路程规划问题 – 导航软件是怎么工作的

暑假到了，同学们都准备好暑假的出行计划了吗？怎样做一个最好的出行计划的呢？今天我们就来学习这类问题的解决方法。除了路程规划，现实生活中还有很多这类问题（比如导航设计、AlphaGo等棋类游戏AI的最优步数计算、迷宫设计和解答、电路板设计等等），学习它们的解决方法可是大有作用的呢。

假如你住在广州，准备坐高铁或火车到重庆去旅游。下面是一个中国内地大城市的铁路图 – 请问经过的站点数最少的路程是哪一条呢？

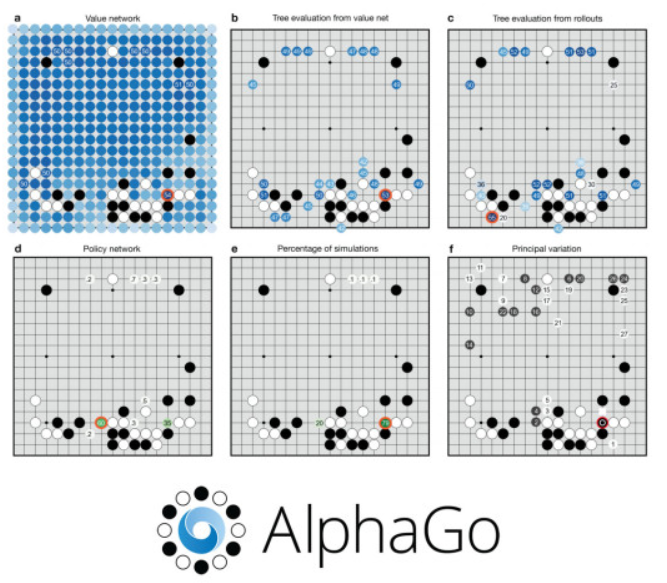


为了找到经过的站点数最少的路线，我们可以这样思考：

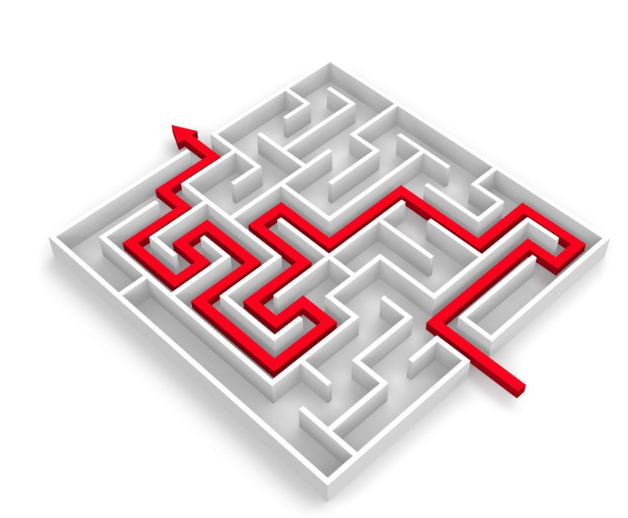
1. 广州坐1个站可以到达的城市有哪些？很明显，有深圳和长沙。--重庆不在里面，所以一个站到不了重庆。
2. 广州坐2个站可以到达的城市有哪些？深圳出发坐一个站不包括广州只有福州；长沙出发坐一个站不包括广州有贵阳、杭州、武汉。所以广州坐2个站可以到达的站点包括福州、贵阳、杭州、武汉，还是到不了重庆。
3. 广州坐3个站可以到达的城市有哪些？根据上面的规律，我们找到武汉坐一个站可以到合肥、郑州、重庆。耶，重庆出现了。所以我们知道广州到重庆站点数最少的路线是广州-长沙-武汉-重庆，一共要坐三个站。

