北京航空航天大学《算法设计与分析》第四次作业

1 对下面的每个描述，请判断其是正确或错误，或无法判断正误。对于你判为错误的描述，请说明它为什么是错的。(每小题 5 分，共 20 分)

1. 任何 NP 完全问题都不存在多项式时间内的解法。
2. P 类问题是 NP 类问题的真子集。
3. 对某问题 $X \in NP$ 而言，若可以证明规约式 $3-SAT \leq_p X$，则 $X \in NPC$。
4. 对于一个 NP 完全问题，其所有种类的输入均需要用指数级的时间求解。

1.1 任何 NP 完全问题都不存在多项式时间内的解法。

无法判断正误

由 NP 完全问题的定义可知，NP 完全问题是 NP 问题的子集，并且如果有一个多项式时间算法可以解决其中一个问题，那么就可以解决所有 NP 问题。也就是说，NP 完全问题是否存在多项式时间内的解法是不确定的，只是现在还没有找到能够解决该类问题的算法，因此无法判断正误。

1.2 P 类问题是 NP 类问题的真子集。

无法判断正误

P 类问题是 NP 类问题的子集，但是目前 $P \neq NP$ 问题依然是一个没有解决的问题，也就无法确定 P 类问题是否是 NP 类问题的真子集。这是因为目前尚无法确定 NP 类问题究竟能不能找到多项式时间内的解法，所以无法得知 $P = NP$ 或 $P \neq NP$，因此无法判断正误。

1.3 对某问题 $X \in NP$ 而言，若可以证明规约式 $3-SAT \leq_p X$，则 $X \in NPC$。

正确

因为 3-SAT 问题是一个已知的 NP 完全问题，如果它可以在多项式时间内规约到问题 X，那么可以说明 X 至少和 NP 中最难的问题一样难，也就是说 $X \in NPC$。

1.4 对于一个 NP 完全问题，其所有种类的输入均需要用指数级的时间求解。

错误

NP 完全问题是指目前无法找到多项式时间内解决所有输入的算法，但是并不意味着所有的输入都需要指数级的时间来求解，有些输入在多项式时间内就能解决。例如有些特殊情况下的旅行商问题就可以在多项式时间内解决。

2 节点寻找问题 (20 分)

现存在一张由 n 个节点，m 条边组成的有向图 G。图 G 中所有顶点按照 1 到 n 进行编号，第 i 条有向边起点为 s_i，终点为 t_i。令函数 $F(i)$ 表示由顶点 i 出发能够到达的最小的顶点编号（每一个顶点都可以到达它自己）。请设计一个算法计算图 G 中每一个顶点 i 对应的 $F(i)$ 值，请描述算法的核心思想，给出算法伪代码并分析其对应的时间复杂度。

2.1 算法核心思想

这个问题可以使用深度优先搜索（DFS）来解决，但是由于题目没有限定给定的图一定是无环图，因此如果直接 DFS 的话会导致环中某些点的 F 值是环的入口点的 F 值，而此时入口点的 F 值还没有计算完成，最终出现错误。因此，这道题我们需要考虑使用增强版的 DFS 来解决。

分析发现，在图中的一个强连通分量里，每个点的 F 值都一定是相同的，所以我们可以使用压点的思想，将一个强连通分量压成一个点，这样一来求 F 就变成了对有向无环图进行 DFS 的简单操作。

具体压点的思路是，首先对原图的转置图 G^T 进行一遍 DFS，得到转置图的逆后序遍历的顶点数组，然后在原图中按照刚才求出的逆后序的顶点顺序的倒序进行 DFS（这样就能保证每次操作的“压点”能够到达的其他“压点”的 F 值已经求出），在搜索过程中标记每个强连通分量能够到达的最小点编号。这样一来，对原图每次搜索的过程相当于对其中一个强连通分量的遍历，其中每个点的 F 值相同，等于该强连通分量中最小点的编号。而整个过程又相当于是对“压点”后的图进行的整体 DFS，后面进行 DFS 的强连通分量能到达的其他强连通分量的最小编号一定已经在该次 DFS 之前计算完成。执行上述步骤直到求出每个点的 F 值，算法结束。

