算法分析第三次作业

石发强 ZY1806707

14061115@buaa.edu.cn

北京航空航天大学 计算机学院

2019 年 1 月 8 日

* 题目

用分支定界算法求以下问题：

某公司于乙城市的销售点急需一批成品，该公司成品生产基地在甲城市。甲城市与乙城市之间共有 n 座城市，互相以公路连通。甲城市、乙城市以及其它各城市之间的公路连通情况及每段公路的长度由矩阵 M1 给出。每段公路均由地方政府收取不同额度的养路费等费用，具体数额由矩阵 M2 给出。请给出在需付养路费总额不超过 1500 的情况下，该公司货车运送其产品从甲城市到乙城市的最短运送路线。

具体数据参见文件：

* M1.txt

各城市之间的公路连通情况及每段公路的长度矩阵 (有向图)，甲城市为城市 Num.1，乙城市为城市 Num.50

* M2.txt

每段公路收取的费用矩阵（非对称）

* 算法分析

**2.1** 算法流程

1. 读取所有城市间的距离矩阵 m1 和和代价矩阵 m2
2. 采用 Floyd 算法计算所有城市对之间的最短路径长度和最小代价，用于计算步骤 4中当前代价与当前距离
3. 初始化一个栈，将 0 节点、即出发点甲城市压栈

1

1. 取出栈顶节点，检查其所有相邻节点，确定下一个当前最优路径上的节点压栈；在检查的过程中如果发现超出最短路径或者代价限制 1500，则进行剪枝 (分支限界优化的核心步骤)，然后弹栈回溯
2. 找到一个解后，保存该解，然后重复步骤 4，直到栈空，即可获得最优解

**2.2** 核心剪枝策略

在算法流程步骤 4中剪枝的核心策略是当前路径超出当前最优路径或者当前代价超出当前最优代价，即

*cur*\_*cost* + *mincost*[*cur*][*n* 1] *> cost*\_*bound or cur*\_*distance* + *mindist*[*cur*] *> distacne*\_*bound*

时则进行剪枝回溯，其中 *n* 为所有城市的个数。

**2.3** 算法复杂度

该实现算法中 Floyd 算法的时间复杂度为 *O*(*n*3)，而后进行 DFS 深度优先搜索的时间复杂度

* *O*(*n* + *e*)，但是因为采取了分支限界的办法，所以实际的计算量会远小于 *O*(*n* + *e*)，因此总的

时间复杂度为*O*(*n*3)，其中 *n* 是所有城市的个数、*e* 所有城市间互相联通的有向图边的个数。

因为采用邻接矩阵的方式存储所有城市间的距离和联通性且没有其它空间开销，因此空间复杂度为*O*(*n*2)，其中 *n* 是所有城市的个数。

* 计算结果

根据本题的具体数据，计算结果为从城市甲到城市乙：

* 最短路径长度 464
* 养路费花费 1448
* 最短路径 1->3->8->11->15->21->23->26->32->37->39->45->47->50，需经过 14 个城市
* Python 实现耗时约 0.9s,cpp 实现耗时在毫米级内。
* 具体实现

**4.1 Python** 实现

代码文件见附件 assignment\_2.py

2

**4.1.1 Python** 代码

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

#!/usr/bin/env python

* -\*- coding:utf-8 -\*-

'''

\* @Author: shifaqiang(石发强)--[14061115@buaa.edu.cn]

\* @Date: 2019-01-07 16:10:00

\* @Last Modified by: shifaqiang

\* @Last Modified time: 2019-01-07 16:10:00

\* @Desc: python implementation for algorithm analysis assignment\_3

'''

import numpy as np

import time

def floyd(graph):

'''

from wikipad (https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall\_algorithm), floyd algorithm is an method for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles).

A single execution of the algorithm will find the lengths (summed weights) of shortest paths between all pairs of vertices.

1 let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)

2 for each edge (u,v)

* dist[u][v] ← w(u,v) // the weight of the edge (u,v) 4 for each vertex v

5 dist[v][v] ← 0

6 for k from 1 to |V|

7 for i from 1 to |V|

8for j from 1 to |V|

9if dist[i][j] > dist[i][k] + dist[k][j]

10dist[i][j] ← dist[i][k] + dist[k][j]

1. end if

'''

assert isinstance(graph, np.ndarray)

# define 9999 as infinity n = graph.shape[0]

for k in range(n): for i in range(n):

3

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

for j in range(n):

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

return graph

def dfs(distance, cost, mindist, mincost):

"""

dfs with branch and bound

"""

# check if parameters are legal

assert isinstance(distance, np.ndarray) and isinstance(mindist, np.ndarray) assert isinstance(cost, np.ndarray) and isinstance(mincost, np.ndarray) assert distance.shape == cost.shape and distance.shape == mindist.shape assert cost.shape == mincost.shape