类似这种找经过最少站点的问题叫做**最短路径问题。**这是生活中经常会遇到的问题，除了最常用到的出行规划，你还想到什么类似的问题吗？下面这些都是最短路径问题：

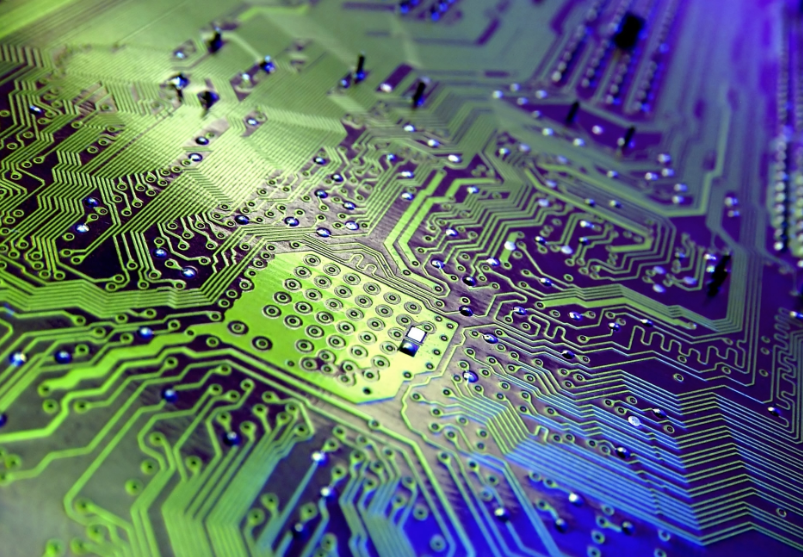
* 国际象棋、围棋等计算把对方将死的最少步数。还记得鼎鼎大名的AlphaGo吗？



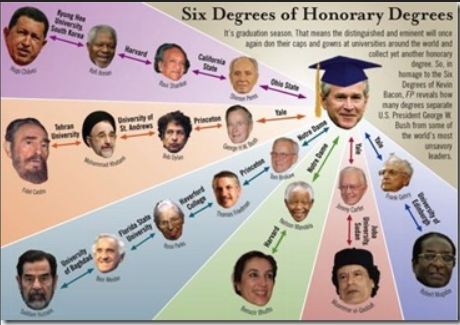
* 设计迷宫，以及解决如何走出迷宫的问题。



* 芯片/电路板的设计问题。



* 人脉关系度数计算 -- 有一个六度人脉关系理论：地球上所有人都可以通过六层以内的熟人链和任何其他人联系起来。就是说，最多通过六个人你就能够认识任何一个陌生人。微博等社交软件就是通过最短人脉关系来计算和推荐内容给用户。



等等

我们上面用到的解决方法叫做广度优先算法或宽度优先算法。它的过程可以看成是一个构造一颗倒立的大树的过程：

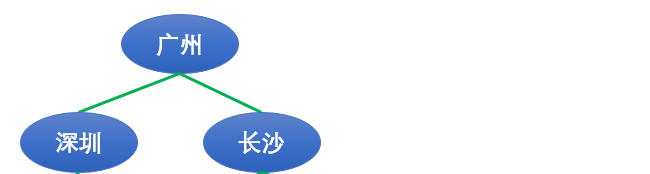
1. 先画根节点，即开始的点。
2. 接着依次访问所有未被访问的邻节点，把它们画成树叶并和根节点连起来。
3. 检查树叶里是否有要找的终点，如果没有的话，重复此过程，直到所有节点都被访问完。

我们用找广州到重庆的最小路径方法来示范:

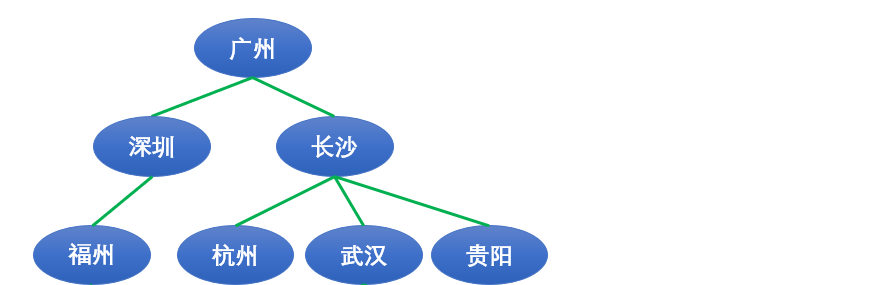
1. 画根节点：广州



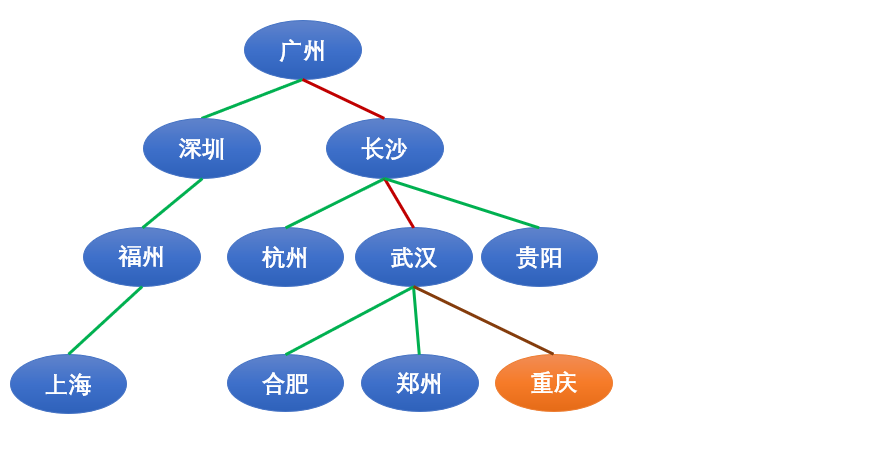
1. 画第一层树叶：昆明未被访问的邻节点 – 深圳和长沙



1. 重庆还没有出现，继续画下一层树叶：深圳的未被访问的邻节点是福州；长沙的未被访问邻节点有杭州、武汉、贵阳

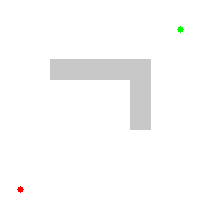


1. 重庆还没有出现，继续画下一层树叶：福州的未被访问邻节点是上海；杭州的邻节点都访问过了；武汉的未被访问邻节点有合肥、郑州、重庆。重庆出现了！树的构造也到此为止，最短路径也找到了：



通常将这种由若干个顶点以及连接某些顶点的边所组成的图形称为图。这种图形通常用来描述事物之间的某种特定关系，用点代表事物，用连接两点的线表示相应两个事物间具有这种关系。

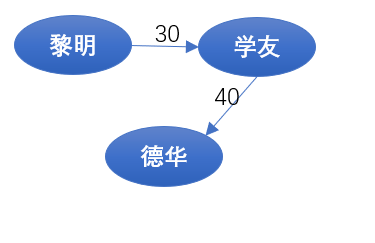
下面这个动画形象地演示了广度优先算法找两个点（红点到绿点）最短距离的过程：



这里再介绍几个常见的和图有关的名词：

**有向图**

如果顶点之间的边是有方向，表示一种先后顺序或依赖关系等，则称为有向图，比如黎明欠学友30块钱，学友欠德华40块钱，可以用下面的有向图表示，注意边的箭头：



**无向图**

相反，顶点之间的边是没有方向关系的，则称为无向图。比如我们的铁路图的例子里，相邻城市之间的边（铁路）是没有方向的，广州到深圳和深圳到广州的意义（站点数）是一样的。



**权重**

如果我们给边加上一些特别的数字描述，比如下面的图里我们给每个相邻站点之间的边加上一个数字代表列车在它们之间通行的时间（小时），则这些数字称为权重。不带权重的图称为非加权图。带权重的图称为加权图。我们一开始的线路图是非加权图。下面这个是加权图：

