人 工 智 能 课 程

报　　告

题　　目　**实现罗马尼亚度假问题**

班级学号　191181-20181001095

姓　　名　常文瀚

评　　分

中国地质大学（武汉）计算机学院

2020 年 10 月

一、题目描述

分别采用：宽度优先、深度优先、贪婪算法和 A*算法实现罗马尼亚度假问题。

二、基本思想

1、宽度优先

首先访问初始点 v 并将其标志为已经访问。接着通过邻接关系将邻接点入队。然后每访问过一个顶点则出队。按照顺序，访问每一个顶点的所有未被访问过的顶点直到所有的顶点均被访问过。

2、深度优先

从图中的某一个顶点 V 出发，访问此顶点后，依次访问顶点 V 的各个同层未访问过的邻接点，然后分别从这些邻接点出发，直至图中所有顶点都被访问到。该算法探索所有顶点的所有邻接点，并确保每个顶点只访问一次，没有访问两次的顶点。

3、贪婪算法

建立数学模型来描述问题，把求解的问题分成若干个子问题，对每一子问题求解，得到子问题的局部最优解，把子问题的解局部最优解合成原来解问题的一个解。

4、A*算法

使用一个评估函数 f(n)给每个节点估计他们的希望值，优先扩展最有希望的未扩展节点，避免扩展代价已经很高的节点。

三、软件设计

1、需求分析

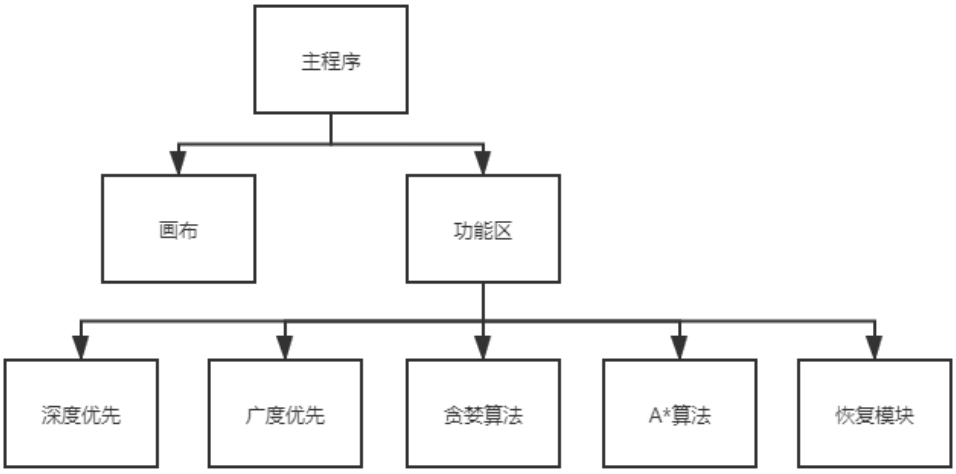
（1）实现四种算法对指定两个节点之间的最短路径求解。

（2）将求出的路径以 GUI 界面显示出来，达到更加直观的效果。

（3）能够在使用一种算法求出最短路径后恢复初始化状态。

（4）可以反复运行，不会崩溃。

2、总体设计

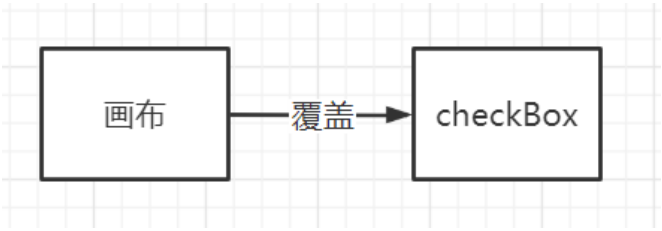


3.详细设计

主要程序分为画布以及功能区两个部分，其中功能区又具有通过深度优先搜索算法、广度优先算法、贪婪算法、A*算法搜索指定节点间最短路径的功能，并且添加了将画布恢复至初始状态的功能。整个程序使用 Python 语言，并使用了 PyQt 进行可视化处理。

