Java 大作业【图上的搜索算法】项目文档

小组成员：胡天择、张沛、曾博湛

【亮点】

1. API 和 Key 同时视为图的结点，以及对应权值方案（详见问题抽象-改进结构）
2. 图的搭建（详见任务分步 2）
3. 额外注意的问题
4. 测试结果

【任务描述】

实现一个面向多关键词查询的子图搜索模块。对于给定一个数据图和一个查询需求（多个关键词），利用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）。

【任务分析】

在 mashup.csv 中包括了众多真实的 Mashup 应用及其调用过的 API、所属类别关键词；
在 api.csv 中包括了众多 api 和各个 api 所属类别关键词。
首先，需要提取两部分的信息：

1. API 之间的兼容性（来自 mashup.csv）；
2. API 及其关键词（来自 api.csv）。

随后，利用这些信息，搭建图模型，实现搜索最小斯坦纳树算法，找出能够覆盖给定关键词集合、兼容度最高的 API 组合。
最后，测试模型和算法的性能：

1. 有效性维度

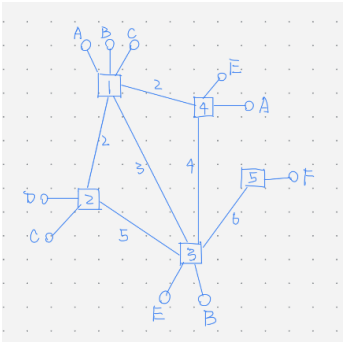
将真实 Mashup 应用的关键词作为输入，观察输出的 API 结果与其实际调用的 API（视为一种标准）是否一致，若一致（Hit）则认为输出结果是一个好的结果，计算 Hit Rate。

2. 效率维度

记录搜索的时间开销 Computation time。

【问题抽象】

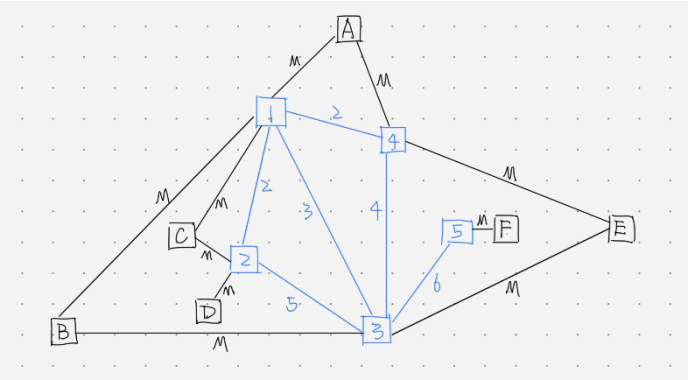
基础结构：



图中，
方块代表 API 结点，
圆圈代表对应 API 的功能，
存在边代表相连 API 之间存在协作，
边权代表 API 之间的兼容程度，权值越小兼容程度越高

在这种结构下，实现题目需求的思路是：对于需求功能集合中的每个功能，选择一个拥有该功能的 API（存在中多可能的组合），再对选中的 API 集合生成最小斯坦纳树，最后再比较所有不同组合的最小斯坦纳树的边权总和，找到总和最小一种。

改进结构：



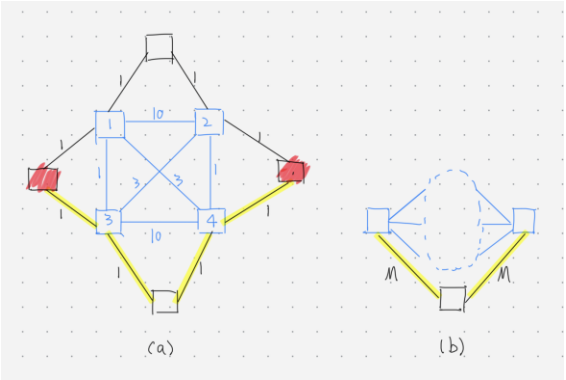
将功能关键词也作为图的结点，功能结点与具有该功能的 API 之间存在无向边，并且赋权一个合适的较大值 M。如此，将所需要的功能关键词作为给定的结点集合，在图中搜索最小斯坦纳树即可。

权值取为较大值的原因如下：

我们所需要的每一个功能至少由一个 API 来实现，体现在斯坦纳树中即：每个需要的功能都至少有一条连接到 API 的边。即最小斯坦纳树的权重和最少为 nM，n 为需求功能个数，M 为 API 和 Key 之间的边权。