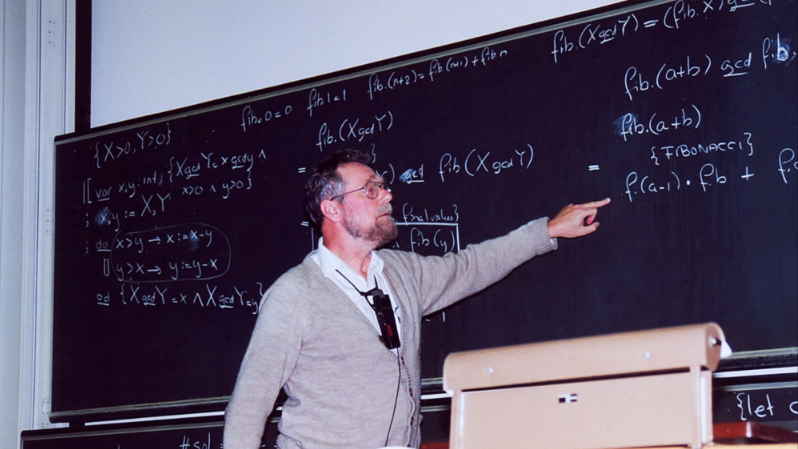


**图算法**

解决图问题相关的算法叫做图算法。

我们上面介绍的广度优先算法就是一种图算法，它可以用来解决非加权图最短路径问题。那么对于加权图怎么办呢？比如我们现在要找从深圳到上海的**时间最短**的路径，如果用广度优先算法，得出来得结果是深圳-福州-上海。但是这条路需要2+5+7=14小时，而如果我们选深圳-广州-长沙-杭州-上海的路径，只需要2+2+5+2=11小时。虽然经过的站点数多，但时间反而省了3个小时。

处理加权图最著名的一种算法叫做Dijkstra算法，一般翻译为迪杰斯特拉算法。该算法是荷兰计算机科学家迪杰斯特拉于1959 年提出的。下面这位就是Dijkstra本尊：

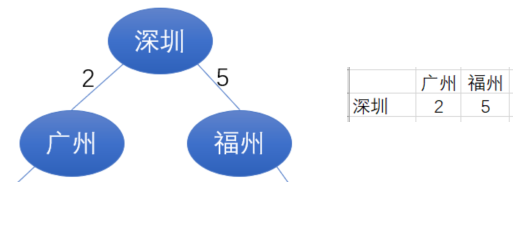


Dijkstra算法其实也是一种广度优先算法，但需要遍历所有节点。

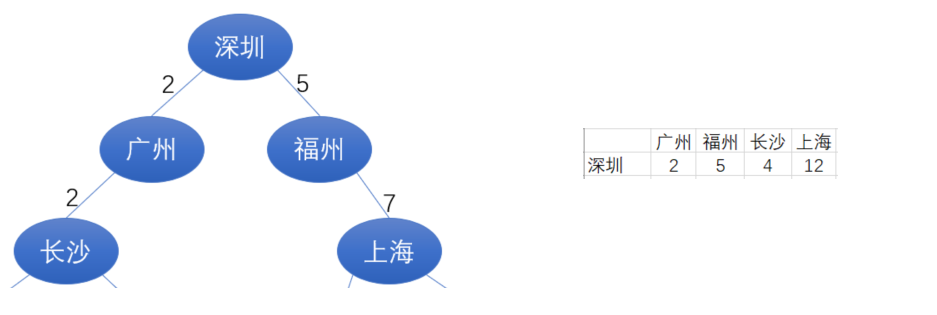
我们把铁路图简化成只有广州、福州、长沙、上海、武汉、杭州、南京、合肥、深圳的版本（如下图所示），来说明用迪杰斯特拉算法寻找深圳到上海最短时间路径的过程：