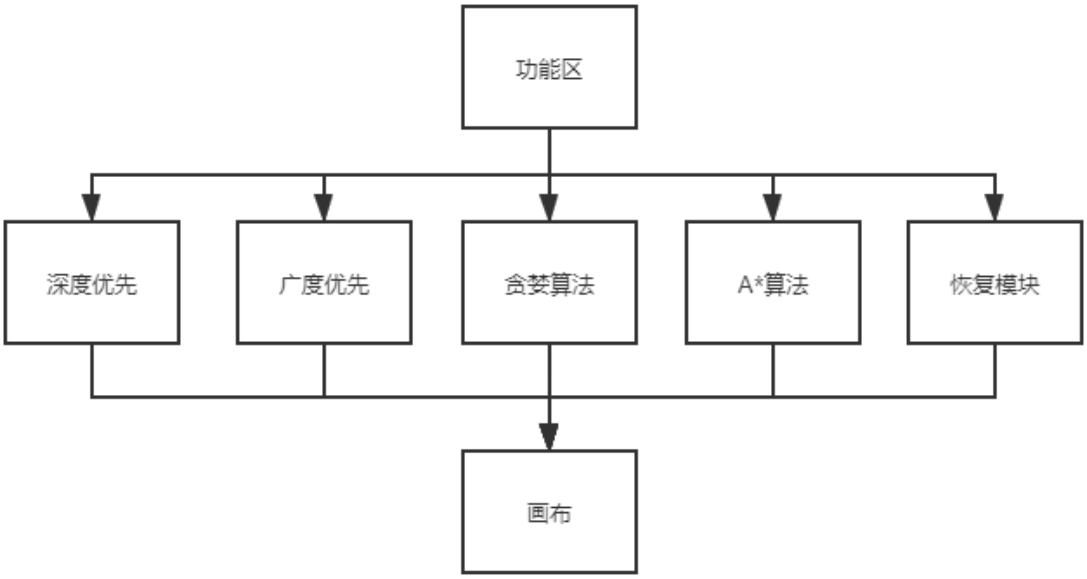
（1）画布部分：

画布部分先将 checkBox 控件按照原图对应的比例摆放，然后在能覆盖所有 checkBox 控件的区域内设置 QPainter 画布，使使用者不能直接操作 checkBox 的状态，即使人为操作不会影响到 checkBox 与函数的连接，具体结构如下：



（2）功能部分：

功能部分细分为五个部分，分别是：深度优先搜索算法、广度优先算法、贪婪算法、A*算法搜索指定节点间最短路径的功能和初始化的功能，用户可以直接选择想要使用的算法求解，但是在每次使用后必须选择恢复，使所有发生变化的控件和变量等数据恢复刚运行的状态，与此同时，在实现算法的功能还要注意算法与画布的沟通，使画布及时反映出程序状态的变化和最短路径，具体结构如下：



4、代码实现

（1）主要程序初始化：

这一部分代码通过继承 UI 界面类，并向其中添加算法实现函数，做到了界面与逻辑分离，使实现功能的代码更为简洁，易读性更高，代码更加安全（PyQt 更新界面时会将整个 py 文件重写，导致已经实现的算法函数和已经生成的数据被删除，但是做到界面与逻辑分离就可以避免这一点），代码如下：

```
class MainWindow(QMainWindow, Ui_MainWindow):
    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        self.setupUi(self)
        self.initialBox()
        #按键的槽函数
        self.signalAndConnection()
        #图的数据以及节点的标号
        self.readyPlaceHolder()
        # 为 A 星算法进行准备工作
        self.aStarPlaceHolder()
        # 为深度遍历进行准备工作
        self.dfsPlaceHolder()
        # 为广度遍历进行准备工作
        self.bfsPlaceholder()
        #为贪婪算法进行准备工作
        self.greedyPlaceHolder()
        # 添加画布以及绘画事件
        self.painterPlaceHolder()
        self.show()
```

