基于公交换乘的算法设计

常远 420230171)

1)(西南财经大学 数学学院 四川 成都 611130)

摘 要 城市中公交车站点的分布存在一定的规律，规划者追求从起始点到终点着乘车人所化时间最短、公交车能耗最小、公交车路线及换乘站点不交叉的最优路径。在本文中，只考虑公交车路线中的换乘问题，求一个人在任意两个站点的最少换乘次数，我们采取动态规划思想和贪心算法思想，首先建立一个二维矩阵，再通过Floyd算法和Dijkstra算法两种解法求出换乘次数的最小值。Dijkstra算法的时间复杂度相较Floyd算法更低，但其为单源最短路径，分析公交换乘问题不够全面。在实际应用中，需要根据具体情况选择合适的算法来解决公交车换乘次数最少问题。

关键词 动态规划；贪心算法；Floyd算法；Dijkstra算法

# 1问题介绍

## 1.1引言

我们大家都坐过公交车。一个城市中有很多辆公交车，并且有着不同线路，构成了方便市民的公交车线路网络，这些公交车的路线错综复杂，从一个起始点车站到目标点车站可能有数十条线路，有好多种换乘搭配，通常来说，因为部分公交车到站的间隔时间比较长，我们都会尽可能减少换乘的次数，达到减少时间的效果。

## 1.2问题描述

我们对该问题进行如下的建模与简化，在这里我们用数组routes，表示一系列公交路线，其中数组中的每一个列表都表示一辆公交车的路线，routes[i]表示第i辆公交车的路线。

例如，路线routes[0]=[1,3,6,9]表示第0辆公交车按照1 -> 3 -> 6 -> 9 -> 1 -> 3 -> 6 ->...的路线行驶。

现在我们假设starting为起始站，terminal为终点站，期间可以换乘公交车。试求解从starting到terminal最少乘坐的公交车数量。

为了使得题解更加直观可使，设计一个公交路线图，由图 1所示，为环形路线，共有6条公交线路，即有6个不同的换乘公交，有19个站点，其中站点4,8,12,17为重要的交点站，我们在执行算法时着重对交点站进行验证。

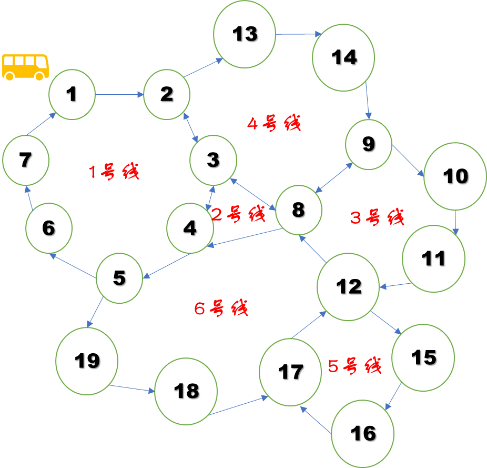


图1 公交路线图

# 2动态规划解法

## 2.1动态规划思路分析

在这里我们采取使用了动态规划思想的Floyd算法。

Floyd算法可以一次性计算出所有点之间相互的最短距离。

在该算法中，我们将公交车路线构造为一个二维矩阵graph[i][j]来表示顶点之间的最短距离。如果i和j之间有边，则graph[i][j]等于该边的权值；如果i和j之间没有边，则graph[i][j]等于inf（表示不可达）。

在算法迭代过程中，对于每对顶点i和j，考虑在路径上是否经过顶点k，将顶点集合{1,2,…,n}分为两部分，第一部分是除了k以外的所有顶点，第二部分是k顶点，按照此划分，可得状态转移方程为：

graph[i][j]=min(graph[i][k]+graph[k][j]),

如果graph[i][j]>graph[i][k]+graph[k][j]成立，则更新graph[i][j]的值，同时令P[i][j]=k，P[i][j]二维列表代表的是最少换乘车数的站点。

