



北京大学
PEKING UNIVERSITY

JAVA 课程大作业

学院 软件与微电子学院

组员 潘江 2301210367

组员 于畅 2301210509

组员 常逸歌 2301210174

2024 年 1 月 4 日

目录

1	作业地址	3
2	任务一：基本算法实现	3
2.1	问题描述	3
2.2	算法设计思路	3
2.3	测试方案	9
2.4	部分结果如下所示	9
3	任务二：基于多线程的预测	10
3.1	改进内容	10
3.2	测试方案	12
3.3	部分结果如下所示	12

1 作业地址

- Gitee: https://gitee.com/jiang000/java_homework
- Overleaf: <https://www.overleaf.com/read/qzpzjpnpytg#316605>
- 结果: [output.txt](#)

2 任务一：基本算法实现

2.1 问题描述

对于给定一个数据图和一个查询需求（多个关键词），通过图搜索算法在给定数据图中给出一组合适的 Web APIs，这组 APIs 覆盖所有的关键词，而且能形成连通路径（API 具有兼容性）。

2.2 算法设计思路

- **数据预处理**：根据题目要求，该任务只需要 *api.csv* 和 *mashup.csv* 中的部分数据。
 - *api.csv* 文件中仅保留 *Name*、*PrimaryCategory*、*SecondaryCategories* 三列，得到新的 *api.csv* 文件。
 - *mashup.csv* 文件中仅保留的 *Name*、*RelatedAPIs* 和 *Categories* 三列，得到新的 *mashup.csv* 文件。
- **构建图**：读取 *api.csv* 和 *mashup.csv* 中的数据，从而构建出图。使用了 *OpenCsv* 和 *JgraphT* 工具包来完成 *csv* 文件的读取，以及整个图的构建。其中 *OpenCsv* 对于读写 *csv* 文件提供了便利的操作，*JGraphT*（Java 图工具包）是一个用于处理图和图算法的 Java 库。
 - **读入图的顶点和顶点的类别属性**，该部分核心代码如下：


```

1 //定义一个无向图，读取 api.csv 文件，构建图数据结构
2 Graph<String, DefaultWeightedEdge> undirectedGraph = new
   ↳ SimpleWeightedGraph<>(DefaultWeightedEdge.class);
3 String apiCsvFile = "api.csv";
4 try {
5     CSVParser csvParser = new CSVParser(new
   ↳ FileReader(apiCsvFile), CSVFormat.DEFAULT.withHeader());
6     for (CSVRecord record : csvParser) {
7         String name = record.get("Name");
```

```

8         String category = record.get("Primary Category");
9         undirectedGraph.addVertex(name);
10        PrimaryCategory.put(name,category);
11    }
12 } catch (IOException e) {
13     e.printStackTrace();
14 }

```

- **生成边的权值。**读入 mashup.csv 文件，读入其中的 Related APIs 列，每一个单元里面出现的 api，说明他们曾经被一起调用过，则两两之间存在一条边。边的权值对应着一起出现的次数。该部分核心代码如下：

```

1 // 读取 mashup.csv 文件，计算边的权值
2 String mashupCsvFile = "mashup.csv";
3 try {
4     CSVParser csvParser = new CSVParser(new
5         ↪ FileReader(mashupCsvFile),
6         ↪ CSVFormat.DEFAULT.withHeader());
7     for (CSVRecord record : csvParser) {
8         String[] relatedApis = record.get("Related
9             ↪ APIs").split(",");
10        for(String a :relatedApis){
11            for(String b :relatedApis){
12                if(undirectedGraph.containsVertex(a)&&
13                    ↪ undirectedGraph.containsVertex(b)){
14                    if(a != b){
15                        if(!undirectedGraph.containsEdge(a,b)){
16
17                            ↪ addWeightedEdge(undirectedGraph,a,b,1);
18                        }
19                        else{
20                            DefaultWeightedEdge targetEdge =
21                                ↪ findEdge(undirectedGraph,a,b);
22
23                            ↪ setEdgeWeight(undirectedGraph,targetEdge,
24                                ↪ undirectedGraph.getEdgeWeight(targetEdge)
25                                ↪ + 1);
26                        }
27                    }
28                }
29            }
30        }
31    }
32 }