# set some variable

n = distance.shape[0]

res = None

# depth is the top of stack, cur is the current city, cur\_next is the next feasible city of current city, visited is flag for all cities if they are visited

stack, depth, cur, cur\_next, visited = [0]\*(n+2), 0, 0, 0, [False]\*n

* initialization of the dfs stack with start point city 0 visited[0] = True
* seting the bound

cost\_bound, distacne\_bound, cur\_distance, cur\_cost = 1500, np.inf, 0, 0 while depth >= 0:

found, cur, cur\_next = -1, stack[depth], stack[depth+1] for i in range(cur\_next+1, n):

* attempt all neighbor cities for city cur if distance[cur][i] == 9999 or visited[i]:

continue

elif cur\_cost+mincost[cur][n-1] > cost\_bound or cur\_distance+mindist[cur][n-1] > distacne\_bound: continue # prune operation

elif i < n: found = i

break # find a new feasible unvisited city, break the for loop

if found == -1:

* no feasible next city for cur city, backtracking depth -= 1

cur\_distance -= distance[stack[depth]][stack[depth+1]]

4

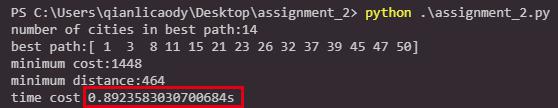
|  |  |
| --- | --- |
| 76 | cur\_cost -= cost[stack[depth]][stack[depth+1]] |
| 77 | visited[stack[depth+1]] = False |
| 78 | else: |
| 79 | # found a feasible next neighbor city for current city, update current |
|  | path with cost, distance, path stack and visited record |
| 80 | cur\_cost += cost[stack[depth]][found] |
| 81 | cur\_distance += distance[stack[depth]][found] |
| 82 | depth += 1 |
| 83 | stack[depth], stack[depth+1], visited[found] = found, 0, True |
| 84 | if found == n-1: |
| 85 | # arrive at terminal city, found a new feasible solution |
| 86 | if cur\_cost > 1500: |
| 87 | continue |
| 88 | res = stack[:depth+1], cur\_cost, cur\_distance |
| 89 | # update bound |
| 90 | distacne\_bound = cur\_distance |
| 91 | # backtracking |
| 92 | for i in range(2): |
| 93 | depth -= 1 |
| 94 | cur\_distance -= distance[stack[depth]][stack[depth+1]] |
| 95 | cur\_cost -= cost[stack[depth]][stack[depth+1]] |
| 96 | visited[stack[depth+1]] = False |
| 97 | return res |
| 98 |  |

1. if \_\_name\_\_ == "\_\_main\_\_":
2. # load data
3. start\_time = time.time()
4. distance = np.genfromtxt("m1.txt", skip\_header=False, delimiter='\t', dtype=np.int32)
5. cost = np.genfromtxt("m2.txt", skip\_header=False, delimiter='\t', dtype=np.int32)
6. assert distance.shape == cost.shape
7. # compute result
8. mindist = floyd(np.copy(distance))
9. mincost = floyd(np.copy(cost))
10. res = dfs(distance, cost, mindist, mincost)
11. print("number of cities in best path:{}\nbest path:{}\nminimum cost:{}\nminimum distance:{}".format(len(res[0]), np.array(res[0])+1, res[1], res[2]))
12. print("time cost:{}s".format(time.time()-start\_time))

5

**4.1.2 Python** 运行结果

经过分支限界优化，assignment\_2.py 在 1s 内计算出了正确结果1，如图 1所示。



* 1: Python 实现运行效果

1

2

3

4

5

6

7

8

9

**4.2 cpp** 实现

**4.2.1 cpp** 实现代码

代码文件见附件 assignment\_2.cpp

#include <stdio.h>

#include <time.h>

#include <iostream>

#include <fstream>

#include <regex>

#include <string>

#include <vector>

#include <limits>

10

11

12

13

14

15

16

17

18

19

using namespace std;

struct record

{

int length; // number of cities

vector<int> city;

int minimum\_cost;

int minimum\_dist;

};

#define size\_t int

1. void print(vector<vector<int>> &matrix)
2. {
3. // auxiliary output function
4. for (auto i : matrix)

1评测所用机器配置为 2 \* Intel(R) Xeon(R) E5-2620 v3 @ 2.40GHz CPU

6

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

{

for (auto j : i)

cout << j << " ";

cout << endl;

}

}

void load\_matrix(string filename, vector<vector<int>> &matrix)

{

* load a 2-dimensions array to matrix from file filename vector<int> tmp;

string line;

ifstream in(filename); //open file filename to ifstream

regex pat\_regex("[[:digit:]]+"); //match a type int number

while (getline(in, line))

{

for (sregex\_iterator it(line.begin(), line.end(), pat\_regex), end\_it; it != end\_it; ++it)

