1 基本算法实现

1.1 数据结构设计---邻接链表法

基本思想：对拓扑图中的每个顶点建立一个单链表，存储该顶点所有邻接顶点及其相关信息，每一个单链表设一个表头结点。本算法会把拓扑图中的无向图拆成两条有向边，它将每个顶点的所有相邻顶点都保存在该顶点对应的元素所指向的一张链表中。我们使用这个邻接表数组就是为了快速访问给定顶点的邻接顶点列表，如下所示(图 1 是拓扑图，图 2 是存储的形式)。

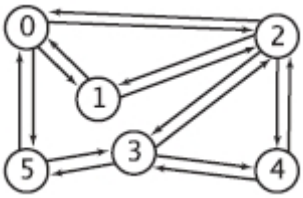

图 1

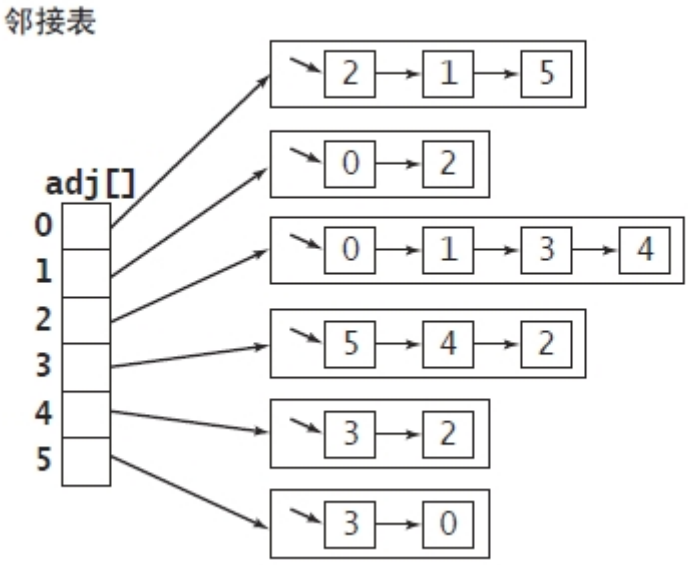

图 2

由图 2 可知，顶点 0 的 adj[0]链表中，访问的顺序是 2,1,5。顶点 2 1 5 在 adj[0]中可能有多种排序，这些排序会有不同的深度优先搜索结果。

本文图的数据结构为如下所示：

```
public class NetworkGraph{
    public int V;                                   //the number of network nodes
    public int E;                                   //the number of network links
    private LinkedList<NetworkLink>[] adj;

    public NetworkLink notRequiredLink;
    public ArrayList<Integer> requiredVertex;
    public ArrayList<NetworkLink> requiredLinks;
    public LinkedHashMap<String,Integer> requiredWeight;

    public int source;
    public int terminal;
```

图 3

其中 notRequiredLink 存储禁止通过的线段；requiredVertex 存储必经节点；requiredLinks 存储必经线段；requiredWeight 的 key 为约束(必经点或者必经线段)，value 为权重值。

这种 graph 的实现有如下特点，使用的空间和 V+E 成正比，添加一条边所需的时间为常数，遍历顶点 v 的所有相邻顶点所需的时间为常数。

1.2.深度优先搜索

搜索连通图的经典递归算法（遍历所有的顶点和边）和 Tremaux 搜索类似，但描述起来更简单。要搜索一幅图，只需用一个递归方法来遍历所有顶点。在访问拓扑图一个顶点时：首先将它标记为已访问，然后递归地访问它的所有没有被标记过的邻居顶点。

为了更好地与迷宫的 Tremaux 搜索对应起来，我们可以把图 1 想象一座完全由单向通道构造的迷宫（每个方向都有一个通道）。和在迷宫中会经过一条通道两次（方向不同）一样，在图中我们也会路过每条边两次（在它的两个端点各一次）。在 Tremaux 搜索中，要么是第一次访问一条边，要么是沿着它从一个被标记过的顶点退回。在无向图的深度优先搜索中，在碰到边 v-w 时，要么进行递归调用（w 没有被标记过），要么跳过这条边（w 已经被标记过）。第二次从另一个方向 w-v 遇到这条边时，总是会忽略它，因为它的另一端 v 肯定已经被访问过了（在第一次遇到这条边的时候）。