由于最小斯坦纳树中只有在 API 之间连通的边具有实际意义（“最小”对应于“最兼容”，而兼容只是 API 之间连通的边的概念，API 与 Key 之间连通的边无所谓兼容与否），而功能之间是不连通的，只通向 api，当我们将几个功能作为关键点时，为了连通它们，一定会有一条它们指向某个 api 的边。

同时，为防止出现如下图的极端情况，



即斯坦纳树中不能出现类似图中(b)的“捷径”结构，“越过”两个 API 之间代表兼容性的路径的部分。在给 API 与 Key 之间连通的边赋一个较大权值后，在生成最小斯坦纳树时，这种边的权对总权重影响巨大，所以斯坦纳树中同一个 Key 肯定不会有两条连向 API 的边。

综上所述，生成的树中每个 Key 都会有且仅有一条连向 API 的边，再由 API 之间的边进行连通，最后全部连通。这与基础结构所能实现的效果相同，但是只需寻找一次最小斯坦纳树。

【额外注意的问题】

1. API 的附加功能 update

在观察 mashup.csv 后，我们发现了问题。对于部分 Mashup 应用，其实际选用的 api，如果只有 api.csv 中的功能，并不能实现应用中的所有功能，对于这些 API，除了赋予 api.csv 中功能外，需要加上附加功能，如下例：

Name	Related APIs	Categories
BanksNearMe	Google Maps	Mapping,Banking,Financial

	A	B	C	E
		Name	Primary Category	Secondary Categories
0		Twilio	Telephony	Cloud,Text-to-Speech,Voice,Webhooks
1		del.icio.us	Bookmarks	Social
2		Yahoo Maps	Mapping	Viewer
3		YouTube	Video	Media
4		Amazon Product Advertising	eCommerce	Advertising
5		Google Maps	Mapping	Viewer
6		Last.fm	Music	

在 api.csv 中，API [Google Maps]只有 Mapping 和 Viewer 的功能，而在 mashup.csv 中 Mashup 应用[BanksNearMe]仅仅用到了 API [Google Maps]，但又能够实现 Banking 和 Financial 的功能，为了能够匹配，需要给 API [Google Maps]附加这两个功能，否则在测试时不可能找到能够实现这些额外功能的 API。

而若是某个 Mashup 应用有两个及以上的 API，并且同样缺少部分功能，这种情况无法判定这部分功能是由哪个 API 实现的，因此给所有该 Mashup 应用用到的 API 都附加这部分额外功能，尽管这样会带来一些问题，在后续的第 3 点中继续讨论。

2. 单个需求功能

当需求功能数为 1 时，在改进结构下搜索最小斯坦纳树算法失效，因为此时斯坦纳树只会包含该 Key 结点，不需要引入连接到 API 的边。对于这种情况，选择与该 Key 相连，并且在 mashup.csv 中出现次数最多的 API 即可。

3. 对 hit 的处理

一般情况下，要求“输出的 API 结果与实际调用的 API 相等”记为一次 hit，而经过上述第 1 点的处理后，某些 API 势必能够实现一些本不能实现的功能，进一步导致，只需要用到更少的 API 就能够满足同样的需求功能。因此，将“输出的 API 结果包含于实际调用的 API 相等”记为一次 hit。

【任务分步】

1. 信息预处理；
2. 构建图结构；
3. 从 csv 到图；
4. 实现搜索算法；
5. 实验测试；

以下为对应分步的具体实现。

【具体实现】

1. 信息预处理

对 mashup.csv 和 api.csv 进行预处理，得到满足如下格式的 output 文件组：

output.csv：每一行代表一个 mashup 应用，首个元素为应用名称，后续元素为 API 名称；

output_2.csv：每一行代表一个 API，首个元素为 API 名称，后续元素为 API 能实现的功能；

output_3.csv：每两行代表一个 mashup 应用，奇数行的元素为 API 名称，偶数行的元素为 mashup 应用的功能；

2. 构建图结构

定义结点类作为图的顶点，图的边由顶点与其邻居结点定义（邻接表结构）。其中顶点有属性 name，label 和 neighbors，label 用于标记该结点是 api 结点还是关键词结点；neighbors 是一个 HashMap<Node, Weight>，key 值 Node 是该结点的邻居结点，而 value 值是结点到其邻居结点的权重。

