**Министерство образования и науки Украины**

**Национальный технический университет Украины "Киевский политехнический институт имени Игоря Сикорского"**

**Факультет информатики и вычислительной техники**

**Кафедра автоматизированных систем обработки**

**информации и управления**

**ОТЧЕТ**

по лабораторной работе № 4 по дисциплине

«Проектирование и анализ вычислительных алгоритмов»

„ **Эвристические алгоритмы** ”

**Выполнил**

(шифр, фамилия, имя, отчество)

*ІП-61, Кушка М.О.*

**Проверил**

(фамилия, имя, отчество )

*Головченко М.Н.*

Киев 2018

Содержание

[1 Цель лабораторной работы 3](#_Toc514656076)

[2 Задание 4](#_Toc514656077)

[3 Выполнение 6](#_Toc514656078)

[3.1 Псевдокод алгоритмов 6](#_Toc514656079)

[3.2 Входные данные задачи 8](#_Toc514656080)

[3.3 Программная реализация 9](#_Toc514656081)

[3.3.1 Исходный код 9](#_Toc514656082)

[3.3.2 Примеры работы 16](#_Toc514656083)

[Выводы 18](#_Toc514656084)

# Цель лабораторной работы

Цель работы – изучить основные подходы к формализации эвристических алгоритмов и решению типовых задач с их помощью.

# Задание

Выбрать 15 городов в стране согласно варианту (таблица 2.1) и записать для них кратчайшее расстояние по дороге, в случае прямого сообщения между ними и расстояние по прямой в отдельные таблицы. Для определения расстояний рекомендуется использовать интернет сервисы (например Google Maps).

Записать алгоритм методов Жадный поиск и поиск А\* для задачи нахождения кратчайшего пути между парами вершин в транспортной сети (неориентированном графе)

Разработать программу, которая будет находить кратчайшие маршруты между каждой парой городов. В качестве методов нахождения маршрутов выбрать Жадный поиск и поиск А\*. В качестве эвристики выбрать расстояние по прямой.

Ответ выводить в виде (Город1-Город2 Расстояние: 234км Маршрут: Город1 → Город3 → Город4 → Город2).

Сделать обобщенный вывод по лабораторной работе, в котором оценить качество алгоритмов.

Таблица 2.1 – Варианты

|  |  |
| --- | --- |
| **№** | **Вариант** |
| 1 | Индонезия |
| 2 | Австралия |
| 3 | Австрия |
| 4 | Азербайджан |
| 5 | Испания |
| 6 | Албания |
| 7 | Алжир |
| 8 | Италия |
| 9 | Ангола |
| 10 | ОАЭ |
| 11 | Аргентина |
| 12 | Армения |
| 13 | Мексика |
| 14 | Афганистан |
| 15 | Молдавия |
| 16 | Бангладеш |
| 17 | Барбадос |
| 18 | Польша |
| 19 | Беларусь |
| 20 | Португалия |
| 21 | Бельгия |
| 22 | Сербия |
| 23 | Болгария |
| 24 | Словакия |
| 25 | Норвегия |
| 26 | Нидерланды |
| 27 | Перу |
| 28 | Сингапур |
| 29 | Таиланд |
| 30 | Турция |

# Выполнение

## Псевдокод алгоритмов

Алгоритм Дейкстры

**function** Dijkstra(*Graph*, *source*) **is**

create vertex set Q

**for each** vertex *v* in *Graph*: *// Initialization*

dist[*v*] ← INFINITY

prev[*v*] ← UNDEFINED

add *v* to *Q*

dist[*source*] ← 0

**while** *Q* is not empty:

*u* ← vertex in *Q* with min dist[u] *// Node with the least distance*

*// will be selected first*

remove *u* from *Q*

**for each** neighbor *v* of *u*: *// where v is still in Q.*

*alt* ← dist[*u*] + length(*u*, *v*)

**if** *alt* < dist[*v*]:

dist[*v*] ← *alt*

prev[*v*] ← *u*

**return** dist[], prev[]

Алгоритм A\*

**function** A\*(start, goal)

*// The set of nodes already evaluated*

closedSet := *{}*

*// The set of currently discovered nodes that are not evaluated yet.*

*// Initially, only the start node is known.*

openSet := *{start}*

*// For each node, which node it can most efficiently be reached from.*

*// If a node can be reached from many nodes, cameFrom will eventually contain the*

*// most efficient previous step.*

cameFrom := an empty map

*// For each node, the cost of getting from the start node to that node.*

gScore := map **with** default value **of** Infinity

*// The cost of going from start to start is zero.*

gScore[start] := 0

*// For each node, the total cost of getting from the start node to the goal*

*// by passing by that node. That value is partly known, partly heuristic.*

fScore := map **with** default value **of** Infinity

*// For the first node, that value is completely heuristic.*

fScore[start] := heuristic\_cost\_estimate(start, goal)

**while** openSet **is** **not** empty

current := the node **in** openSet having the lowest fScore[] value

**if** current = goal

return reconstruct\_path(cameFrom, current)

openSet.Remove(current)

closedSet.Add(current)

**for** each neighbor **of** current

**if** neighbor **in** closedSet

**continue** *// Ignore the neighbor which is already evaluated.*

**if** neighbor **not** **in** openSet *// Discover a new node*

openSet.Add(neighbor)

*// The distance from start to a neighbor*

*//the "dist\_between" function may vary as per the solution requirements.*

tentative\_gScore := gScore[current] + dist\_between(current, neighbor)

**if** tentative\_gScore >= gScore[neighbor]

**continue** *// This is not a better path.*

*// This path is the best until now. Record it!*

cameFrom[neighbor] := current

gScore[neighbor] := tentative\_gScore

fScore[neighbor] := gScore[neighbor] + heuristic\_cost\_estimate(neighbor, goal)

return failure

## Входные данные задачи

В таблице 3.1 приведены расстояния между городами по дороге, если между ними есть прямое сообщение.

