强连通分量, 当添加一条边后, 如果该边两端点 u 与 v 仍不构成强连通, 那么图的强连通分量不会发生改变:

Algorithm 9 FindArbitrage

```
1: procedure FindArbitrage(clist[], R[][])
2:     for entry in R do
3:         entry = -In(entry)
4:     res = BellmanFord'(R, c[0])
5:     if res == false then
6:         return false
7:     return res
8: procedure BellmanFord'(g, u)                          ▷ g 以邻接矩阵形式输入
9:     for k from 0 to n-1 do
10:        d^(k) = []
11:    for v in V, except u do
12:        d^(0)[v] = Infinity
13:    d^(0)[u] = 0
14:    for k from 1 to n-1 do
15:        for w in V do
16:            if d^(k-1)[w] > min_a{d^(k-1)[a] + weight(a,w)} then
17:                d^(k)[w] = min_a{d^(k-1)[a] + weight(a,w)}
18:                [parent[v] = u
19:            else
20:                d^(k)[w] = d^(k-1)[w]
21:    v = -1
22:    for w in V do
23:        if d^(k-1)[w] > min_a{d^(k-1)[a] + weight(a,w)} then
24:            v = w
25:            break
26:    if v == -1l then
27:        return false
28:    for i in V do
29:        v = parent[v]
30:    k = v
31:    while 1 do
32:        res.add(k)
33:        if k == v and res.length() > 1 then
34:            break
35:        k = parent[k]
36:    return res
```

Algorithm 10 UpdateScc

```
1: procedure UpdateScc(e = (u,v), curscc)▷ e 为添加的边, 端点 u 指向 v, curscc 为当前的强连通分量
2:     for i from 0 to curscc.length()-1 do
3:         if u is in curscc[i] and v is in curscc[i] then
4:             return curscc
5:     return FindScc(u)
6: procedure FindScc(u)                          ▷ 以 u 为根进行 dfs, 以寻找全部强连通分量
7:     DFN(u) = Low[u] = ++Index
8:     s.push(u)                                                    ▷ s 是一个栈
9:     for e in E where e = (u, v) do
10:        if v is not visited then
11:            FindScc(v)
12:            Low[u] = min{Low[u], Low[v]}
13:        else if v is in s then
14:            Low[u] = min{Low[u], DFN[v]}
15:        if DFN[u] == Low[u] then
16:            while u != v do
17:                v = s.pop()
18:                res[i].add(v)                          ▷ i 初始为 0, res 为一个链表数组
19:                i++
```

利用反证法. 假设添加边后形成新的强连通分量, 那么这条强连通分量必定需要包含新添加的边, 换言之也必须包含 u 与 v, 那么 u 与 v 将构成强连通, 与假设矛盾

据此可以构成新的算法, 见算法 11: 算法 UpdateScc ‘

Algorithm 11 UpdateScc ’

```
1: procedure UpdateScc ‘ (e = (u,v), curscc)▷ e 为添加的边, 端点 u 指向 v, curscc 为当前的强连通分量
2:     su = -2, sv = -2
3:     if JudgeScc(u, v) then
4:         for i from 0 to curscc.length()-1 do
5:             if u in curscc[i] then
6:                 su = i
7:             if v in curscc[i] then
8:                 sv = i
9:             if su >0 and sv > 0 then
10:                break
11:        merge curscc[su] and curscc[sv]
12:    return curscc
13: procedure JudgeScc(u, v)                    ▷ 以 v 为起点, 查找是否存在从 v 到 u 的路径
14:    for w in adj[v] do
15:        if w.visited then
16:            break
17:        w.visited = 1
18:        if w == u then
19:            return true
20:        JudgeScc(u, w)
21:    return false
```

算法 UpdateScc ‘ 在每次添加从 u 到 v 的有向边以后检验是否存在从 v 到 u 的路径, 时间复杂度为 DFS 的时间复杂度, 即 $O(|V| + |E|)$

7.2 b

为了保持原来的强连通分量, 假设要添加从 u 到 v 的有向边, 那么只需检测是否存在从 v 到 u 的路径即可见算法 12: 算法 MaintainScc

算法 MaintainScc 一共进行 m 次循环, 每一次循环内执行一个 DFS 来进行判断, 因而总时间复杂度为 $O(m \cdot (|V| + |E|))$

Algorithm 12 MaintainScc

```
1: procedure MaintainScc(m)
2:     num = 1
3:     while 1 do
4:         input e = (u,v)                          ▷ e 为添加的边, 端点 u 指向 v
5:         if JudgeScc(u, v) then
6:             output "false"                       ▷ 不允许插入
7:         else
8:             insert e
9:             num ++
10:        if num == m then
11:            break
12: procedure JudgeScc(u, v)                        ▷ 以 v 为起点, 查找是否存在从 v 到 u 的路径
13:    for w in adj[v] do
14:        if w.visited then
15:            break
16:        w.visited = 1
17:        if w == u then
18:            return true
19:        JudgeScc(u, w)
20:    return false
```