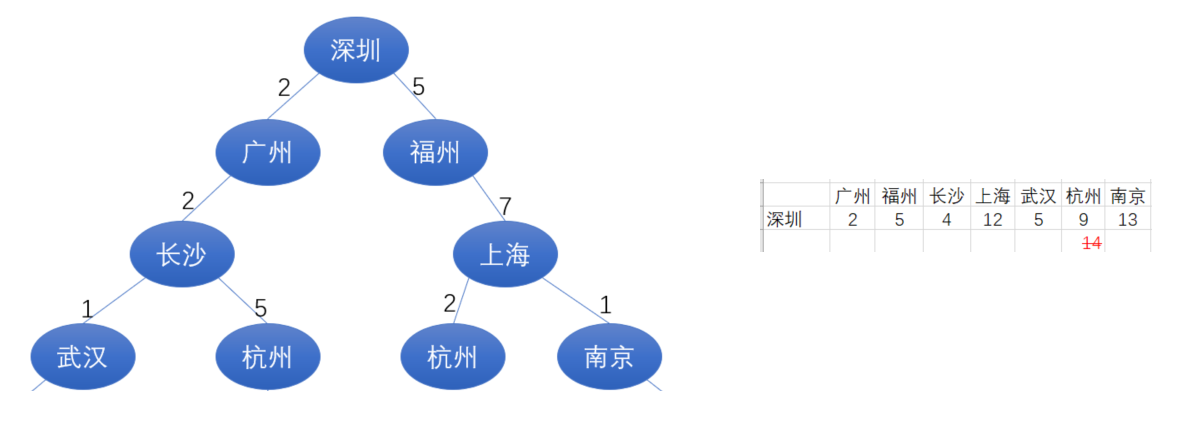
1. 计算起始节点到所有邻节点的时间，先把这个时间记录为起始点到所有邻节点的最短时间。即，第一步记录深圳-广州、深圳-福州的时间



1. 然后像广度优先一样，去遍历起始节点（深圳）的第一层叶子(广州、福州)的所有未访问邻节点，并记录时间。注意由起始点到第二层叶子的时间需要加上第一层的时间。这一步得出深圳-长沙、深圳-上海的时间。  
   这里已经得出了一个深圳-上海的时间了，但这并不一定是最短时间，还需要继续寻找。