我们可以从图 2中看出。结点5是一个重要的中轴点，更新减少了到达结点12，17,19的换乘次数，而结点13会使得换乘次数增加，需舍去。

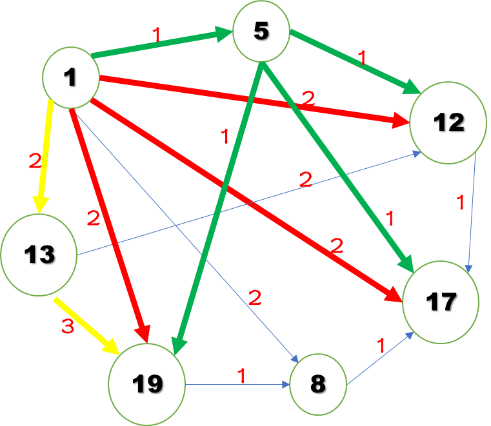


图2 动态规划思想图

## 2.2 动态规划伪代码

表1 动态规划思想伪代码

|  |
| --- |
| Floyd算法伪代码 |
| 输入：出发点starting，终点terminal，二维列表routes[i][j]，其中列表中的每一行代表一个公交车经过的站点路线。  输出:公交车从出发点starting到重点terminal的最少换乘公交的次数k。   1. 将routes的每个元素转为集合 2. rows 🡨 length(routes) 3. start 🡨 0 4. end 🡨 0 5. 建立一个二维数组 6. graph[v][u] 7. FOR i, row\_route do //定义路线row\_route 8. IF starting, terminalrow\_route 9. return 1 10. IF startingrow\_route 11. return start 🡨 i 12. IF terminalrow\_route 13. return end 🡨 i 14. graph[i][i] 🡨 0 15. FOR j 🡨 i+1 to rows do 16. IF 第i行的站点也出现在第j行的站点上 17. THEN graph[i][j]=graph[j][i]=1 18. FOR k 🡨 0 to rows do 19. FOR i 🡨 0 to rows do 20. FOR j 🡨 0 to rows do 21. Graph[i][j] 🡨 min(graph[i][j],   graph[i][k]+ graph[k][j])   1. IF graph[start][end] != inf 2. return graph[start][end] + 1 |

# 3贪心算法解法

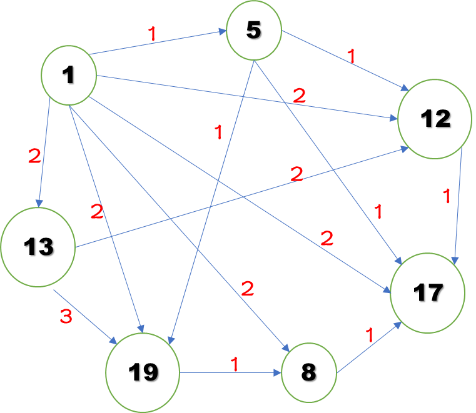
## 3.1贪心算法思路分析

在这里我们采取使用了贪心思想的Dijkstra算法。

我们假设路径和长度都已知，通过Dijkstra算法计算最短距离。Dijkstra算法只能求一个顶点到其他点的最短距离而不能任意两点。

在这个分析中我们认为换乘等待的时间远大于坐过站数的时间，没有考虑换乘车辆时站点中间需要坐过的站数，使得模型更容易分析。

在这里，我们选取具有代表性的几个站点作实例分析。如图 3所示。



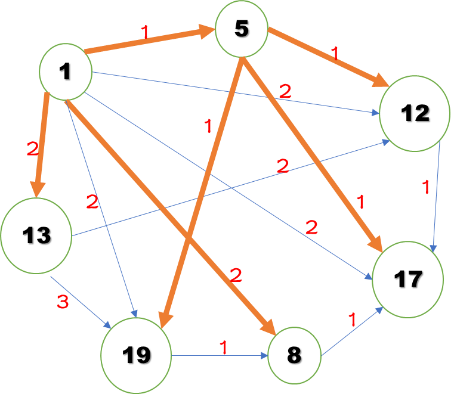


图3 贪心算法思想图

我们用到了贪心算法的思想。

S代表结点的集合，dist[i]表示到i点的距离长度，L[j]表示新加入的结点j相连的结点。

第一步，S={1},下面计算结点5,8,12,13,17,19相对于S的最短路径。

dist[5]=1,dist[13]=2,dist[8]=2,

dist[12]=dist[17]=dist[19]=0.

其中最短距离是1，于是结点5加入到S中，得L[5]=1。

第二步，S={1,5},修改距离dist如下：

dist[19]=min{1+1,**2**}=2,

dist[12]=min{1+1,**2**}=2,

dist[17]=min{1+1,**2**}=2.