（2）深度优先搜索算法实现部分：

```
def dfs(self,current,flag,flag2):
        self.run[current] = 1
        if current == 3:
            self.flag2 = True
        if self.flag == True and self.flag2 == True:
```

```
            print('[%d] ' % current, end='')
            self.dict[current].setChecked(True)
        if current == 10:
            self.flag = False
        ptr = self.head[current].next
        while ptr != None:
            if self.run[ptr.val] == 0:  # 如果顶点尚未遍历，
                self.dfs(ptr.val,self.flag,self.flag2)  # 就进行 dfs 的递归调用
            ptr = ptr.next
```

（3）广度优先搜索算法实现部分：

```
def bfs(self,current):
        self.enqueue(current)  # 将第一个顶点存入队列
        self.run[current] = 1  # 将遍历过的顶点设置为 1
        self.dict[current].setChecked(True)
        print('[%d]' % current, end='')  # 打印出该遍历过的顶点
        while self.front != self.rear:  # 判断当前的队伍是否为空
            current = self.dequeue()  # 将顶点从队列中取出
            tempnode = self.Head[current].first  # 先记录当前顶点的位置
            while tempnode != None:
                if self.run[tempnode.x] == 0:
                    self.enqueue(tempnode.x)
                    self.run[tempnode.x] = 1  # 记录已遍历过
                    if self.flag3 == True:
                        print('[%d]' % tempnode.x, end='')
                        self.dict[tempnode.x].setChecked(True)
                    if tempnode.x == 10:
                        self.flag3 = False
                if self.flag3 == True:
                    tempnode = tempnode.next
                else:
                    break
```

（4）贪婪算法实现部分：

```
def Greedy(self, n, weights):
        # 创建一个 flag 数组，用于保存遍历情况
        flag4 = np.zeros(n, bool)
        # 创建一个 dist 数组，用于保存最短路径
        dist = np.array(weights[0])
        # 创建一个 prev 数组，用于保存对应的最短节点
        prev = np.zeros(n, int)
        # 将源节点放入集合 S 中
        flag4[0] = True
        for i in range(n - 1):
            temp = float('inf')
            u = 0
            for j in range(n):
                if not flag4[j] and dist[j] != 0 and dist[j] < temp:
                    u = j
                    temp = dist[j]
            flag4[u] = True
            for j in range(n):
                if not flag4[j] and weights[u][j] != 0:
                    if dist[u] + weights[u][j] < dist[j] or dist[j] == 0:
                        dist[j] = dist[u] + weights[u][j]
                        prev[j] = u
```

(5) A*算法实现部分：

```
def aStar(self, nodeNow, path, pathTest, pathWeight):
        while nodeNow != 10:
            print(nodeNow)
            for i in self.weightForAStar:
                if i[0] == nodeNow:
                    self.pathTest.append(i[1])
                    self.pathWeight.append(i[2] + self.disDict[self.get_key(self.numNameDict, i[1])])
                    # print(self.pathTest)
```

```
                # print(self.pathWeight)
                index = self.pathWeight.index(min(self.pathWeight))
            self.path.append(self.pathTest[index])
            self.pathTest = []
            self.pathWeight = []
            self.dict[nodeNow].setChecked(True)
            nodeNow = self.numNameDict[self.get_key(self.numNameDict, self.path[len(self.path) - 1])]
        self.dict[10].setChecked(True)
        self.aStarPlaceHolder()
```

（6）按钮与函数链接实现部分:

```
def signalAndConnection(self):
        self.dfsButton.clicked.connect(self.dfsDeal)
        self.bfsButton.clicked.connect(self.bfsDeal)
        self.greedyButton.clicked.connect(self.greedyTest)
        self.clearButton.clicked.connect(self.clearChecked)
        self.aStarButton.clicked.connect(self.clickTest)
```