深度优先搜索标记与起点连通的所有顶点所需的时间和顶点的度数之和成正比。

1.3.最短路径----SPFA 算法

本文采用 SPFA 算法来计算最短路径，SPFA 算法的精妙之处在于不是盲目的做松弛操作，而是用一个队列保存当前做了松弛操作的结点。只要队列不空，就可以继续从队列里面取点，做松弛操作，举例如下。

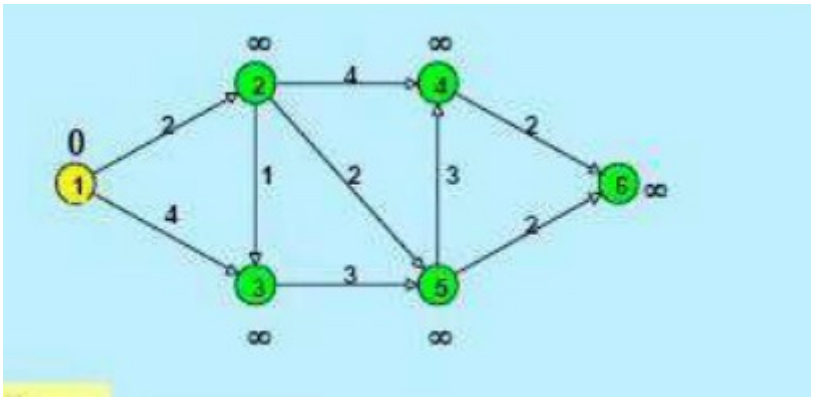

图 4

当前源点 1 在队列里面，于是我们取了 1 结点来进行松弛操作，显然这个时候 2,3 结点的距离更新了，入了队列，我们假设他们没入队列,即现在队列已经空了，那么还有没有必要继续做松弛操作呢？显然没必要了啊，因为源点 1 要到其他结点必须经过 2 或 3 结点啊。

算法大致流程是用一个队列来进行维护。初始时将源点加入队列，每次从队列中取出一个元素，并对所有与他相邻的点进行松弛，若某个相邻的点松弛成功，如果该点没有在队列中，则将其入队，直到队列为空时算法结束。

本文会实现 SPFA 算法的两个优化即 SLF 和 LLL。SLF：Small Label First 策略，设要加入的节点是 j，队首元素为 i，若 dist(j)<dist(i)，则将 j 插入队首，否则插入队尾。 LLL：Large Label Last 策略，设队首元素为 i，队列中所有 dist 值的平均值为 x，若 dist(i)>x 则将 i 插入到队尾，查找下一元素，直到找到某一 i 使得 dist(i)<=x，则将 i 出对进行松弛操作。 SLF 可使速度提高 15 -20%；SLF + LLL 可提高约 50%。

SPFA 算法在计算最短路径时，在最坏情况下其复杂度是 O(VE)。

2.算法设计思路

2.1 产生初始解空间

由于要尽量满足 4 个约束，两个必经节点+两条必经线段，所以采用深度优先搜索的方式产生初始解。程序开始时，把输入文件的数据读入 initGraph(本算法使用的数据结构，如图 3 所示)，用以下步骤产生一个初始解：