其中三个点的最短距离都为2，于是结点12,17,19加入到S中，得L[12]= L[17]= L[19]=5。

……

最后一步，S={1,5,12,13,17,19}，不修改距离。把最后一个结点8加入到S中，L[8]=1,S={1,5,12,13,17,19,8},S=V,算法结束。得到

dst[1]=0, dist[5]=1, dist[8]=2, dist[12]=2, dist[13]=2, dist[17]=2, dist[19]=2.

最后的最短路径如图橘色粗线所示。

## 3.2 贪心算法伪代码

表2 贪心算法思想伪代码

|  |
| --- |
| Dijkstra算法伪代码 |
| 输入：出发点starting，终点terminal，二维列表routes[i][j]，其中列表中的每一行代表一个公交车经过的站点路线。  输出:公交车从出发点starting到重点terminal的最少换乘公交的次数k。   1. 将routes的每个元素转为集合 2. rows 🡨 length(routes) 3. start 🡨 0 4. end 🡨 0 5. 建立一个二维数组 6. graph[v][u] 7. FOR i, row\_route do //定义路线row\_route 8. IF starting, terminalrow\_route 9. return 1 10. IF startingrow\_route 11. return start 🡨 i 12. IF terminalrow\_route 13. return end 🡨 i 14. graph[i][i] 🡨 0 15. FOR j 🡨 i+1 to rows do 16. IF 第i行的站点也出现在第j行的站点上 17. THEN graph[i][j]=graph[j][i]=1 18. not\_nodes 🡨 list(元素为未成为start的点) 19. FOR k 🡨 0 to rows-1 do 20. min\_cost 🡨 inf 21. min\_node 🡨 -1 22. FOR not\_nodenot\_nodes do 23. IF graph[start][not\_node] < min\_cost 24. THEN min\_cost 🡨graph[start][not\_node] 25. THEN min\_node 🡨 not\_node 26. 遍历列表中的结点后，选择代价最小的节   点移除   1. FOR not\_node in not\_nodes do 2. Graph[start][not\_node] 🡨 min(   Graph[start][not\_node],Graph[start][min\_node]+ Graph[min\_node][not\_node])   1. IF graph[start][end] != inf 2. return graph[start][end] + 1 |

# 4分析与总结

## 4.1算法时间复杂度分析

在Floyd算法中，用map函数将一个列表转化，时间复杂度为O(mn)，其中n代表二维列表routes的长度，即公交车线路数量，m代表routes中最长的一个list的长度。之后，我们初始化了一个二维数组graph，时间复杂度为O(n^2)，其中n表示列表routes的长度。之后，我们用enumerate()函数按顺序返回一个二元组(i,row\_route)，其中i代表索引号，row\_route代表对应的集合其时间复杂度为O(nm)。之后在循环中使用if语句遍历列表routes的值，然后判断、负值，其时间复杂度为O(nm^2)。接下来是典型的Floyd算法的三层循环，时间复杂度为O(n^3)。因此，Floyd算法的时间复杂度为O(nm^2+n^3)。

在Dijkstra算法中，同样用enumerate()函数按顺序返回一个二元组(i,row\_route)，其时间复杂度为O(nm)。之后在循环中使用if语句遍历列表routes的值，然后判断、负值，其时间复杂度为O(nm^2)。然后是Dijkstra算法，循环共执行了n-1次，其中n代表的是列表routes的长度,即公交车数量，且每次循环中都会选出当前未访问过的节点中代价最小的点，其时间复杂度为O(n^2)。因此，Dijkstra算法的时间复杂度为O(nm^2+n^2)。

我们可以发现，在时间复杂度效果上，使用Dijkstra算法比Floyd算法略强。

## 4.2 总结

在本文中，我们只针对换乘次数进行分析，没有考虑在一辆公交车上经过的站点，对于现实生活中的公交车换乘需要将两者花费时间结合来具体分析。

我们对公交路线采用两种算法思想进行分析，分别是动态规划思想和贪心算法思想，其中，我们具体使用动态规划思想的Floyd算法和贪心算法思想的Dijkstra算法来解决问题。

