图论大作业 1

> 使用 `Bellman-Ford` 算法求多条最短路、次短路并分析计算量

算法

最短路算法

采取 `Bellman-Ford` 算法，对该算法使用队列进行优化，构成所谓的 `Shortest Path Faster Algorithm` 即 `spfa` 算法

算法的主要思想以 `Bellman-Ford` 算法为基础，在于对于某一个节点v_i，如果与该节点相连的节点最短路值发生改变，那么该点对应也要进行更新。但是这种方式对于一个边比较多的算法来说有很多的多余计算次数。因此我们使用队列对其进行优化，具体算法如下：

我们首先将起始节点入队，初始化节点信息等，当队列不会空的时候，我们获取当前节点的队首元素，对该元素连接的所有边进行松弛操作，更新对应的节点的最短路信息，随后将所有更新数据的节点都加入队列，循环直到队列为空停止

具体算法与代码实现

> 使用 `Python` 进行编程

首先介绍一下文件的构成

```
+ spfa
+-- data
|   +-- graph-generated.csv
+-- graph
|   +-- __pycache__
|   +-- graph.py
|   +-- __init__.py
+-- README.md
+-- spfa.py
```

`data` 文件夹中存储的是数据

graph 文件夹存储的是图的数据结构，其中 graph.py 中定义了图的数据结构和读取 csv 文件数据的方法

spfa.py 文件为主文件，其中包含实现 spfa 算法、实现以及路径的回溯、计算、校验等等

数据结构

1. 图

```
class Graph:
    def __init__(self, edge=None):
        if edge is None:
            edge = []
        self.edge = edge
        self.vertex = []
        self.vertex_number = 0
```

其中 Graph.edge 记录边的信息，其记录形式如下：

```
Graph.edge[from][to] = value
```

将每一条边记录到数组中

Graph.vertex 记录点的信息，主要包括点的名称等等
Graph.vertex_number 记录点的数量

2. spfa 算法

```
class SPFA:
    def __init__(self, g: graph.Graph):
        # 存储图的信息
        self.graph = g
        # 存储前置节点信息
        self.pre_vertex = [{"ShortestPreVertex": [-1], "SecondShortestPreVertex": [-1]} for i in
                          range(g.vertex_number + 1)]
        # 存储入队次数
        self.count = [0 for i in range(g.vertex_number + 1)]
        # 存储该点是否在队列中
        self.flag = [False for i in range(g.vertex_number + 1)]
        # 存储最短路的前置节点
        self.distance = [sys.maxsize for i in range(g.vertex_number + 1)]
        # 存储单源点起点位置
        self.vertex = -1
```

多条最短路和次短路的算法描述

对于次短路的算法具体如下所述：

当对某一节点 i 进行松弛操作时进行记录：

- 如果到当前节点 i 的最短路与途径其他节点 j 之后再到当前节点 i 的路径距离相比较短，则不做操作
- 如果到当前节点 i 的最短路与途径其他节点 j 之后再到当前节点 i 的路径距离相同，那么将途径的节点 j 记录进入最短路前置节点的字典中，即

```
self.pre_vertex[i]["ShortestPreVertex"].append(j)
```

- 如果到当前节点 i 的最短路与途径其他节点 j 之后再到当前节点 i 的路径距离相比较长，那么将当前节点 i 的次短路的前置节点集合应该是之前的最短路的前置节点集合，最短路前置节点应该是 j ，即

```
self.pre_vertex[i]["SecondShortestPreVertex"] = self.pre_vertex[i]["ShortestPreVertex"]
self.pre_vertex[i]["ShortestPreVertex"] = [j]
```

最终记录形成字典，之后进行递归还原多级字典，并且计算、校验次短路经并将结果输出

> 经验证，最终结果与标准结果有一定的偏差，可能在于某些判断条件上的问题，随后时间允许的情况下会进行修复

时间复杂度

由于采用队列进行了优化，所以在稀疏图中，最好时间复杂度为$O(kE)$，其中 k 是一个常数；在稠密图中，最坏时间复杂度达到了为$O(VE)$。