```



```

5 private Set<String> KeywordsOfTree = new HashSet<>(); //存储该树结点
   ↪ 包含的类别
6 private double totleweight = 0; //存储该树的权值

```

- 两个队列和对队列的操作

```

1 Vector<MidTree> Q = new Vector<>();
2 Vector<MidTree> RQ = new Vector<>();
3 public static void enqueue(Vector<MidTree> WQ, MidTree t){
4     if(WQ.contains(t)) return;
5     WQ.add(t);
6 }
7 public static MidTree dequeue(Vector<MidTree> WQ){
8     if(WQ.size() == 0){
9         return null;
10    }
11    MidTree now = WQ.get(0);
12    WQ.remove(0);
13    return now;
14 }
15 public void updatequeue(Vector<MidTree> WQ){
16    sortVector(WQ,
   ↪ Comparator.comparingDouble(MidTree::getTotleweight));
17 }

```

- Tree Merging 算法：树的合并

```

1 public static Vector<MidTree> mergeTreesWithSameRoot(Vector<MidTree>
   ↪ Q) {
2     Vector<MidTree> mergedTrees = new Vector<>();
3     HashSet<Integer> processedIndices = new HashSet<>();
4     for (int i = 0; i < Q.size(); i++) {
5         if (processedIndices.contains(i)) continue;
6         MidTree tree1 = Q.get(i);
7         for (int j = i + 1; j < Q.size(); j++) {
8             if (processedIndices.contains(j)) continue;
9             MidTree tree2 = Q.get(j);
10            if (tree1.getRoot().equals(tree2.getRoot())) {
11                // 合并 tree1 和 tree2

```

```

12         MidTree mergedTree = mergeTwoTrees(tree1, tree2);
13         mergedTrees.add(mergedTree);
14         processedIndices.add(i);
15         processedIndices.add(j);
16         break;
17     }
18 }
19 if (!processedIndices.contains(i)) {
20     mergedTrees.add(tree1);
21 }
22 }
23 return mergedTrees;
24 }
25
26 private static MidTree mergeTwoTrees(MidTree tree1, MidTree tree2) {
27     // 创建一个新的 MidTree 实例来合并 tree1 和 tree2
28     MidTree mergedTree = new MidTree(tree1.getGraph());
29     mergedTree.setRoot(tree1.getRoot());
30     // 合并顶点集
31     for (String vertex : tree1.getVertexSetofMidTree()) {
32         mergedTree.addVertexToMidTree(vertex);
33     }
34     for (String vertex : tree2.getVertexSetofMidTree()) {
35         mergedTree.addVertexToMidTree(vertex);
36     }
37     // 合并边集
38     for (DefaultWeightedEdge edge : tree1.getEdgeSetofMidTree()) {
39         mergedTree.addEdgeToMidTree(edge);
40     }
41     for (DefaultWeightedEdge edge : tree2.getEdgeSetofMidTree()) {
42         mergedTree.addEdgeToMidTree(edge);
43     }
44     // 合并关键字集
45     for (String keyword : tree1.getKeywordsofTree()) {
46         mergedTree.setKeywordsofTree(keyword);
47     }
48     for (String keyword : tree2.getKeywordsofTree()) {

```

```

49         mergedTree.setKeywordsofTree(keyword);
50     }
51     return mergedTree;
52 }