结点类 GraphNode 的定义如下：

```
class GraphNode {
    private String name;
    public final Integer label;  //标记该结点，0 是 api, 1 是关键词
    private Map<GraphNode, Long> neighbors;
    public GraphNode(Integer label){
        this.label = label;
    }
    public GraphNode(String name, Integer label) {
        this.name = name;
        this.label = label;
        this.neighbors = new HashMap<>();
    }
    public  int get_degree(){
        return neighbors.size();
    }
    @Override
    public int hashCode() {// 获取对象标识 Hash 码
        return Objects.hash(this.name, label); // 获取 Hash 码
    }
    @Override
    public boolean equals(Object o){
        if (o == null || getClass() != o.getClass())// 对象类型不匹配
            return false;// 对象不同
        GraphNode o1 = (GraphNode) o;
        return this.name.equals(o1.name) && (Objects.equals(this.label, o1.label));
    }

    //禁止直接获取
    public Map<GraphNode, Long> getNeighborMap(){
        return this.neighbors;
    }
    //返回是否为邻居
    public boolean isNeighbor(GraphNode node){
        return this.neighbors.containsKey(node);
    }
    //获取所有邻居结点
    public Set<GraphNode> getNeighbors(){
        return this.neighbors.keySet();
    }
    public String getName(){
        return this.name;
    }
    public void setName(String name){
        this.name = name;
    }
}
```

图本身也是由一个 HashMap<Name, Node>储存，key 值 Name 储存结点（api 或关键词）的名称，而 value 值 Node 储存名称对应的结点。图实现了 IAPIGraph 接口，用于添加各种功能。图的具体实现如下：

```
//API 图功能
interface IAPIGraph{
    //根据功能需求组合，返回关键词结点数组
    public GraphNode[] getKeyNodes(String[] keys);

    //返回图中所有结点
    public List<GraphNode> getNodes();
```

```
    //向 API&Key 图中添加 API 结点
    public void addAPI(String name);

    //向 API&Key 图中添加 API 结点
    public void addAPI(GraphNode apiNode) throws Exception;

    //向 API&Key 图中添加 Key 结点
    public void addKey(String name);

    //向 API&Key 图中添加 Key 结点
    public void addKey(GraphNode keyNode) throws Exception;

    //连接 API&Key 中两个点并赋权值
    public void link(GraphNode startNode, GraphNode endNode, Long weight) throws Exception;

    //修改两点之间的权值
    public void updateWeight(GraphNode node1, GraphNode node2, Long weight) throws
Exception;

    //找出两个顶点之间边的权重，如果没有连接，则权重为 null
    public Long getWeight(GraphNode node1, GraphNode node2) throws Exception;

    //根据名称和 label 找出对应结点，没有就返回 null
    public GraphNode findNode(String name, Integer label);

    //判断图中是否存在该结点
    public boolean isContain(GraphNode node);

    //获取某个 API 具有的所有关键词
    public Set<GraphNode> get_keys(GraphNode node) throws Exception;

    //找出具有关键词的所有 API，需要提前判断是否为 key
    public Set<GraphNode> get_APIs(GraphNode node) throws Exception;

    //协作次数转换为权重
    public void weightTransfer();
}

// 图类
class AKGraph implements IAPIGraph{
    /*
     * AKGraph
     * 关键词和 api 抽象为相同顶点
     * api 之间的权重根据 mashup 内的协作次数计算
     * api 和关键词之间的权重设定为 Integer.MAX_VALUE
     *
     * */

    //储存 API 结点和 keys 结点
    Map<String, GraphNode> nodes;
    //图的边
    public AKGraph() {
        this.nodes = new HashMap<>();
    }
}
```