2.2 算法伪代码

Algorithm 1: 节点寻找问题（主函数）

Input: 节点个数 n，边的个数 m，边的起点数组 s，边的终点数组 t

Output: 表示每个顶点能到达的最小顶点编号的数组 F

```
1  function main(n, m, s, t):
2      let graph[n+1][n+1] be a new two-dimension array;
3      let edgeNum[n+1] be a new array;
4      let visited[n+1] be a new array;
5      let F[n+1] be a new array;
6      let back[n+1] be a new array;
7      for i = 0 → m − 1 do
8          graph[s[i]][edgeNum[s[i]] + +] = t[i];
9          //初始化邻接表
10     end
11     //得到转置图;
12     graph^T, edgeNum^T = reverse(graph, edgeNum);
13     for i = 1 → n do
14         if visited[i] == 0 then
15             //对转置图进行 DFS 得到逆序数组;
16             DFS(i, graph^T, edgeNum^T, visited, back);
17         end
18     end
19     //未完接下页;
20 end
```

```
//接上页;
visited.clear();
for i = n → 1 do
    if visited[back[i]] == 0 then
        let temp be a new array;
        //按照逆序数组的倒序对原图进行 DFS 得到 F 数组;
        min = DFSM(back[i], graph, edgeNum, visited, F, temp);
        for v ∈ temp do
            F[v] = min;
        end
    end
end
return F
```

Algorithm 2: 节点寻找问题（其他部分）

```
function DFS(node, graph, edgeNum, visited, back):
    visited[node] = 1;
    for i = 0 → edgeNum[node] − 1 do
        if !visited[graph[node][i]] then
            DFS(graph[node][i], graph, edgeNum, visited, back);
        end
    end
    back.push(node);
end
function DFSM(node, graph, edgeNum, visited, F, temp):
    visited[node] = 1;
    low = node;
    for i = 0 → edgeNum[node] − 1 do
        if !visited[graph[node][i]] then
            m = DFSM(graph[node][i], graph, edgeNum, visited, F, temp);
            low = min(low, m);
        end
        else
            low = min(low, F(graph[node][i]));
        end
    end
    temp.push(node);
    return low
end
```

2.3 算法时间复杂度分析

- 初始化邻接表的时间复杂度为 $O(m)$，转置图（初始化反图邻接表）时间复杂度为 $O(m)$
- DFS 得到逆序数组和 DFSM 更新 F 数组的过程中，每个节点只被处理一遍，并且处理每个节点时要遍历以它为起点的所有边，因此该过程的时间复杂度为 $O(n+m)$
- 总时间复杂度为 $O(n+m)$

3 二分图判定问题 (20 分)

二分图是指一个无向图 $G=(V,E)$，它的顶点集可被分成两个互不相交子集，且同一个子集中任何两顶点间都没有边相连。换言之，G 为二分图，当且仅当存在两个集合 $V1,V2$ 满足 $V1 \cup V2 = V, V1 \cap V2 = \emptyset$，$E$ 中每条边都连接了 $V1$ 中某个点与 $V2$ 中某个点。请设计一个基于广度优先搜索 (BFS) 的算法来判断无向图 G 是否为二分图，请描述算法的核心思想，给出算法伪代码并分析其对应的时间复杂度。

3.1 算法核心思想

按照题目要求，这道题我采取宽度优先搜索（BFS）的方式来进行计算。

具体的做法是采取图着色的思想，首先将一个顶点加入队列进行初始化，然后开始执行 BFS，如果该节点尚未着色，将其标记为 1，然后遍历该节点的子节点。如果子节点没有染色，将其推入队列并标记为和当前节点不同的颜色（1 或 2）；否则，判断子节点颜色与当前节点是否相同，若相同则说明不是二分图，返回 false，算法结束。