1. 构造 initGraph 的副本(很重要)，遍历所有顶点的邻接链表，对于顶点 v 的邻接链表 adj[v]，有两种情况：①如果 v 不是必经线段(2-4,13-14)中的其中一个端点(即 v 不在顶点集合 2,4,13,14 中)，直接打乱 adj[v]中的顶点排列顺序，举例若 v=0，adj[0]中顶点排序为 1 2 3，那么打乱后 adj[0]中的顶点排序不为 1 2 3 当然有多种可能，1 3 2 就是其中之一。②如果 v 是必经线段(2-4,13-14)中的其中一个端点(即 v 在顶点集合 2,4,13,14 中)，那么首先打乱 adj[v]中的顶点排列顺序，如①的操作。然后做以下的操作，假设 v=2，打乱其邻接链表 adj[2]中元素原有排序，得到新排序为 0 1 4 3 5，那么就把顶点 4 移到 adj[2]的首部，也就是让顶点 4 变为 adj[2]的第一个元素。

这样做的好处就是从起点(0)开始进行深度优先搜索，如果搜索路径经过了必经线段中的其中一个端点，那么搜索路径必然会经过必经线段中的另一个端点，这样的搜索路径就可以满足必经线段约束了。

2. 从起点 0 开始，进行深度优先搜索，目的为终点(17)，会得到一条从起点到终点的搜索路径。

3. 检查得到的搜索路径，若路径经过的节点数不超过 9,而且满足至少一个约束，那么成功产生了初始解，结束。否则跳转 1。

为了得出最好的参考解，本算法初始解空间大小为 40，即产生 40 个初始解。

2.2 优化初始解空间

由 2.1 产生的初始解空间中每一个初始解至少满足一个约束(可能是必经顶点约束，或者必经线段)，对解空间中的每个初始解做以下初步优化，如下。

1. 从初始解路径的起点(0)开始正序遍历解路径中的每一个点，找到路径中第一个顶点（该顶点是约束集合 2,4,13,14,7,12 中的其中一个顶点,离起点最近，在这里把该顶点命名为 leftNode)，记下其索引 index1。

2. 从初始解路径的终点(17)开始倒序遍历解路径中的每一个点，找到第一个顶点(该顶点是约束集合 2,4,13,14,7,12 中的其中一个顶点,离终点最近，在这里把该顶点命名为 rightNode),记下其索引 index2。

3. 如果 index1!=index2，那么使用 SPFA 算法计算从起点到 leftNode 的最短路径 path1，使用 SPFA 算法计算 rightNode 的最短路径 path2，和初始解路径进行拼接，得到新的初始解路径，若新的初始解路径经过的节点数大于 9，那么初始解路径不更新，否则更新；如果 index1==index2，那么直接结束，不对初始解路径做任何处理。

初步优化完初始解之后，然后对解空间的每一个初始解做进一步优化，如下。

1.遍历初始解路径中每一个顶点，计算解路径尚未满足的那些约束(顶点约束，或者线段约束，可能未满足其中一个，或者多个)。

2.如果尚未满足的约束个数大于 0，而且解路径经过的节点数不超过 9，那么执行如下操作，否则退出。①查找解路径中已经满足的约束，并返回一个索引列表(列表中每个索引所对应的顶点都是约束集合(2,4,13,14，7,12)中的顶点)，举例有一个解路径 0 3 7 8 14 13 17。那么返回的索引列表应为 2 4 5。②遍历索引列表，把相邻的索引差值最大的两个索引 index1，index2 取出来，如果 index1 和 index2 差值>1，执行③；否则直接退出 2(不允许拆分初始解路径中的必经线段)。比如 2 4 两个索引差值为 2，而 4 5 两个索引差值为 1，那么取出的索引为 index1=2 对应的顶点为 leftNode，index2=4 对应顶点为 rightNode。③从解路径尚未满足的约束中随机选择一个顶点 X,，利用 SPFA 计算从 leftNode 到顶点 X 的最短路径 path1，并计算从顶点 X 到 rightNode 的最短路径 path2，利用 path1，path2，拆分重组初始解路径，得到新的初始解路径。④如果新的初始解路径经过的节点数不超过 9，那么更新初始解路径，否则不更新。