3. 从 csv 到图；

读取预处理后的 csv，从 mashup 应用使用的 API 入手建立 API 结点之间的边。

首先，读取 output.csv 建立图中的 API 结点以及对应边、权重（协作次数）

随后，读取 output_2.csv 建立各个 API 能够相连的 Key 结点以及对应边

最后，读取 output_3.csv 给各个 API 添加附加功能

```
interface ICreateAKGraph{
    public Long AK_WEIGHT = (long)Integer.MAX_VALUE;
    //读取 csv 文件，在这个函数内，直接完成建图
   public void createGraph(String path_cre, String path_lin, String path_upd);
```

```
    /*
     * 根据 app 和 api 协作关系完成图中的 api 部分
     *
     * 读取 mashup 中的 Name(app)和 Related APIs 两列
     * Related APIs 获取每个 api 名称，并向 graph.nodes 中添加结点
     * 再根据协作关系更新 graph.edges
     * */
    public void createAPIPart(IAPIGraph graph, String[] app_l_apis);

    /*
     * 完成连接 API 和 keys
     * 读取 api.csv 文件将已经建立的 APIPart 部分再添加 keys 部分
     * */
    public void link_API_Key(IAPIGraph graph, String[] api_l_keys);

    /*
     * 更新 API 和 keys 关系
     *
     * 再次遍历 mashup.csv 文件，根据表中的 Categories 列更新 api 和 keys 连接
     * */
    public boolean update_API_Key(IAPIGraph graph, String[] apis, String[] keys);
}
```

对接口进行实例化，则主要运用 API 图接口提供的方法进行实现，详见源代码。

4. 实现搜索算法

图建立后需要找到拥有所具有给定功能的最兼容 api 组合。需要的功能称为关键结点，在只需一个功能时不考虑 api 兼容性,即找到一个拥有该功能的 api。在需要至少两个功能时，需要一个算法，能够找到覆盖关键结点的，权值和最小的生成树。基于斯坦纳树的路径寻找算法,即找到包含给定结点的最小生成树，该生成树会覆盖所需功能和最小生成树上的 api，该算法即可以满足需求。

由于参考代码中只有一个 dp 数组表示存储状态压缩 DP 的结果，但项目需要记录选中的结点，所以第二步是添加代码记录找到斯坦纳树过程中选择的结点。小组用二维数组 path 来实现，这个二维数组每个数组空间用来储存一个链表，与 dp 对应，代表当前状态下权重和最小时选中的结点，每次对 dp 更新时也对 path 更新。

在上文已经实现包含功能和 api 结点的图结构，而斯坦纳树求解需要使用索引号进行操作。所以需要进行图的转换，即将已实现的图结构转换为带索引号的图结构。

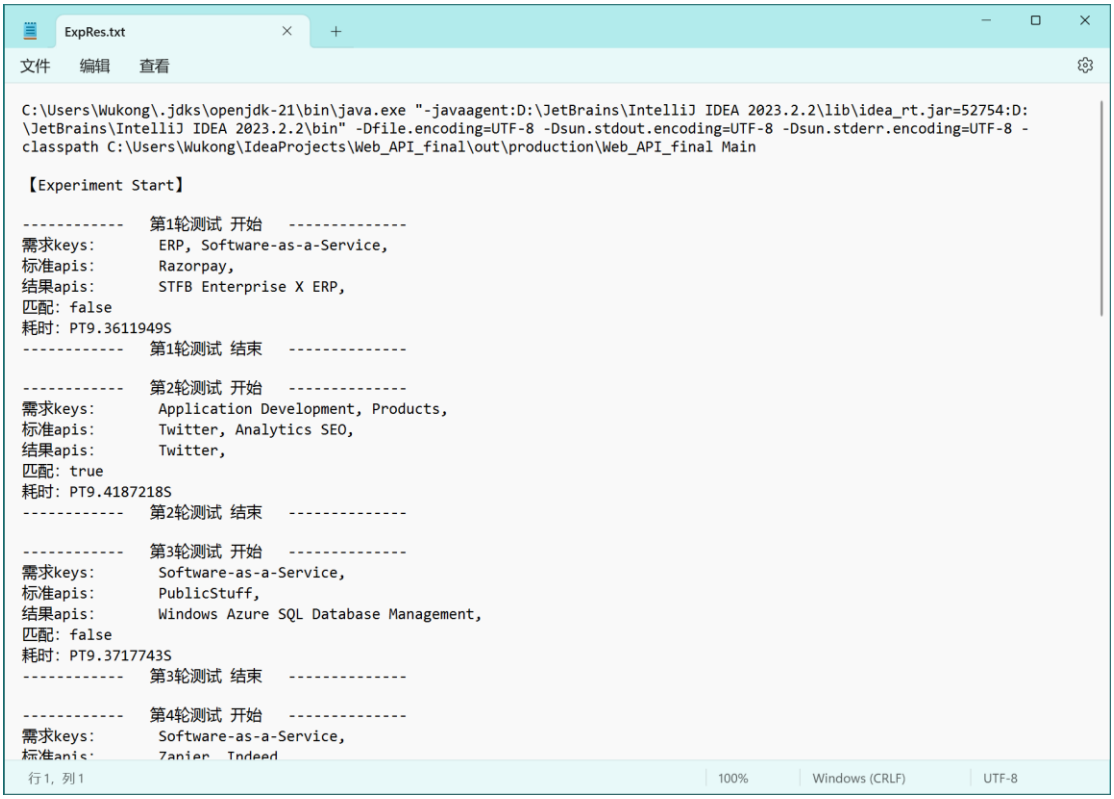
具体实现代码如下:

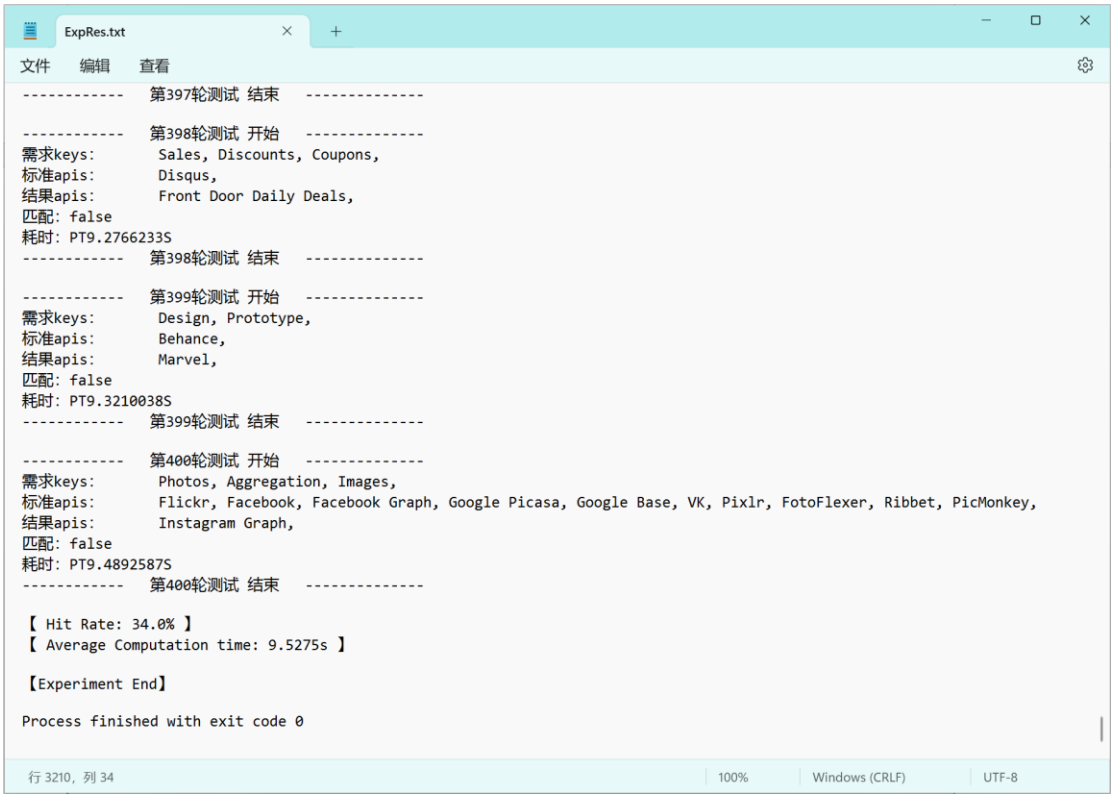
```
//斯坦纳树核心代码
private List<Integer> getMultiKeyPath(int n, int k, int[] x){
    int y = -1;//y 主要用于标识输入中的最后一个特殊结点，并在计算和输出最短路径和权重时作为参考点。
    long[][] dp = new long[n][1 << (k + 1)];//创建一个动态规划（DP）数组 dp，用于存储到达每个结点的最小权重。
    List<Integer>[][] path = new ArrayList[n][1 << k];//创建了一个链表数组 path 来存储到达每个结点的最佳路径。
    for (int i = 0; i < n; i++) {//dp 数组和 path 数组的初始化
        Arrays.fill(dp[i], Long.MAX_VALUE);
        for (int j = 0; j < (1 << k); j++) {
            path[i][j] = new ArrayList<>();
        }
    }
    //对关键结点进行处理
    for (int i = 0; i < k; i++) {
        dp[x[i]][1 << i] = 0;//关键结点在只连通它自己的状态下最短距离为 0
        path[x[i]][1 << i].add(x[i]); //关键结点在只连通它自己的状态下路径为自己
        y = x[i];//标识输入中的最后一个特殊结点，输出时使用
    }
    // 使用状态压缩求解最小路径和以及路径
    // 遍历从单个结点开始，直到满足所有关键结点都连同的所有可能状态
    for (int s = 1; s < (1 << k); s++) {
        // 遍历以结点 i 为根结点，状态为 s 时的情况
        for (int i = 0; i < n; i++) {
```

```
            //  遍历 s 二进制的所有子集，执行 i 的度数大于 1 时的操作，对于每个状态，更新 dp 数组和 path 数组，以找到包含所有必要关键结点的最短路径。
            for (int t = s & (s - 1); t > 0; t = (t - 1) & s) {
                if (dp[i][s] > dp[i][t] + dp[i][s ^ t]) {
                    dp[i][s] = dp[i][t] + dp[i][s ^ t];
                    path[i][s].clear();
                    //path[i][t]和 path[i][s ^ t]分别是到达结点 i 的两个不同状态 t 和 s ^ t 的最优路径。合并这两条路径，存储在 path[i][s] 中，这代表在状态 s 下到达结点 i 的新的最优路径
                    mergePaths(path[i][s], path[i][t], path[i][s ^ t]);
                }
            }
        }
        deal(s, n, dp, path, vNgraph);//处理状态 s 下的最短路径情况
    }
    return path[y][(1 << k) - 1];
}
//处理状态 s 下的最短路径情况
