算法期末复习（一）代码框架/基础证明篇

2022/6/10

- 算法期末复习（一）代码框架/基础证明篇
  - 一.DFS主框架以及DFS-WRAPPER
    - 有向图的算法
      - 1.活动区间
      - *2.白色路径定理的证明:dfs树中祖先子孙的判断
      - 3.拓扑排序TOPO-ORDER
      - 4.关键路径CRITICAL-PATH
      - 5.强连通分支算法SCC
    - 无向图的算法
      - 1.寻找割点 ARTICULATION-POINT-DFS
      - 2.寻找桥BRIDGE-DFS
    - DFS的非递归形式
  - 二.BFS主框架及BFS-WRAPPER
      - 1.判断二分图 BFS-BIRPARTIE
      - 2.寻找k度子图 K-DEGREE-SUBGRAPG(G,k)
  - 三.图贪心算法——经典的MST算法与sssp算法
    - MST
      - 1.PRIM算法
      - *2.PRIM算法的正确性证明
      - 3.Kruskal算法
      - *4.Kruskcal算法正确性的证明
    - SSSP
      - 1.Dijkstra
      - 2.有向图上的sssp

一.DFS主框架以及DFS-WRAPPER

```
DFS(v):
v.color = GRAY;
<Preorder processing of node v>
foreach neighbor w of v do:
    if w.color == WHITE then
        <Exploratory processing of edge vw>;
        DFS(w);
        <Backtrack processing of edge vw>;
    else
        <check edge vw>;
<Postorder processing of node v>
v.color = BLACK;
```

有向图的BE：color == GRAY

无向图的BE：color == GRAY and w!=v.parent;

```
DFS-WRAPPER(G):
foreach node of G do:
    if node.color == WHITE them
        DFS(node);
```

有向图的算法

1.活动区间

```
DFS(v):
v.color = GRAY;
v.discovertime=time;
time+=1;
<Preorder processing of node v>
foreach neighbor w of v do:
    if w.color == WHITE then
        <Exploratory processing of edge vw>;
        DFS(w);
        <Backtrack processing of edge vw>;
    else
        <check edge vw>;
<Postorder processing of node v>
v.color = BLACK;
v.finishtime=time;
time+=1;
```

*2.白色路径定理的证明:dfs树中祖先子孙的判断

> 在深度优先遍历树中，节点v是w的祖先，当且仅当在遍历过程中发现点v的时刻，存在一条从v到w的全部由白色节点组成的路径。

3.拓扑排序TOPO-ORDER

```
DFS(v):
v.color = GRAY;
<Preorder processing of node v>
foreach neighbor w of v do:
    if w.color == WHITE then
        <Exploratory processing of edge vw>;
        DFS(w);
        <Backtrack processing of edge vw>;
    else if w.color==GRAY then
        No TopoOrder
<Postorder processing of node v>
globalnum--;
v.topoNum = globalnum-1;
v.color = BLACK;
```

4.关键路径CRITICAL-PATH

```
//翻转G
//TOPO序最小的几个检查一下出度是否为0，为0的就把v.eft设为v.l.
CRITICAL-PATH(v):
v.color = GRAY;
v.est = -∞;
v.CritDep = -1;
foreach neighbor w of v do:
    if w.color == WHITE then
        <Exploratory processing of edge vw>;
        CRITICAL-PATH(w);
        <Backtrack processing of edge vw>;
        if v.est < w.eft then:
            v.est := w.eft;
            v.CripDep := w;
    else
        <check edge vw>;
        if v.est < w.eft then:
            v.est := w.eft;
            v.CripDep := w;
<Postorder processing of node v>
v.color = BLACK;
v.eft := v.est + v.l;
```

5.强连通分支算法SCC

sink SCC是指G的收缩图里面没有出度的节点。

找到转置G中的finishtime最大的顶点。按finishtime排一个序（用堆实现比较好）
从该顶点开始在G中做一次DFS，找出其连通片上的点，每找到一个就删掉一个节点，然后找到finishtime最大的那一个，继续dfs，直到存finishtime的地方为空。

无向图的算法

1.寻找割点 ARTICULATION-POINT-DFS

割点基于dfs的定义：v不是遍历的根节点，v为割点当且仅当在遍历树中**存在v的一棵子树，没有任何BE指向v的祖先**。

```
ARTICULATION-POINT-DFS(v):
v.color = GRAY;
v.discovertime = time;
time+=1;
v.back := v.discovertime;
<Preorder processing of node v>
foreach neighbor w of v do:
    if w.color == WHITE then
        <Exploratory processing of edge vw>;
        w.parent := v;
        ARTICULATION-POINT-DFS(w);
        if w.back >= v.discovertime then
            output w as articulation point;
        v.back:=min(v.back,w.back);
```

```
    else
        <check edge vw>;
        if w.color==GRAY and w!=v.parent then
            v.back:=min(v.back,w.discovertime);

<Postorder processing of node v>
v.color = BLACK;
```

2.寻找桥BRIDGE-DFS

桥基于dfs的定义：给定遍历树中的TE边的uv(u是父节点)，uv是桥，当且仅当以v为根的所有遍历树子树中没有BE指向v的祖先。

```
BRIDGE-DFS(v):
v.color = GRAY;
v.discovertime := time;
v.back := time;
time+=1;
<Preorder processing of node v>
foreach neighbor w of v do:
    if w.color == WHITE then
        <Exploratory processing of edge vw>;
        DFS(w);
        v.back:=min(v.back,w.discovertime);
        if w.back > v.discovertime then
            output vw as a bridge.
    else
        <check edge vw>;
        if w.color==GRAY and w!=v.parent then
            v.back:=min(v.back,w.discovertime);

<Postorder processing of node v>
v.color = BLACK;
```