Floyd算法和Dijkstra算法对于该公交车换乘次数问题都能很好解决，且时间复杂度不超过x^3,代码效率较高。Floyd算法是多源最短路径算法，可以解决任意两个两点之间的最短路问题，而Dijkstra算法是单源最短路径解法，所以Floyd算法相比其更为全面。然而，如果城市规模较大，公交车的路线更加繁杂，导致二维列表矩阵graph中的元素非常多，这会提高算法的时间复杂度，造成计算瓶颈。在本文的代码的时间复杂度分析中，我们得知Dijkstra算法略强于Floyd算法，同时Dijkstra算法还可以使用堆优化进一步降低时间复杂度到O(m\*log n)。因此，在实际应用中，需根据实际情况选择合适的算法来解决公交车换乘次数最少问题。

致 谢 感谢施龙老师在我的论文完成过程中的细心指导和支持。从选题阶段到构思、撰写和修改，给予了宝贵的建议和专业知识。我衷心感谢施龙老师所付出的时间和精力，谨向他表达最诚挚的谢意。

参 考 文 献

[1] 屈婉玲、刘田、张立昂、王捍贫．算法设计与分析（第2版）：清华大学出版社，2016

附录：

Python代码：

#Floyd算法

def numBusesToDestination(self, routes: List[List[int]], S: int, T: int) -> int:

if not routes:

return -1

if S == T:

return 0

routes = list(map(set, routes))

rows = len(routes)

start = 0

end = 0

graph = [[float('inf') for \_ in range(rows)] for \_ in range(rows)]

for i, row\_route in enumerate(routes):

if S in row\_route and T in row\_route:

return 1

if S in row\_route:

start = i

if T in row\_route:

end = i

graph[i][i] = 0

for j in range(i + 1, rows):

if any([route in routes[j] for route in row\_route]):

graph[i][j] = 1

graph[j][i] = 1

for k in range(rows):

for i in range(rows):

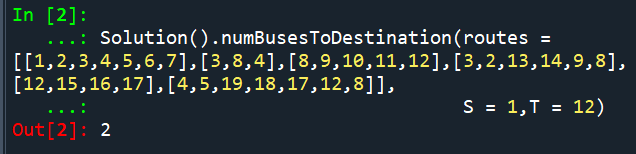
for j in range(rows):

graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j])

return -1 if graph[start][end] == float('inf') else graph[start][end] + 1

Floyd算法输入实例：

Solution().numBusesToDestination(routes = [[1,2,3,4,5,6,7],[3,8,4],[8,9,10,11,12],[3,2,13,14,9,8],[12,15,16,17],[4,5,19,18,17,12,8]],S = 1,T = 12)



#Dijkstra算法

def numBusesToDestination(self, routes: List[List[int]], S: int,T: int) -> int:

if not routes:

return -1

if S == T:

return 0

rows = len(routes)

start = 0

end = 0

graph = [[float('inf') for \_ in range(rows)] for \_ in range(rows)]

for i, row\_route in enumerate(routes):

if S in row\_route and T in row\_route:

return 1

if S in row\_route:

start = i

if T in row\_route:

end = i

graph[i][i] = 0

for j in range(i + 1, rows):

if any([route in routes[j] for route in row\_route]):

graph[i][j] = 1

graph[j][i] = 1

not\_visited\_nodes = [i for i in range(rows) if i != start]

for \_ in range(rows - 1):

min\_cost = float('inf')

min\_node = -1

for not\_visited\_node in not\_visited\_nodes:

if graph[start][not\_visited\_node] <= min\_cost:

min\_cost = graph[start][not\_visited\_node]

min\_node = not\_visited\_node

not\_visited\_nodes.remove(min\_node)

for not\_visited\_node in not\_visited\_nodes:

graph[start][not\_visited\_node] = min(graph[start][not\_visited\_node], graph[start][min\_node]+graph[min\_node][not\_visited\_node])

return -1 if graph[start][end] == float('inf') else graph[start][end] + 1

Dijkstra算法输入实例：

Solution().numBusesToDestination(routes = [[1,2,3,4,5,6,7],[3,8,4],[8,9,10,11,12],[3,2,13,14,9,8],[12,15,16,17],[4,5,19,18,17,12,8]],S = 1,T = 12)