重复上述操作直到队列为空。

如果存在节点尚未被染色，重新执行上述步骤，直到所有节点完成着色，说明是二分图，返回 true，算法结束。

3.2 算法伪代码

Algorithm 3: 二分图判定问题

Input: 图 $G(V,E)$

Output: 是否为二分图 (true or false)

```
1  function main(G(V,E)):
2      let color[V.length] be a new array;
3      let queue be a new queue;
4      for v ∈ V do
5          if color[v] == 0 then
6              queue.push(v);
7              color[v] = 1;
8              while !queue.empty() do
9                  node = queue.pop();
10                 for vnext ∈ node.next do
11                     if color[vnext] == 0 then
12                         queue.push(vnext);
13                         //着色为和当前节点不同的颜色;
14                         color[vnext] = 3 − color[node];
15                     end
16                     else if color[vnext] == color[node] then
17                         return false;
18                     end
19                 end
20             end
21         end
22     end
23     return true;
24 end
```

3.3 算法时间复杂度分析

- BFS 的过程中，每个节点只被处理一遍，因此该过程的时间复杂度为 $O(|V|+|E|)$
- 总时间复杂度为 $O(|V|+|E|)$

4 道路改建问题 (20 分)

在某一座城市的郊区存在 n 个村庄，它们通过一个由 m 条双向可通行的公路组成的道路网络彼此互通，第 i 条公路连接村庄 s_i 和 t_i，长度为 w_i 公里。为了支持各个村庄的经济发展，当地政府计划将其中的一些公路修建为高速公路。但是受限于地理因素，政府只会选择 $n-1$ 条公路进行改建，使得完工后的高速公路网络满足：

1. 这 $n-1$ 条高速公路恰好可以使得这些村庄彼此互通。
2. 最长的若干条（可以是一条或者多条）高速公路长度恰好为 K。

但是，修建高速公路的过程不能改变原公路的长度，因此，在现有的道路网络上可能无法找到合适的 $n-1$ 条待改建公路，所以当地政府计划先改变一些现有公路的长度。将现有的任意一条公路扩充 1 公里或者缩短 1 公里都需要 1 单位成本，请设计一个算法帮忙计算最少需要多少单位成本改建现有的道路网络才能使得这个高速公路修建工程顺利开工。请描述算法的核心思想，给出算法伪代码并分析其对应的时间复杂度。

4.1 算法核心思想

这道题可以采用 Kruskal 算法的修改版本来解决。具体的思路是首先对所有的边按照长度 w_i 进行排序，然后执行 Kruskal 算法生成最小生成树。如果生成过程中**遇到长度大于 K 的边，就将其长度缩减到 K，并且在总成本中加上对应的缩减成本**。

如果最终的最小生成树中所有的边长度都小于 K，那么我们可以用所有边中长度最接近 K 的一条边（|e-K| 最小）来替换最小生成树中的一条边，得到一棵目标生成树，而**此时的总成本就是将这条边缩短或延长到 K 花费的成本，即 |e-K|**。具体的替换规则如下：

1. 如果在最小生成树中，边 e 的两个顶点之间有边，那么将其替换为 e；
2. 否则，将 e 加入最小生成树中，然后删掉新形成的环中除了边 e 以外任意一条边。

下面证明 e 一定会成为新的生成树中最长的边之一。如果 $e > K$，因为原来最小生成树中的边都小于 K，所以该结论是显然的。如果 $e \le K$，因为对于原最小生成树中的任何一条边 e'，都有 $e' < K$ 和 $|e'-K| \ge |e-K|$，所以 $e' \le e$，此时 e 依然是新生成树中最长的边。因此上述结论成立。