DFS的非递归形式

```
DFS(v):
    S.Push(v);
    v.color=GRAY;
    while !S.isEmpty() do
        v := S.getTopElement();
        flag = 1;
        foreach neighbor w of v do :
            if w.color == WHITE then
                S.Push(w);
                w.color=GRAY;
                flag = 0;
                break;
        if flag then
            v:=S.pop();
            v.color=Black;
            <check v>
```

二.BFS主框架及BFS-WRAPPER

```
BFS-WRAPPER(G):
foreach node v in G do :
    v.volor = WHITE;
    v.parent = NULL;
    v.dis = +∞;
foreach node v in G do :
    if v.color == WHITE then
        BFS(v);
return ;
```

```
BFS(v):
Initialize an empty queue Q;
v.color = Gray;
v.dis=0;
Q.Enqueue(v);
while(!Q.isEmpty())
    v = Q.Dequeue();
    foreach neighbor w of v do:
        if w.color ==  WHITE them :
            w.color = GRAY;
            w.parent = v;
            w.dis = v.dis + 1;
            Q.Enqueue(w);
w.color = BLACK;
```

1.判断二分图 BFS-BIRPARTIE

```
BFS-BIRPARTIE(v):
Initialize an empty queue Q;
v.color = Gray;
v.dis=0;
Q.Enqueue(v);
while(!Q.isEmpty())
    v = Q.Dequeue();
    foreach neighbor w of v do:
        if w.color ==  WHITE them :
            w.color = GRAY;
            w.biColor = ~v.biColor
            w.parent = v;
            w.dis = v.dis + 1;
            Q.Enqueue(w);
        else
            if w.biColor == v.biColor then
                return False;
w.color = BLACK;
```

```
return True;
```

2.寻找k度子图 K-DEGREE-SUBGRAPG(G,k)

```
K-DEGREE-SUBGRAPGH(v):
Initialize an empty queue Q;

for node v in G do:
    if v.degree < k then
        Q.Enqueue(v);

while(!Q.isEmpty())
    v = Q.Dequeue();
    foreach neighbor w of v do:
        delete edge wv;
        w.degree--;
        if w.degree < k and w is not in Q then:
            Q.Enqueue(w);
    delete v from G;
```

三.图贪心算法——经典的MST算法与sssp算法

MST

1.PRIM算法

```
PRIM(G):
Initialize a MinHeap;
v.candidateEdge:=NULL;/*标记它是通过哪条边被连入最小生成树的；起点没有candidate*/
s.priority = -∞;
MinHeap.INSERT(s);
while MinHeap != empty do
    v := MinHeap.EXTRACT-MIN();
    MST := MST∪{v.candidateEdge};
    UPDATE-FRINGE(MinHeap,v);
```

```
UPDATE-FRINGE(MinHeap,v):
    foreach neighbor w of v do
        newWeight := vw.weight;
        if w is UNSEEN then
            w.candidateEdge := vw;
            w.priority := newWeight;
            MinHeap.INSERT(w);
        else
            if newWeight < w.priority then
                w.priority := newWeight;
                MinHeap.DECREASE-KEY(w);
```

PRIM算法步骤：

①从源点开始，将其priority置很小的值，初始化一个最小堆

②每次从最小堆取出priority最小的顶点，将该顶点加入MST集合中

③对该顶点的邻居priority做更新。

PRIM算法时间复杂度分析：

$$n * EXTRACT - MIN + n * INSERT + m * DECREASE - KEY = O((n + m)logn)$$

***2.PRIM算法的正确性证明**

最小生成树-间接定义：给定图G的生成树T，定义T是图G的最小生成树，如果它满足"最小生成树"性质：对任意不在T中的边e,T∪{e}含有一个环，并且e是环中最大权值的边（可能不唯一）。

使用数学归纳法证明。

3.Kruskal算法

Kruskal算法步骤：

①把所有边加入一个最小堆中

②每次从最小堆取出边权最小的边，判断两个顶点是否在一个并查集中，不在的话加入一个并查集中，否则不要这条边。

```
KRUSKAL(G):
while MinHeap != empty do
    vw:= MinHeap.EXTRACT-MIN();
    if FIND(v)!=FIND(w) then
        MST :=  MST∪{vW};
        if MST.size == n then
            return ;
        UNION(v,w);
```

KRUSKAL算法时间复杂度分析：

$$m * EXTRACT - MIN + m * INSERT + O(l(m) + n) = O(mlogm + n) = O(mlogm)$$

***4.Kruskcal算法正确性的证明**

SSSP

1.Dijkstra

```
Dijkstra(G):
Initialize a MinHeap;
Initialize D[i]=∞
s.priority = -∞;
D[s]=0;
foreach neighbor w of s do :
    path[w] = s;
    w.priority := sw.weight;
    MinHeap.INSERT(w);

while MinHeap != empty do
    v := MinHeap.EXTRACT-MIN();
    UPDATE-FRINGE(MinHeap,v);
```

```
UPDATE-FRINGE(MinHeap,v):
    foreach neighbor w of v do
        newWeight := vw.weight + v.priority;
        if w is UNSEEN then
            path[w]=v;
            w.priority := newWeight;
            MinHeap.INSERT(w);
        else
            if newWeight < w.priority then
                w.priority := newWeight;
                Path[w]=v;
                MinHeap.DECREASE-KEY(w);
```

DIJKSTRA算法时间复杂度分析:

$$n*EXTRACT-MIN+n*INSERT+m*DECREASE-KEY=O((n+m)logn)$$

2.有向图上的sssp