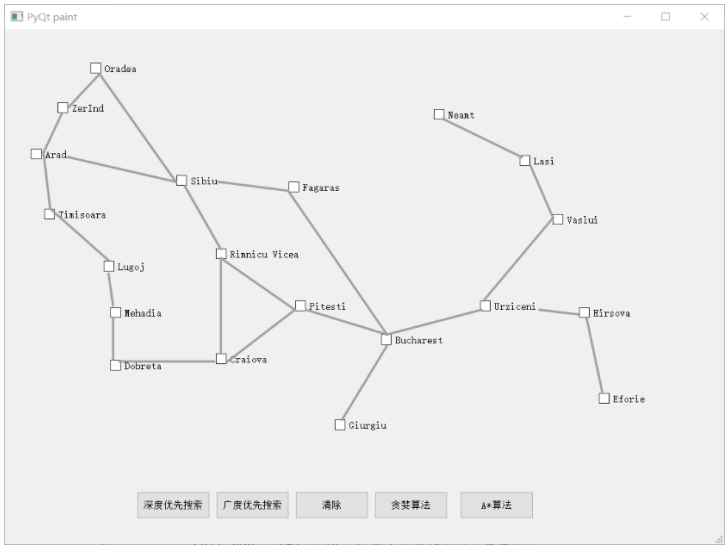
5、测试

经测试，程序可以成功将用四种算法搜索出的路径在 GUI 界面上显示出来，没有错误，具体结果请参考下文中的运行结果及分析。

四、运行结果及分析

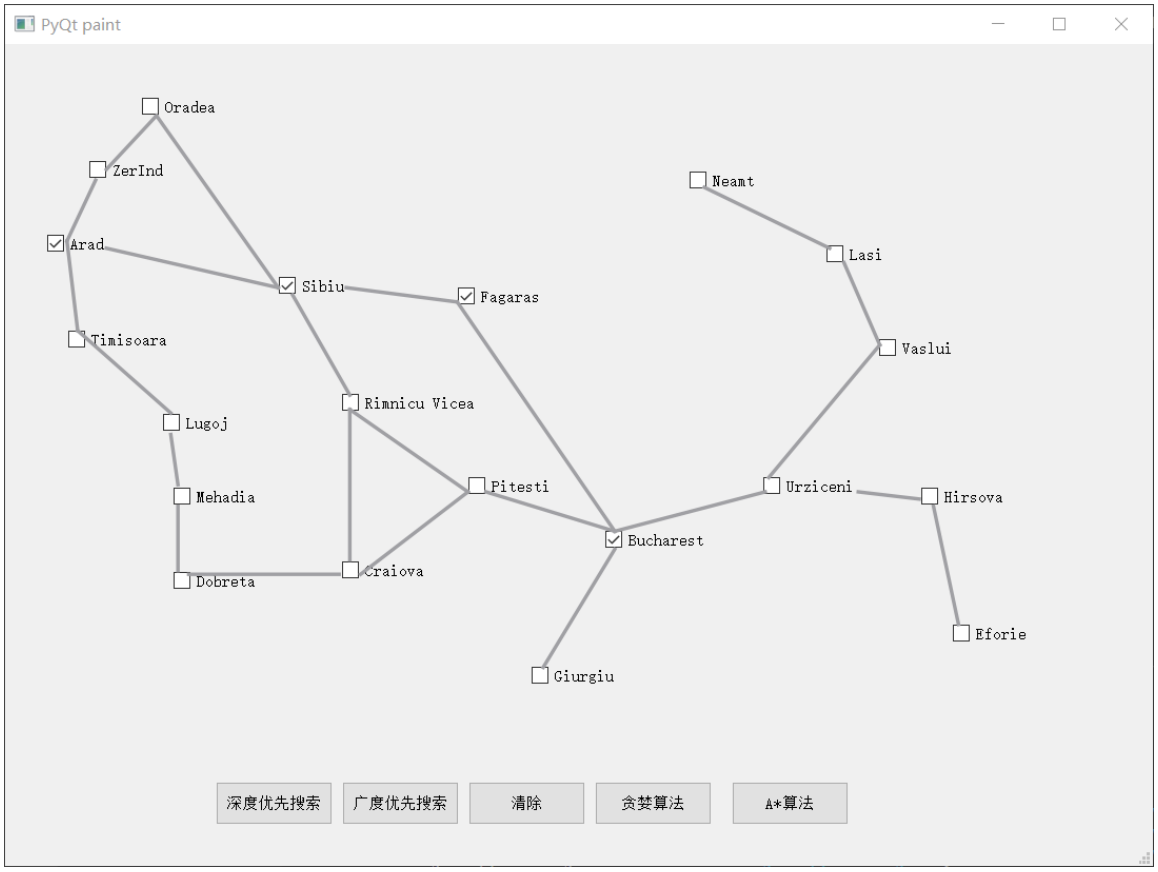
因为笔记本电脑算力比较强无法记录算法运行时间，所以这里把所有算法分别写入一个 Python 文件，放置在阿里云 ESC 云服务器上运行记录运行时间