4.2 算法伪代码

Algorithm 4: 道路改建问题（主函数）

Input: 村庄个数 n，道路条数 m，道路起点数组 s，道路终点数组 t，道路长度数组 w，目标最长长度 K

Output: 最小改建成本 $cost$

```
function main(n, m, s, t, w, K):
    let edges[m] be a new array;
    let cost = 0;
    for i = 0 → m − 1 do
        edges[i] = (s[i], t[i], w[i]);
    end
    sort edges by w in ascending order;
    tree, cost = Kruskal(edges);
    if cost == 0&&tree[tree.size − 1] < K then
        //第二种情况;
        let min = 0;
        for e ∈ edges do
            if abs(e.w − K) < abs(min − K) then
                min = e.w;
            end
        end
        cost = min;
    end
    return cost;
end
```

Algorithm 5: 道路改建问题（Kruskal 算法）

```
function Kruskal(edges):
    let tree be a new array;
    let cost = 0;
    let node[n + 1] be a new array;
    for i = 1 → n do
        node[i].parent = node[i];
        node[i].height = 0;
    end
    for e ∈ edges do
        if find(node, e.s) ≠ find(node, e.t) then
            tree.push(e);
            union(node, e.s, e.t);
            if e.w > K then
                cost+ = e.w − K;
            end
        end
    end
    return tree, cost;
end
function find(node, x):
    while node[x].parent ≠ x do
        x = node[x].parent;
    end
    return x;
end
function union(node, x, y):
    xRoot = find(node, x);
    yRoot = find(node, y);
    if xRoot == yRoot then
        return;
    end
    if node[xRoot].height < node[yRoot].height then
        node[xRoot].parent = yRoot;
    end
    else if node[xRoot].height ≥ node[yRoot].height then
        node[yRoot].parent = xRoot;
        if node[xRoot].height == node[yRoot].height then
            node[xRoot].height + +;
        end
    end
end
```

4.3 算法时间复杂度分析

- 初始化边集的时间复杂度为 $O(m)$

- 对所有边按照长度进行排序的时间复杂度为 $O(mlogm)$
- Kruskal 算法的时间复杂度为 $O(mlogn)$
- 寻找距离 K 最近长度的边的时间复杂度为 $O(m)$
- 总时间复杂度为 $O(mlogm)$

5 新最短路径问题 (20 分)

给定一张由 n 个点，m 条有向边组成的有向图 G，其中第 i 条边起点为 s_i ，终点为 t_i，边权为 $w(s_i, t_i)$，保证 $w(s_i, t_i) > 0$，假设 G 中存在一个由 t 个顶点组成的路径 $P = [v_1, v_2, ..., v_t]$。我们在课上学到的有关路径长度的定义为 $dis(P) = \sum_{i=1}^{t-1} w(v_i, v_{i+1})$，即路径上每一条边的边权和。现给出一个新的路径长度定义，如下公式所示：

$$dis(P) = \sum_{i=1}^{t-1} w(v_i, v_{i+1}) - \max_{i=1}^{t-1} w(v_i, v_{i+1}) + \min_{i=1}^{t-1} w(v_i, v_{i+1})$$

特别的，如果 $t = 1$，那么 $dis(P) = 0$。在这个新路径长度定义下，请设计一个算法计算顶点 1 到图中所有顶点，即点 $1, 2, ..., n$ 的最短路径长度。请描述算法的核心思想，给出算法伪代码并分析其对应的时间复杂度。

5.1 算法核心思想

分析新最短路径的定义可以发现，新最短路径实际上就是将原路径中最长边的长度替换成了最短边的长度，这就导致可能出现路径长度随着路径长度不再单调不减的情况，因此原始的 Dijkstra 算法不再奏效。但是我们可以采用改进版的 Dijkstra 算法来解决这一问题。

具体的思路是，我们需要对原来 dist 的状态进行扩展，如果直接将每个点的 dist 和最长边、最小边挂钩的话，状态的数量会非常的庞大，理论上可行但实际上不可行。因此继续考虑其它的扩展方式。前面已经分析过，新最短路径的定义是将路径中最长边删去，最短边翻倍得到的，设这个路径为 L。那么我们在这个路径中删去除了最长边的任意一条边，或是翻倍除了最短边的任意一条边，得到的路径 L' 都显然大于 L。也就是说，如果我们把 dist 和该 dist 对应的路径上是否已经删去一条边或者翻倍一条边挂钩，那么最后寻找到的最短 dist 依然是满足条件的 dist，同时这样可以把状态极大程度上地压缩。