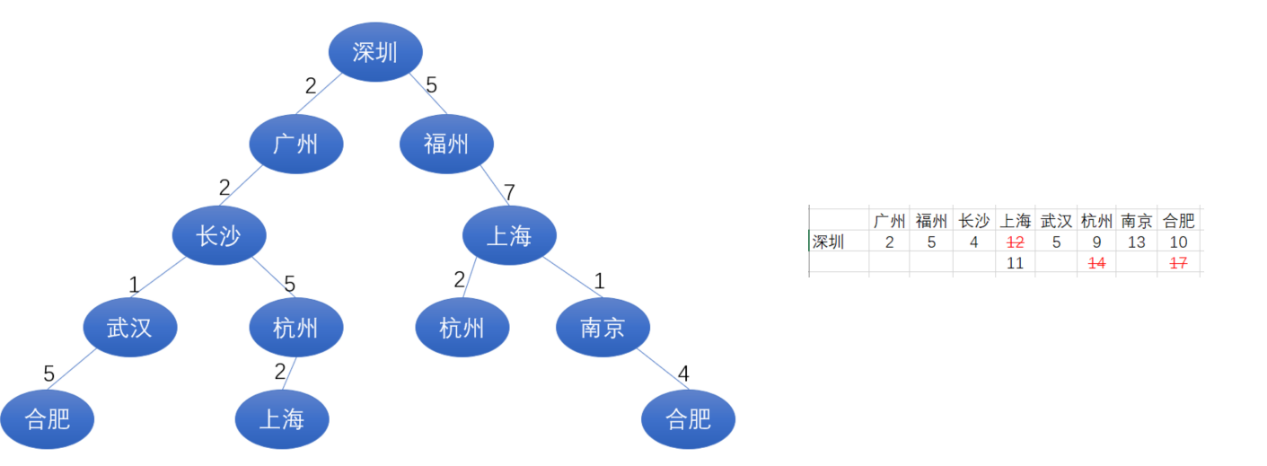


1. 接下来继续遍历下一层叶子，得到深圳-武汉、深圳-杭州、深圳-南京的时间。这一步杭州出现在左右两个不同的分支，我们需要比较两个分支的时间然后取最小值。

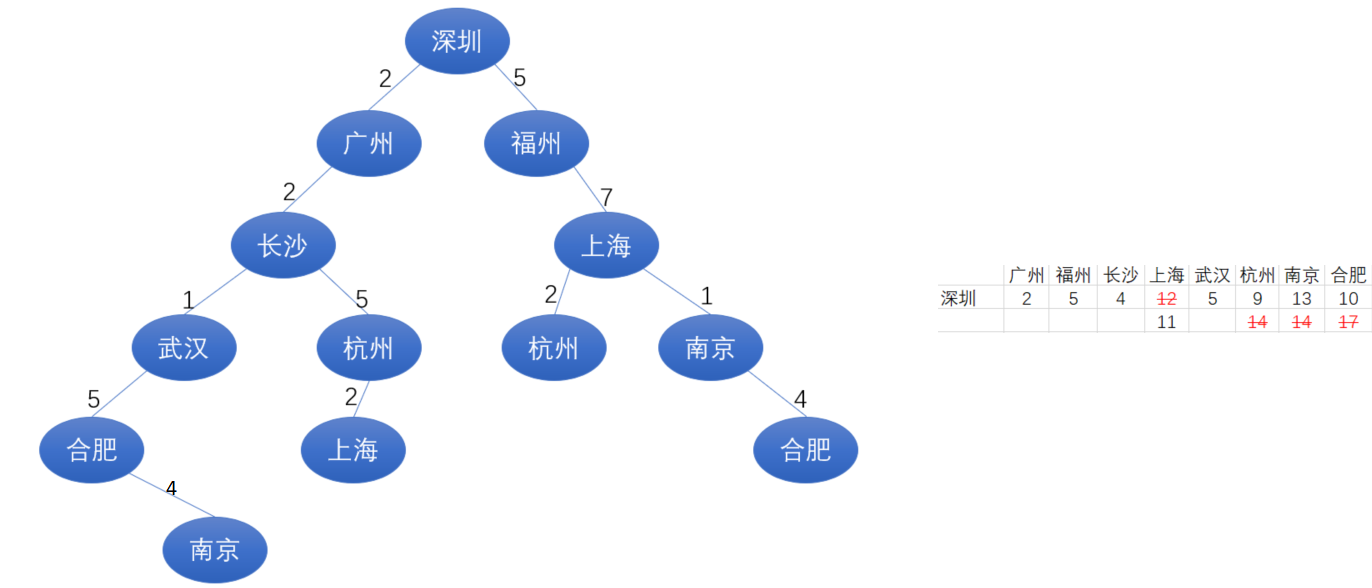


这一步的要点是杭州必须要在左右两个分支里都出现及被检查。“未访问节点”的定义是该节点有没有被作为父节点被检查过；如果只是作为叶子被检查过的话，该节点依然可能是“未访问节点”。这一步其实是检查长沙和上海的叶子，杭州并不在“未访问节点”集合里面，所以它既要作为长沙的叶子被检查，又要作为上海的叶子被检查。

1. 继续检查下一层叶子，得到深圳-合肥的两组时间，以及深圳-上海的一组新的时间。我们也是取最小的一组时间记录下来。



1. 继续检查下一层叶子，得到深圳-南京的另一组时间，同样取最小那组。



1. 接下来已经没有未访问的节点了，即每个节点都作为父节点被检查过了。所以第五步得出的结果就是问题的最终答案--深圳到所有节点的最短时间路径及其对应的最短时间。

上面的步骤里提到几个和树的数据结构有关的词，这里也简单说一下计算机里树的一些基本概念:

**树**是一种数据结构，它是由有限个节点组成一个具有层次关系的集合。把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。它具有以下的特点：

每个节点有零个或多个**子节点（叶子）**；没有**父节点**的节点称为**根节点**；每一个非根节点有且只有一个**父节点**；除了根节点外，每个子节点可以分为多个不相交的**子树**。