```

- Tree growth 算法：树的生长

```

1  //tree growth 算法
2  Set<String> N = new HashSet<>(); //找到所有邻居
3  N.addAll(Graphs.neighborListOf(graph, midTree.getRoot()));
4  Set<String> keyofmidtrees = midTree.getKeywordsofTree();
5  for(String n : N){ //遍历邻接点
6      DefaultWeightedEdge e = graph.getEdge(n, midTree.getRoot()); //对
        ↪ 应边
7      if(e == null) continue;
8      String NprimaryCategory = PrimaryCategory.get(n);
9      Set<MidTree> Willremove = new HashSet<>();
10     Iterator<MidTree> iterator = Q.iterator();
11     Vector<MidTree> increaseQ = new Vector<>();
12     while(iterator.hasNext()){
13         MidTree q = iterator.next();
14         if(q != midTree){
15             if(q.getRoot().equals(n)){
16                 //判断两者包含的 key 是否完全一致
17                 keyofmidtrees.add(NprimaryCategory);
18                 if(q.coverKeywords(keyofmidtrees)){
19                     if(q.getTotleweight() >
                        ↪ midTree.getTotleweight() + graph.getEdgeWeight(e)){
20                         iterator.remove();
21                         midTree.setKeywordsofTree(NprimaryCategory);
22                         midTree.setRoot(n);
23                         midTree.addEdgeToMidTree(e);
24                         midTree.addVertexToMidTree(n);
25                         increaseQ.add(midTree);
26                         continue;
27                     }
28                 }
29             }

```



```

30         midTree.setKeywordsofTree(NprimaryCategory);
31         midTree.setRoot(n);
32         midTree.addEdgeToMidTree(e);
33         midTree.addVertexToMidTree(n);
34         increaseQ.add(midTree);
35     }
36 }
37 }
38 for(MidTree newq : increaseQ){
39     Q.add(newq);
40 }
41 increaseQ.clear();
42 }

```

2.3 测试方案

- 读取 mashup.csv 中的 relatedApis 作为验证结果，然后将查询这些 api 所属的种类作为查询的关键词，使用这些关键词调用算法，将 relatedApis 与算法返回的结点做比较，一致则说明命中。
- 一共命中了 1434 个任务，命中率为 22.17573221757
- 若返回的 api 数少于所给的 api 数也算命中，则命中率为 80.04029133736246
- 每个任务的平均耗时为 1157.6011157601115760(ms)
- 总用时 2 小时 11 分钟

2.4 部分结果如下所示

- 第 329 个任务返回 api 共耗时 4296 毫秒
- 第 329 个任务返回的群 Steiner Tree 的权重和为: 0.5
- 第 329 个任务返回的群 Steiner Tree 的顶点个数为:2
- 第 329 个任务未命中
- 第 367 个任务返回 api 共耗时 1459 毫秒
- 第 367 个任务返回的群 Steiner Tree 的权重和为: 0.5
- 第 367 个任务返回的群 Steiner Tree 的顶点个数为:2

- 第 367 个任务未命中
- 第 392 个任务返回 api 共耗时 26 毫秒
- 第 392 个任务返回的群 Steiner Tree 的权重和为: 0.0
- 第 392 个任务返回的群 Steiner Tree 的顶点个数为:1
- 第 392 个任务命中这是第 88 个命中
- 第 394 个任务返回 api 共耗时 26 毫秒
- 第 394 个任务返回的群 Steiner Tree 的权重和为: 0.0
- 第 394 个任务返回的群 Steiner Tree 的顶点个数为:1
- 第 394 个任务命中这是第 90 个命中

3 任务二：基于多线程的预测

3.1 改进内容

使用 `ExecutorService` 创建多线程算法，通过 `ExecutorService` 能够更方便地管理线程，提高程序的并发性能，并最终减少了算法的运行时间，加快了算法的收敛。核心代码如下：

```

1 Set<CSVRecord> csvSet = new HashSet<>(csvParser.getRecords());
2 ExecutorService executor =
    ↳ Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
3 System.out.println(" 一共" + csvSet.size()+" 任务，任务开始");
4 for (CSVRecord record : csvSet) {
5     executor.submit(() -> {
6         int now = totle.incrementAndGet();
7         System.out.println(" 一共" + csvSet.size()+" 任务，第"+now+"
            ↳ 个任务开始");
8
9         String[] relatedApis = record.get("Related APIs").split(",");
10        //String[] Categories = record.get("Categories").split(",");
11        Set<String> comparedres = new HashSet<>();
12        Set<String> Query = new HashSet<>();
13
14        for (String a : relatedApis) {