具体做法就是将 $dist$ 扩展为 $dist[n][2][2]$ 这个三维的数组，对这个数组应用 Dijkstra 算法。其中 $dist[i][j][k]$ 表示从顶点 1 到顶点 i 的最短路径，其中路径上是否已经删去一条边或者翻倍一条边分别用 j 和 k 表示，j 和 k 的取值为 0 或 1，0 表示没有删去或翻倍，1 表示已经删去或翻倍。这样一来，我们就可以在 Dijkstra 算法的过程中对每个节点的 dist 进行更新，同时更新对应的 j 和 k 以及对应的三维 $vivited$ 数组，最后得到的 dist 就是最终的结果。

5.2 算法伪代码

Algorithm 6: 新最短路径问题（主函数）

Input: 顶点个数 n，边的条数 m，边的起点数组 s，边的终点数组 t，边权数组 w

Output: 顶点 1 到各个顶点的最短路径数组 $dist$

```
1  function main(n, m, s, t, w):
2      let finalDist[n] be a new dimension array;
3      let visited[n][2][2] be a new 3-dimension array;
4      let queue be a new priority queue;
5      let G be a new graph;
6      for i = 0 → m − 1 do
7          G.V[s[i]].next.push(t[i]);
8          G.V[s[i]].w[t[i]] = w[i];
9      end
10     for u ∈ G.V do
11         u.dist[0][0] = u.dist[0][1] = u.dist[1][0] = u.dist[1][1] = ∞;
12     end
13     G.V[0].dist[0][0] = G.V[0].dist[0][1] = G.V[0].[1][0] = G.V[0].dist[1][1] = 0;
14     visited[0][0][0] = visited[0][0][1] = visited[0][1][0] = visited[0][1][1] = 1;
15     queue.push(G.V[0].dist[0][0], 0, 0, 0);
16     Dijkstra(G, queue);
17     for i = 0 → n − 1 do
18         finaiDist[i] = G.V[i].dist[1][1];
19     end
20     return finalDist;
21 end
```

Algorithm 7: 新最短路径问题（Dijkstra）

```
Input: 图 G(V, E)，优先队列 queue
function Dijkstra(G, queue):
    while !queue.isEmpty() do
        u = queue.pop();
        if visited[u[1]][u[2]][u[3]] then
            continue;
        end
        visited[u[1]][u[2]][u[3]] = 1;
        for v ∈ G.V[u[1]].next do
            w = G.V[u[1]].w[v];
            if u[0] + w < v.dist[u[2]][u[3]] then
                v.dist[u[2]][u[3]] = u[0] + w;
                queue.push(v.dist[u[2]][u[3]], v, u[2], u[3]);
            end
            if u[2] == 0&&u[0] < v.dist[1][u[3]] then
                v.dist[1][u[3]] = u[0];
                queue.push(v.dist[1][u[3]], v, 1, u[3]);
            end
            if u[3] == 0&&u[0] + 2 * w < v.dist[u[2]][1] then
                v.dist[u[2]][1] = u[0] + 2 * w;
                queue.push(v.dist[u[2]][1], v, u[2], 1);
            end
            if u[2] == 0&&u[3] == 0&&u[0] + w < v.dist[1][1] then
                v.dist[1][1] = u[0] + w;
                queue.push(v.dist[1][1], v, 1, 1);
            end
        end
    end
end
```

5.3 算法时间复杂度分析

- 初始化图 G 的时间复杂度为 $O(m)$
- 初始化各个节点的时间复杂度为 $O(n)$
- 在执行 Dijkstra 算法的过程中，每个节点最多被访问 4 次，因此该过程总的时间复杂度为 $O((n+m)logn)$
- 最后将结果存入数组的时间复杂度为 $O(n)$
- 因此总时间复杂度为 $O((n+m)logn)$