小技巧：迷宫其实也是一棵树哦。下次解复杂迷宫的时候不妨试试像下面这样先把它变成一棵树来看：





Dijkstra算法可以用一句话总结成：依次遍历所有节点的邻节点，并不断更新两点间最小开销距离（我们的例子是时间）的过程。

Dijkstra算法非常简单、优雅、直观。Dijkstra说他是在一个咖啡馆的阳台里喝咖啡的时候想出该算法的。“事实上，算法是在那三年后才正式发表。”他回忆说，“发表之前的过程也很有意思，其中一个原因是我设计该算法的时候并没有用到铅笔和纸 -- 不用铅笔和纸的时候你就要被迫去避免所有可以避免的复杂元素。最终这个算法也成为我最为有趣，也是最重要的成就。”

你知道吗，Dijkstra算法也是现在我们生活中必不可少的一类软件 -- 谷歌地图、高德地图等导航软件的算法基础。各种导航算法其实都是Dijkstra算法的变种。真的，使得各种路径计算问题变得可能的本质原理就这么简单易懂。

这就是计算机算法的魅力。算法是一种聪明的科学，一种逻辑推理的自然表示方式 – 数学归纳法。一个好的算法就像庖丁解牛一样，抽丝剥茧后轻松把一个复杂问题的核心灵魂肢解开来。问题的一堆乱七八糟的属性和关系变成一个个简单的重复关系，再采用一组简单的重复步骤生成无穷无尽的复杂状态。-- 事实上，只有真正的聪明才智才能如此看透现实的复杂性。

下面是广度优先算法和Dijkstra算法的python程序实现：

* Tree().build\_tree() 函数用广度优先算法计算路径
* Dij().dij() 函数用Dijkstra算法计算路径

from collections import deque  
#represent each city and the distance to their neighbours:  
cities = {  
 'harbin': [('shenyang', 4)],  
 'shenyang': [('harbin',4), ('dalian',3), ('qinhuangdao',5), ('beijing',4)],  
 'beijing': [('shenyang',4), ('tianjin',1), ('shijiazhuang',1)],  
 'shijiazhuang': [('beijing',1), ('taiyuan',3), ('jinan',5),('zhengzhou',2)],  
 'taiyuan': [('shijiazhuang',3)],  
 'tianjin': [('beijing',1), ('jinan',2)],  
 'qinhuangdao': [('shenyang',5)],  
 'qingdao': [('jinan',4)],  
 'dalian': [('shenyang',3)],  
 'jinan': [('shijiazhuang',5), ('tianjin',2), ('qingdao',4), ('xuzhou',1)],  
 'xuzhou': [('jinan',1), ('nanjing',1), ('zhengzhou',3)],  
 'zhengzhou': [('xuzhou',3), ('shijiazhuang',2), ('xian',4), ('wuhan',2)],  
 'xian': [('zhengzhou',4), ('baoji',2), ('jiangyou',6)],  
 'baoji': [('xian',2), ('lanzhou',3)],  
 'lanzhou': [('baoji',3), ('xining',3)],  
 'xining': [('lanzhou',3)],  
 'jiangyou': [('xian',6)],  
 'wuhan': [('zhengzhou',2), ('chongqing',4), ('hefei',5), ('changsha',1)],  
 'chongqing': [('wuhan',4), ('chengdu',3)],  
 'chengdu': [('dazhou',6), ('chongqing',3)],  
 'dazhou': [('chengdu',6)],  
 'hefei': [('wuhan',5), ('nanjing',4)],  
 'nanjing': [('xuzhou',1), ('hefei',4), ('shanghai',1)],  
 'shanghai': [('nanjing',1), ('hangzhou',2), ('fuzhou',7)],  
 'hangzhou': [('shanghai',2), ('changsha',5)],  
 'changsha': [('hangzhou',5), ('wuhan',1), ('guiyang',5), ('guangzhou',2)],  
 'guiyang': [('changsha',5), ('kunming',3)],  
 'kunming': [('guiyang',3)],  
 'guangzhou': [('changsha',2),('shenzhen',2)],  
 'fuzhou': [('shanghai',7), ('shenzhen',5)],  
 'shenzhen': [('fuzhou',5),('guangzhou',2)]  
}  
  