private void deal(int s, int n, long[][] dp, List<Integer>[][] path, List<List<VNode>> graph) {
    PriorityQueue<long[]> pq = new PriorityQueue<>(Comparator.comparingLong(o -> o[1]));//建立优先队列，数组中索引值为 1 的数字越小，优先级越高
    boolean[] vis = new boolean[n];// 记录已经访问的结点
    // 遍历所有结点，将所有满足当前状态的结点加入优先队列
    for (int i = 0; i < n; i++) {
        if (dp[i][s] != Long.MAX_VALUE) {
            pq.add(new long[]{i, dp[i][s]});
        }
    }
    // 使用 Dijkstra 算法求解最短路径
    while (!pq.isEmpty()) {
        long[] tmp = pq.poll();
        int u = (int) tmp[0];
        if (vis[u]) continue;
        vis[u] = true;
        for (VNode edge : graph.get(u)) {
            int v = edge.to;
            long newWeight = tmp[1] + edge.weight;
            if (newWeight < dp[v][s]) {
                dp[v][s] = newWeight;
                pq.add(new long[]{v, dp[v][s]});
                path[v][s].clear();

                //将所有在状态 s 下到达结点 u 的最优路径添加到 path[v][s]  中。这意味着结点 v 的最优路径在当前情况下是通过结点 u 的路径再加上从 u 到 v 的这一步。
                path[v][s].addAll(path[u][s]);
                if (!path[v][s].contains(v)) { // 防止重复添加结点

                    //如果结点 v 还不在路径中，则将其添加到路径列表 path[v][s]的末尾。这代表结点 v 现在是在状态 s 下的最优路径的一部分。
                    path[v][s].add(v);
                }
            }
        }
    }
}
// 合并两条路径，确保不会有重复结点
private void mergePaths(List<Integer> mergedPath, List<Integer> path1, List<Integer> path2)
{
    Set<Integer> set = new HashSet<>(path1);
    mergedPath.addAll(path1);
    for (int node : path2) {
        if (!set.contains(node)) {
            mergedPath.add(node);
        }
    }
}
```

【测试结果】

由于时间有限，在此仅对前 400 组 mashup.csv 中的功能集合进行测试，结果如下：





可以看到，尽管根据每组需求 keys 仅生成一组结果 apis，

但命中率仍有为 34%，平均耗费时间为 9.5275s。