```

```

15         if(undirectedGraph.containsVertex(a)){
16             comparedres.add(a);
17             Query.add(PrimaryCategory.get(a));
18         }
19
20     }
21     /* for (String b : Categories) {
22         Query.add(b);
23     }*/
24     SteinerTree steinerTree1 = new
25     ↪ SteinerTree(undirectedGraph,PrimaryCategory);
26     long startTime = System.currentTimeMillis();
27     MidTree ress = steinerTree1.bulidSteinerTree(Query);
28     long endTime = System.currentTimeMillis();
29     if (ress != null) {
30         System.out.println(" 第"+now+" 个任务返回 api 共耗时"+
31         ↪ (endTime - startTime)+" 毫秒");
32         System.out.println(" 第"+now+" 个任务返回的群 Steiner Tree
33         ↪ 的权重和为: "+ ress.getTotleweight());
34         Vector<String> resV = ress.getVertexSetofMidTree();
35         System.out.println(" 第"+now+" 个任务返回的群 Steiner Tree
36         ↪ 的顶点个数为:"+ resV.size());
37         if(resV.size()<=comparedres.size())
38         ↪ lessthan.incrementAndGet();
39         if (comparedres.containsAll(resV)) {
40             int nowright = right.incrementAndGet();
41             System.out.println(" 第"+now+" 个任务命中"+" 这是第" +
42             ↪ nowright+" 个命中");
43         }
44         else{
45             System.out.println(" 第"+now+" 个任务未命中");
46         }
47     }
48     else{
49         System.out.println(" 第"+now+" 个任务未命中");
50     }
51 }
52 });

```

```

46
47     }
48
49     executor.shutdown();
50     try {
51         executor.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
52     } catch (InterruptedException e) {
53         e.printStackTrace();
54     }
55     double hitrate = right.doubleValue()/totle.doubleValue();
56     System.out.println(" 一共命中了"+right.get()+" 个任务, "+" 命中率为" +
    → hitrate*100+"%");
57     System.out.println(" 若返回的 api 数少于所给的 api 数也算命中, 则命中
    → 率为"+(lessthan.doubleValue()/totle.doubleValue())*100 + "%");
58     System.out.println(" 任务结束");
59 } catch (IOException e) {
60     e.printStackTrace();
61 }

```

3.2 测试方案

- 读取 mashup.csv 中的 relatedApis 作为验证结果, 然后将查询这些 api 所属的种类作为查询的关键词, 使用这些关键词调用算法, 将 relatedApis 与算法返回的结点做比较, 一致则说明命中。
- 一共命中了 1457 个任务, 命中率为 22.58
- 若返回的 api 数少于所给的 api 数也算命中, 则命中率为 80.04029133736246
- 每个任务的平均耗时为 4054.669929669923(ms)
- 总用时 20 分钟, 显著的减少了运行时间。

3.3 部分结果如下所示

- 第 4818 个任务返回 api 共耗时 336 毫秒
- 第 4818 个任务返回的群 Steiner Tree 的权重和为: 0.5
- 第 4818 个任务返回的群 Steiner Tree 的顶点个数为:2
- 第 4818 个任务命中这是第 1056 个命中

- 第 4578 个任务返回 api 共耗时 35202 毫秒
- 第 4578 个任务返回的群 Steiner Tree 的权重和为: 6.56212121212125
- 第 4578 个任务返回的群 Steiner Tree 的顶点个数为:18
- 第 4578 个任务未命中
- 第 4840 个任务返回 api 共耗时 85 毫秒
- 第 4840 个任务返回的群 Steiner Tree 的权重和为: 0.5
- 第 4840 个任务返回的群 Steiner Tree 的顶点个数为:2
- 第 4840 个任务命中这是第 1063 个命中
- 第 4849 个任务返回 api 共耗时 8 毫秒
- 第 4849 个任务返回的群 Steiner Tree 的权重和为: 0.0
- 第 4849 个任务返回的群 Steiner Tree 的顶点个数为:1
- 第 4849 个任务命中这是第 1066 个命中
- 第 4708 个任务返回 api 共耗时 21958 毫秒
- 第 4708 个任务返回的群 Steiner Tree 的权重和为: 4.75
- 第 4708 个任务返回的群 Steiner Tree 的顶点个数为:12
- 第 4708 个任务未命中