（1）界面初始化



（2）深度优先搜索算法结果

GUI 界面结果：



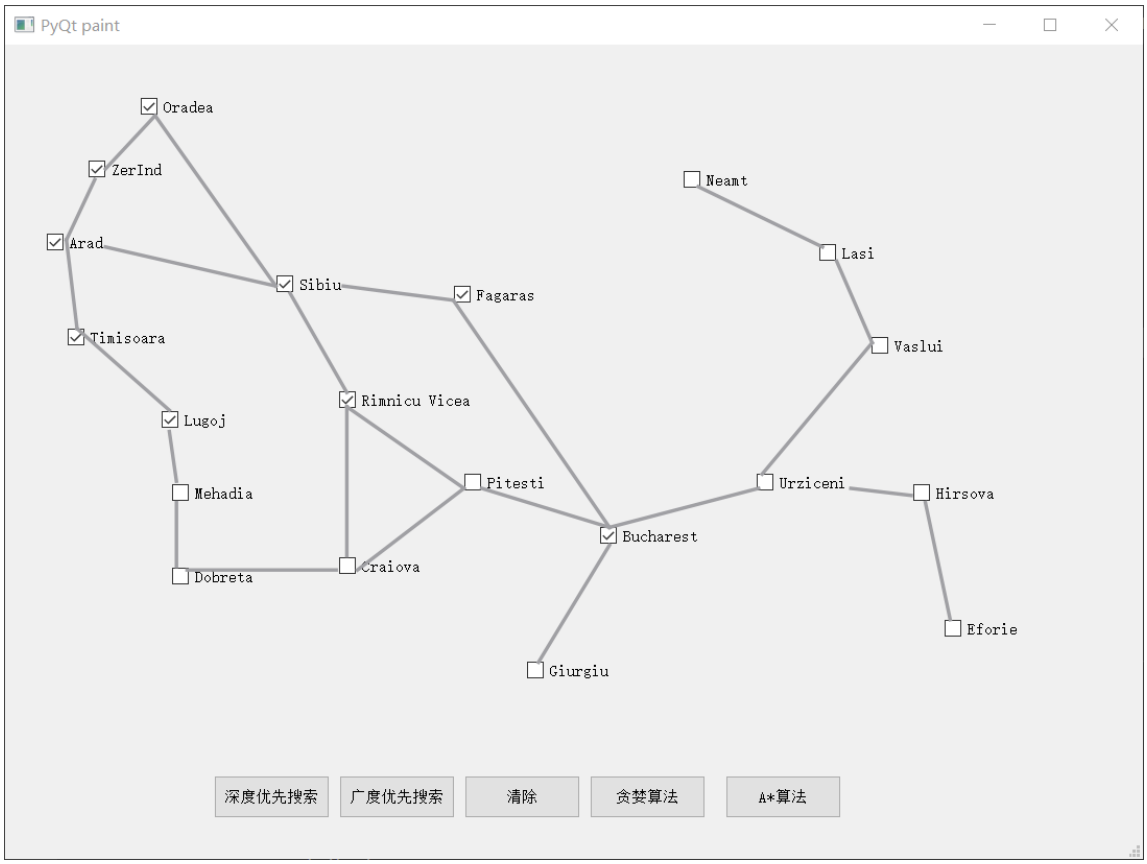
ESC 云服务器运行结果：运行一次算法花费了 0.000275 秒

```
root@iZ2zedc8d3ehprpmbkm5y7Z:~# python3 dfs.py
图的邻接表内容：
顶点 1=> [2] [12]
顶点 2=> [1] [3]
顶点 3=> [2] [12] [4]
顶点 4=> [3] [5]
顶点 5=> [4] [6]
顶点 6=> [5] [7]
顶点 7=> [6] [8]
顶点 8=> [7] [9] [13]
顶点 9=> [8] [10] [13]
顶点 10=> [9] [11]
顶点 11=> [10] [12]
顶点 12=> [1] [3] [11] [13]
顶点 13=> [8] [9] [12]
深度优先遍历的顶点：
[3] [2] [1] [12] [11] [10]
0.000275
```