3.重新计算解路径尚未满足的那些约束(顶点约束，或者线段约束，可能未满足其中一个，或者多个)。执行步骤 2。

2.3 排序解空间中的所有初始解

本算法一开始通过输入文件可以给不同的约束分配不同的权重，而且解空间中的初始解都满足经过的节点数不超过 9 的要求，初始解经过两轮优化的目的是尽量让其满足更多的约束，所以从解空间中选择最优的解应考虑两个条件，一是解路径所满足的约束权重和大小，二是解路径的最终长度。

1.遍历解空间中的所有解，计算它的适应度。每个解的适应度通过两个值来体现，totalWeight 是解路径所满足约束的权重和，totalCost 解路径的总长度。

2.实现比较器，用于排序所有的初始解，如下所示。

```
Comparator<OurIndividual> SortList = new Comparator<OurIndividual>() {
    @Override
    public int compare(OurIndividual one, OurIndividual two) {
        if(one.theFitness.weight!=two.theFitness.weight)
            return -1*(one.theFitness.weight-two.theFitness.weight);
        else
            return one.theFitness.totalCost-two.theFitness.totalCost;
    }
};
```

这里的比较器含义是：如果两个解路径所满足约束的权重和不相等，那么就优先比较解路径的权重和；如果两个解路径所满足约束权重和相等，那么就比较解路径的总长度即可。

比较器的实现方式有多种，本人这里的实现优先考虑权重和，再考虑总长度。也可以优先考虑总长度，再考虑权重和。也可以把解路径所满足的权重、总长度以某种代数关系综合成一种因素 Y，这样排序只需考虑 Y 即可，总之排序方式比较自由。

3.根据比较器，排序解空间中所有的解，得到最优参考解。

2.4 有关该算法的思考

要想得到最优的参考解，也就是说需要初始解空间大小 K 是一个值得深究的问题，这个值不能太小，也不能太大。K 太小的话，通过深度优先搜索对拓扑图进行搜索得到初始解时，对图的搜索不够完全，得到的搜索路径不够，导致最终找不到最优的参考解。K 太大的话，可能会搜索到很多重复的初始解，导致了一些冗余路径的优化。本文取的值是 K 值是 40，基本上可以很快地找到最优的参考解。

3. 算法验证

1. 首先测试约束权重不同时，得到的参考解可能不同，如图 5 的上下两个图所示。

```
必经点约束: 7 权重: 1
必经点约束: 12 权重: 2
必经线段约束: 2 4 权重: 3
必经线段约束: 13 14 权重: 2
禁止通过线段: 11-12 1

参考path: S->N2->N4->N5->N12->N13->N14->N13->E
参考path满足 3 个约束，获得的total weight为: 7
参考path所需的total cost: 12
```

```
必经点约束: 7 权重: 1
必经点约束: 12 权重: 2
必经线段约束: 2 4 权重: 1
必经线段约束: 13 14 权重: 2
禁止通过线段: 11-12 1

参考path: S->N3->N7->N8->N14->N13->N12->N16->E
参考path满足 3 个约束，获得的total weight为: 5
参考path所需的total cost: 9
```

图 5

2. 然后修改其中的一个必经节点约束，如图 6 所示参考解(与图 5 下参考解不同)。

```
必经点约束: 6 权重: 1
必经点约束: 12 权重: 2
必经线段约束: 2 4 权重: 1
必经线段约束: 13 14 权重: 2
禁止通过线段: 11-12 1

参考path: S->N3->N6->N12->N13->N14->N13->E
参考path满足 3 个约束，获得的total weight为: 5
参考path所需的total cost: 10
```

图 6

3. 最后修改禁止通过线段为 3-6，得到的参考解(已满足所有约束)如下图所示。

```
必经点约束: 6 权重: 1
必经点约束: 12 权重: 2
必经线段约束: 2 4 权重: 1
必经线段约束: 13 14 权重: 2
禁止通过线段: 3-6 2

参考path: S->N2->N4->N5->N12->N6->N14->N13->E
参考path满足 4 个约束，获得的total weight为: 6
参考path所需的total cost: 14
```

由以上验证可知：该算法具有通用性，可以适应不同的约束修改，以及权重分配。