第6050个任务返回api共耗时58127毫秒
第6050个任务返回的群Steiner Tree的权重和为: 86.85119047619048
第6050个任务返回的群Steiner Tree的顶点个数为:115
第6050个任务未命中
第6294个任务返回api共耗时36939毫秒
第6294个任务返回的群Steiner Tree的权重和为: 14.124999999999998
第6294个任务返回的群Steiner Tree的顶点个数为:28
第6294个任务未命中
第5848个任务返回api共耗时96702毫秒
第5848个任务返回的群Steiner Tree的权重和为: 26.32784250063662
第5848个任务返回的群Steiner Tree的顶点个数为:47
第5848个任务未命中
第6451个任务未命中
第6438个任务返回api共耗时66549毫秒
第6438个任务返回的群Steiner Tree的权重和为: 19.98053391053391
第6438个任务返回的群Steiner Tree的顶点个数为:49
第6438个任务未命中
超时未收敛
第6244个任务未命中

图 2: 预测结果输出

第6450个任务返回api共耗时30毫秒
第6450个任务返回的群Steiner Tree的权重和为: 0.0
第6450个任务返回的群Steiner Tree的顶点个数为:1
第6450个任务未命中
一共6453任务,第6451个任务开始
第6404个任务返回api共耗时3100毫秒
第6404个任务返回的群Steiner Tree的权重和为: 0.5
第6404个任务返回的群Steiner Tree的顶点个数为:2
第6404个任务未命中
一共6453任务,第6452个任务开始
第6452个任务返回api共耗时23毫秒
第6452个任务返回的群Steiner Tree的权重和为: 0.0
第6452个任务返回的群Steiner Tree的顶点个数为:1
第6452个任务命中这是第1456个命中
一共6453任务,第6453个任务开始
第6453个任务返回api共耗时13毫秒
第6453个任务返回的群Steiner Tree的权重和为: 0.0
第6453个任务返回的群Steiner Tree的顶点个数为:1
第6453个任务未命中
第6416个任务返回api共耗时2851毫秒
第6416个任务返回的群Steiner Tree的权重和为: 0.5
第6416个任务返回的群Steiner Tree的顶点个数为:2
第6416个任务未命中

图 3: 预测结果输出

一共6453任务,第5739个任务开始
第5739个任务返回api共耗时2毫秒
第5739个任务返回的群Steiner Tree的权重和为: 0.0
第5739个任务返回的群Steiner Tree的顶点个数为:1
第5739个任务命中这是第1276个命中
一共6453任务,第5740个任务开始
第5676个任务返回api共耗时6956毫秒
第5676个任务返回的群Steiner Tree的权重和为: 0.6666666666666666
第5676个任务返回的群Steiner Tree的顶点个数为:3
第5676个任务未命中
一共6453任务,第5741个任务开始
第5741个任务返回api共耗时12毫秒
第5741个任务返回的群Steiner Tree的权重和为: 0.0
第5741个任务返回的群Steiner Tree的顶点个数为:1
第5741个任务未命中
一共6453任务,第5742个任务开始
第5686个任务返回api共耗时5788毫秒
第5686个任务返回的群Steiner Tree的权重和为: 0.125
第5686个任务返回的群Steiner Tree的顶点个数为:2
第5686个任务未命中
一共6453任务,第5743个任务开始
第5743个任务返回api共耗时24毫秒
第5743个任务返回的群Steiner Tree的权重和为: 0.0
第5743个任务返回的群Steiner Tree的顶点个数为:1
第5743个任务命中这是第1277个命中
一共6453任务,第5744个任务开始
第5744个任务返回api共耗时18毫秒
第5744个任务返回的群Steiner Tree的权重和为: 0.0
第5744个任务返回的群Steiner Tree的顶点个数为:1
第5744个任务未命中

图 4: 预测结果输出