  
class Tree:  
 # destTree=None  
  
 *'''  
 return (whether has path, the path)  
 '''* def build\_tree(self, start, dest):  
 startingTree = {# build the root, who has no parent  
 'parent': None,  
 'root': start # ,  
 # 'leaves': []  
 }  
 pendingSearch = [  
 startingTree  
 ]  
 searched = [start] # the searched nodes, put the starting node in it now  
  
 #destTree = None  
 searchlist = deque(pendingSearch)# the pending search list, start from the root  
 while len(searchlist) > 0:  
 searching = searchlist.popleft()# remove the first element of the searching list  
 children = cities[searching['root']]# to find the leaves of the first element's for looping  
 for i in children:  
 if dest == i[0]: # found the destination, then return the tree to this node  
 newTree = {'parent': searching, 'root': i[0]}  
 # searching['leaves'].append(destTree)  
 return True, self.getPath(newTree)  
 elif i[0] not in searched: # if this leaf node has not been searched, add it as a leaf  
 newTree = {'parent': searching, 'root': i[0]}  
 # searching['leaves'].append(newTree)  
 searched.append(i[0])# do not search the same node  
 searchlist.append(newTree)  
  
 return False, None # return None if cannot find out the path  
  
 '''  
 this function return the path of a tree  
  
 '''  
 def getPath(self,node):  
 path=[]  
 while node != None:  
 path.insert(0,node['root'])  
 node = node['parent']  
 return path  
  
class Dij:  
 dist = {}# to store the shortest path's distance  
 path = {}# to store the shortest path  
 checked = {}# store checked nodes  
 checklist = None#store nodes to be checked  
  
 '''  
 start the dijkstra algorithm  
 return the (dist, path)  
 '''  
 def dij(self,start, dest):  
 self.dist = {(start, start): 0}  
 self.path = {(start, start): [start]}  
 self.checked = {start: 1}  
 self.checklist = deque(cities[start])  
  
 while self.checklist:  
 self.dd(self.checklist.popleft()[0], start)  
  
 return self.dist[(start, dest)], self.path[(start, dest)]  
  
 def dd(self,checkPoint,start):  
 children = cities[checkPoint] # get checkPoint's children  
 tempdist = 999999  
 tempTarget = None # to store the leaf target that has the shortest path to checkPoint  
 saved\_dist = {} # the distance that searched before, but need to re-check here  
 for i in children: # to loop leaves of checkPoint  
 checking = i[0] # a city of leaves  
 if checking in self.checked: # if the city already checked before, then try to see if any shorter path, else this city need to be check  
 saved\_dist[checking] = i[1] # the dist between checkPoint and the checking leaf  
 if (self.dist[(start, checking)] + i[1]) < tempdist:  
 tempdist = self.dist[(start, checking)] + i[1] # find the shorter dist  
 tempTarget = checking  
 elif i not in self.checklist: # if this leaf not in checklist then add it to be checked later   
 self.checklist.append(i)  
 self.dist[(start, checkPoint)] = tempdist # update the dist between start and check point  
 self.path[(start, checkPoint)] = self.path[(start, tempTarget)] + [checkPoint]  
 self.checked[checkPoint] = 1 # marked checkpoint as checked, pls note that it's not marking its leaves!  
 # recheck the checked points to see if any shorter path  
 del saved\_dist[tempTarget] # remove the one that in the path to checkPoint  
 for i in saved\_dist: # to find if a shorter path to leaves that already checked before  
 if self.dist[(start, checkPoint)] + saved\_dist[i] < self.dist[(start, i)]:  
 self.dist[(start, i)] = self.dist[(start, checkPoint)] + saved\_dist[i]  
 self.path[(start, i)] = self.path[(start, checkPoint)] + [i]