广度优先搜索的时间复杂度为 O(V+E).即是图邻接表大小的线性函数。

（3）广度优先搜索算法结果

GUI 界面结果：



ESC 云服务器运行结果：运行一次算法花费了 0.000444 秒

```
root@iZ2zedc8d3ehprpmbkm5y7Z:~# python3 bfs.py
图的邻接表内容：
[2][12]
[1][3]
[2][12][4]
[3][5]
[4][6]
[5][7]
[6][8]
[7][9][13]
[8][10][13]
[9][11]
[10][12]
[1][3][11][13]
[8][9][12]
广度优先遍历的顶点：
[3][2][12][4][1][11][13][5][10][8][9][6][7]
0.000444
```

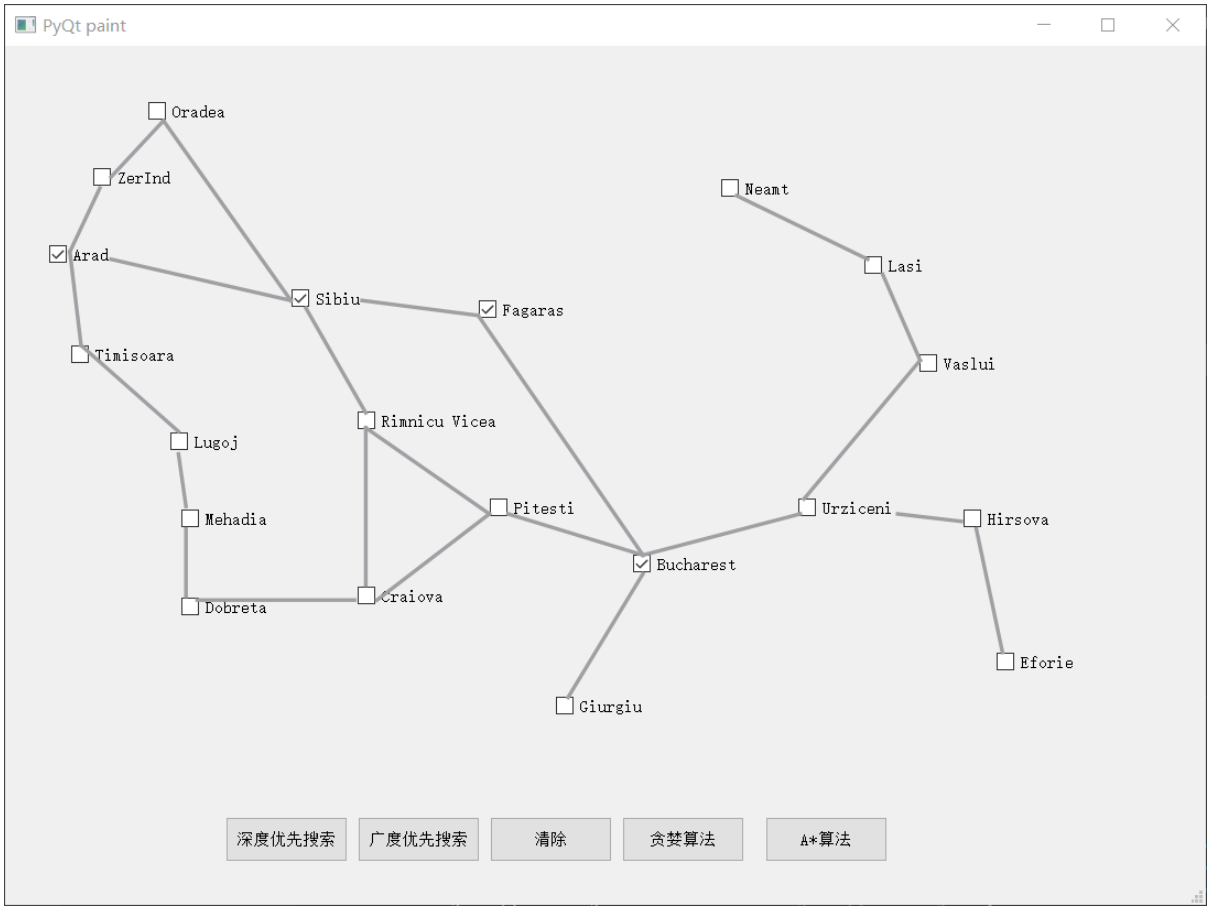
广度优先搜索的时间复杂度分析:

每个节点入栈和出栈各一次，时间均为 O(1),入栈和出栈总时间为 O(V);由于对每个节点的邻接表进行扫描，时间为 O(Adj[u]),总时间为 O(E);综上所示，广度优先搜索的时间复杂度为 O(V+E).即

是图邻接表大小的线性函数。

（4）贪婪算法搜索结果

GUI 界面结果：

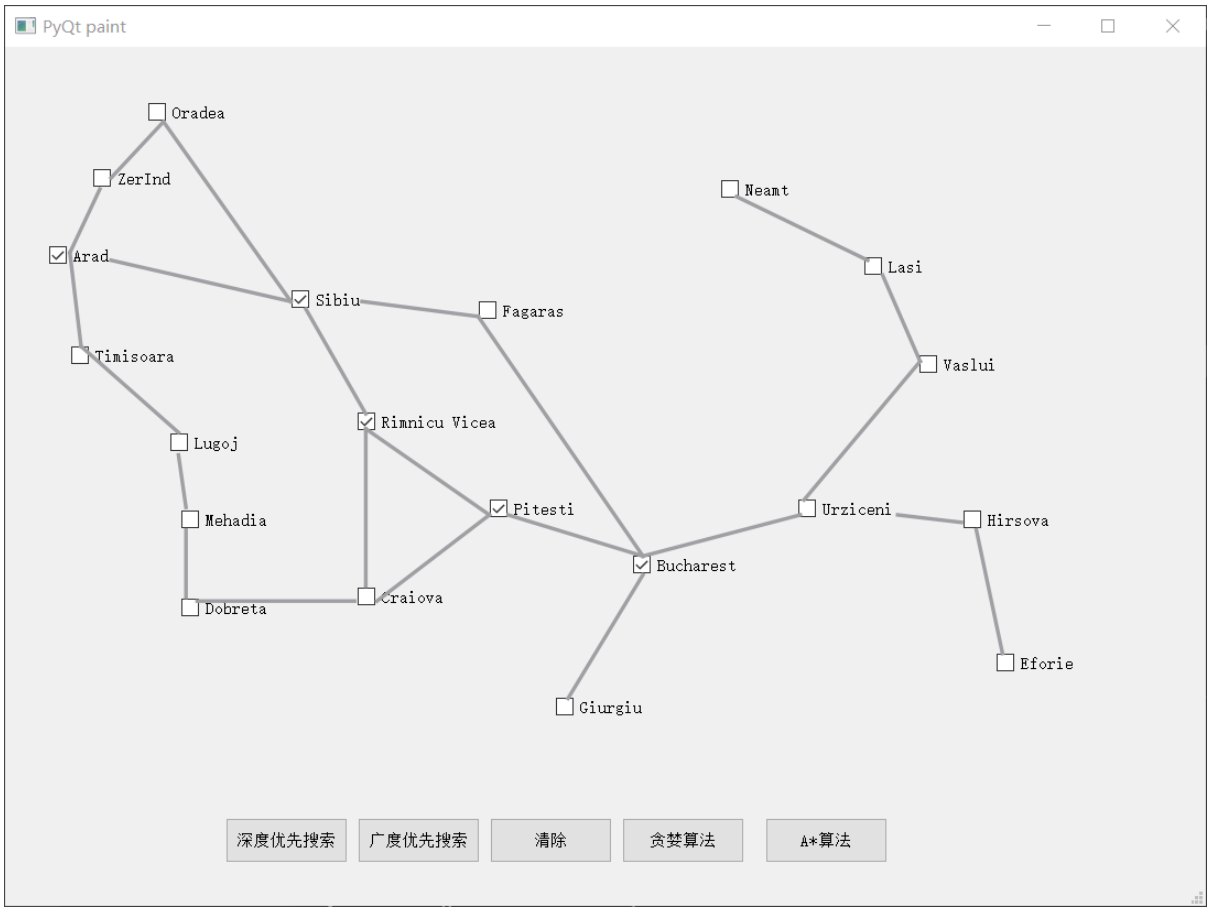


ESC 云服务器运行结果：运行一次算法花费了 0.000373 秒

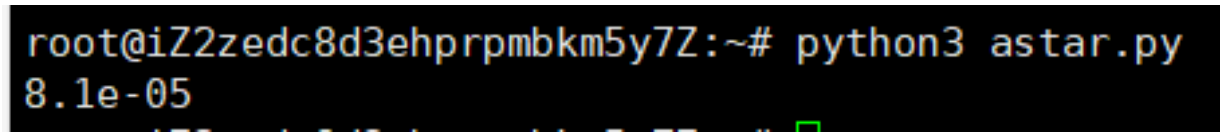
```
root@iZ2zedc8d3ehprpmbkm5y7Z:~# python3 dij.py
75: 1->2
146: 1->2->3
118: 1->4
229: 1->4->5
299: 1->4->5->6
374: 1->4->5->6->7
494: 1->4->5->6->7->8
632: 1->4->5->6->7->8->9
hello world
450: 1->12->11->10
239: 1->12->11
140: 1->12
640: 1->4->5->6->7->8->13
0.000373
```

（5）A*搜索算法结果

GUI 界面结果：



ESC 云服务器运行结果：运行一次算法花费了 8.1e-05 秒



五、使用说明

1、程序运行后可以直接选择希望使用的算法查找最短路径，但是在算法运行后必须点击清除恢复初始化状态。

六、小结

通过本次编程作业，我更加深入的了解了宽度优先搜索算法、深度优先搜索算法、贪婪算法和A*搜索算法，通过编程使在课堂上一些不懂的问题更加容易被解决，在自己一步一步的重新推导算法的时候算法的结构也更加清晰。与此同时，可视化的要求，使我更多的思考了界面变化与算法的链接应该如何更加正确，例如在算法运行使数据产生变化时，该在什么地方更新 GUI 画面的界面等问题。在查询更多的文献和网络信息的时候，我也更加深刻地认识到了算法世界的奇妙以及创造出各种算法的人的伟大，希望在这门课程的学习过程中，我也能学到更多的知识，为今后的学习和工作打好基础！

七、参考文献

[1] https://blog.csdn.net/u012878643/article/details/46723375

[2] https://blog.csdn.net/tianbwin2995/article/details/51094152

[3] https://blog.csdn.net/ccystewart/article/details/90143543

[4] https://blog.csdn.net/tintinetmilou/article/details/78962098

[5] https://blog.csdn.net/xc_zhou/article/details/80837850

[6] https://www.jianshu.com/p/613ef51394ec

[7] https://blog.csdn.net/v_JULY_v/article/details/6093380

[8] https://www.youtube.com/watch?v=ibFvkG-7h38

[9] https://www.youtube.com/results?search_query=a*%E7%AE%97%E6%B3%95

[10] https://blog.csdn.net/qq_25424545/article/details/79885239

[11] https://www.cnblogs.com/Lao-zi/articles/6143036.html

[12] https://www.cnblogs.com/wujing-hubei/p/6375906.html

[13] http://www.pythontip.com/acm/post/1227

[14] https://www.pythonf.cn/read/121286

[15] https://blog.csdn.net/qq_38454977/article/details/72495742