{

tmp.push\_back(stoi(it->str()));

}

matrix.push\_back(tmp);

tmp.clear();

}

}

void floyd(vector<vector<int>> &graph)

{

int count = graph.size();

for (size\_t k = 0; k < count; k++)

for (size\_t i = 0; i < count; i++)

for (size\_t j = 0; j < count; j++)

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

}

record branch\_and\_bound(vector<vector<int>> &dist, vector<vector<int>> &cost, vector<vector<int>> &mindist, vector<vector<int>> &mincost)

{

/\*

depth is the top of stack, cur is the current city, cur\_next is the next feasible city of current city, visited is flag for all cities if they are visited

\*/

7

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

int n = dist.size(), depth = 0;

record res;

int cost\_bound = 1500, distance\_bound = INT32\_MAX, cur\_distance = 0, cur\_cost = 0;

int \*stack = new int[n + 2];

bool \*visited = new bool[n];

memset(stack, 0, sizeof(int) \* (n + 2));

memset(visited, false, sizeof(bool) \* n);

visited[0] = true; //push first city 0 to stack and visit it

while (true)

{

int found = -1, cur = stack[depth], cur\_next = stack[depth + 1]; for (size\_t i = cur\_next + 1; i < n; i++) {

/\* attempt all neighbor cities for city cur \*/ if ((dist[cur][i] == 9999) || visited[i])

continue;

else if ((cur\_cost + mincost[cur][n - 1] > cost\_bound) || (cur\_distance + mindist[cur][n - 1] > distance\_bound))

continue; // prune operation

else if (i < n)

{

found = i; // find a new feasible unvisited city i, break the for loop

* cout << "found city " << i << " for cur city " << cur << endl; break;

}

}

if (found == -1)

{

* no feasible next city for cur city, backtracking depth--;

if (depth < 0) break;

cur\_distance -= dist[stack[depth]][stack[depth + 1]]; cur\_cost -= cost[stack[depth]][stack[depth + 1]]; visited[stack[depth + 1]] = false;

}

else

{

// found a feasible next neighbor city for current city, update current

8

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

130

131

132

133

134

135

136

path with cost, distance, path stack and visited record cur\_cost += cost[stack[depth]][found];

cur\_distance += dist[stack[depth]][found];

depth++;

stack[depth] = found, stack[depth + 1] = 0, visited[found] = true;

if (found == n - 1)

{

* arrive at terminal city, found a new feasible solution if (cur\_cost > 1500)

continue;

* update bound and record current best result

distance\_bound = cur\_distance;

res.length = depth + 1;

res.city.clear();

for (size\_t i = 0; i < res.length; i++)

res.city.push\_back(stack[i] + 1);

res.minimum\_cost = cur\_cost;

res.minimum\_dist = cur\_distance;

* backtracking for other feasible solution for (size\_t i = 0; i < 2; i++)

{

depth--;

cur\_distance -= dist[stack[depth]][stack[depth + 1]]; cur\_cost -= cost[stack[depth]][stack[depth + 1]]; visited[stack[depth + 1]] = false;

}

}

}

}

return res;

}

int main(int argc, char const \*argv[])

{

time\_t start = clock();

// load data

string dist\_filename = "m1.txt", cost\_filename = "m2.txt"; vector<vector<int>> dist, cost, mindist, mincost; load\_matrix(dist\_filename, dist); mindist = dist;

floyd(mindist);

9

1. load\_matrix(cost\_filename, cost);
2. mincost = cost;
3. floyd(mincost);
4. record res = branch\_and\_bound(dist, cost, mindist, mincost);
5. cout << "number of cities in best path:" << res.length << "\nbest path:";
6. for (size\_t i = 0; i < res.length; i++)
7. cout << res.city[i] << " ";
8. cout << "\nminimum cost:" << res.minimum\_cost << "\nminimum distance:" << res.minimum\_dist << endl;
9. cout << "time cost:" << (clock() - start) / double(CLOCKS\_PER\_SEC) << "s" << endl;
10. return 0;
11. }

**4.2.2 cpp** 运行结果

经过分支限界优化，assignment\_2.exe 在毫秒级时间内计算出正确结果，如图 2所示。

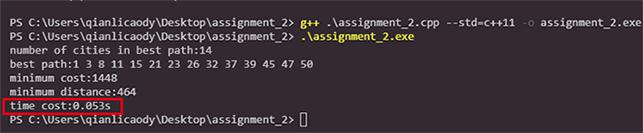


图 2: cpp 实现运行效果

* 附件
  1. assignment\_2.py
  2. assignment\_2.cpp
  3. assignment\_2.exe2

2运行代码时请将 Python 代码文件或可执行文件与参数文件 m1.txt、m2.txt 置于同一目录下

10