Таблица 3.1 – Расстояния между городами по дороге

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 19 | 11 | 12 | 13 | 14 | 15 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 69 | 74 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 157 | 0 | 0 | 0 | 0 | 142 | 81 | 104 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 126 | 135 | 0 | 0 | 130 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 157 | 0 | 0 | 150 | 0 | 0 | 0 | 74 | 0 | 0 | 110 | 0 | 0 | 103 |
| 5 | 0 | 0 | 126 | 150 | 0 | 0 | 0 | 0 | 174 | 0 | 0 | 0 | 0 | 0 | 137 |
| 6 | 0 | 0 | 135 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 68 | 0 | 0 | 0 | 0 |
| 7 | 69 | 0 | 0 | 0 | 0 | 0 | 0 | 99 | 0 | 0 | 0 | 193 | 115 | 0 | 0 |
| 8 | 74 | 0 | 0 | 0 | 0 | 0 | 99 | 0 | 0 | 69 | 0 | 0 | 0 | 76 | 0 |
| 9 | 0 | 142 | 130 | 74 | 174 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 0 | 81 | 0 | 0 | 0 | 0 | 0 | 69 | 0 | 0 | 0 | 0 | 80 | 85 | 0 |
| 11 | 0 | 104 | 0 | 0 | 0 | 68 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 110 | 0 | 0 | 193 | 0 | 0 | 0 | 0 | 0 | 90 | 0 | 156 |
| 13 | 0 | 0 | 0 | 0 | 0 | 0 | 115 | 0 | 0 | 80 | 0 | 90 | 0 | 0 | 0 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 76 | 0 | 85 | 0 | 0 | 0 | 0 | 0 |
| 15 | 0 | 0 | 0 | 103 | 137 | 0 | 0 | 0 | 0 | 0 | 0 | 156 | 0 | 0 | 0 |

В таблице 3.2 приведены расстояния между городами по прямой.

Города:

1. Boston
2. Chicago
3. Denver
4. Duluth
5. Helena
6. Kansas City
7. Montreal
8. New York
9. Omaha
10. Pittsburg
11. Saint Louis
12. Ste Marie
13. Toronto
14. Washington
15. Winnipeg

Таблица 3.2 – Расстояния между городами по прямой

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 240 | 107 | 270 | 110 | 254 | 176 | 193 | 195 | 150 | 152 | 180 | 0 | 90 | 238 | 156 |

## Программная реализация

### Исходный код

import math

class Dijkstra:

"""Graph traversal using Dijkstra's algorithm"""

def \_\_init\_\_(self, startNode, filename='matrix.txt'):

"""Init necessary params"""

# Get adjacency matrix from file

self.matrix = self.\_\_getMatrixFromFile(filename)

# Number of nodes in the graph

self.n = len(self.matrix)

# Start node to find path from it to all other nodes

self.startNode = startNode

# Mark all nodes as unvisited

self.unvisited = [True for \_ in range(self.n)]

# Mark distances to all the nodes except first one as infinity

self.distances = [math.inf for \_ in range(self.n)]

self.distances[self.startNode] = 0

def showAdjacencyMatrix(self):

"""Display adjacency matrix on the screen"""

print('\n======================')

print('== Adjacency matrix ==')

print('======================')

for row in self.matrix:

for val in row:

print('{:3}'.format(val), end='')

print()

print()

def process(self):

"""Traverse the graph using Dijkstra's algorithm"""

q = [i for i in range(self.n)] # vertexes

previous = [[] for \_ in range(self.n)]

while q != []:

# Node with the least distance

minVertex = 0

minValue = math.inf

for vertex in q:

if self.distances[vertex] < minValue:

minVertex = vertex

minValue = self.distances[vertex]

u = minVertex

q.remove(u)

for v in self.\_\_getNeighbors(u):

alt = self.distances[u] + self.matrix[u][v]

# A shortest path to v has been found

if alt < self.distances[v]:

self.distances[v] = alt

previous[v] = u

return self.distances, previous

def \_\_getNeighbors(self, node):

"""Get list of neighbors for the current node"""

lst = []

for i in range(self.n):

if self.matrix[node][i] != 0:

lst.append(i)

return lst

def \_\_getMatrixFromFile(self, filename):

"""Read adjacency matrix from file"""

with open(filename, 'r') as f:

lines = f.readlines()

matrix = []

matrixInt = []

for line in lines:

matrix.append(line.replace('\n', '').split(' '))

# Convert string values to int

for row in matrix:

matrixInt.append([int(val) for val in row])

return matrixInt

class Astar:

"""Graph traversal using A\* algorithm"""

def \_\_init\_\_(self, start, filename='A-star-matrix.txt',

heuristicFilename='A-star-heuristic.txt'):

"""Init necessary params"""

# Get adjacency matrix from file

self.matrix = self.\_\_getMatrixFromFile(filename)

self.heuristicFilename = heuristicFilename

# Number of nodes in the graph

self.n = len(self.matrix)

# Start node to find path from it to all other nodes

self.start = start

self.visited = []

self.previousNodes = [[] for \_ in range(self.n)]

self.totalDistances = [math.inf for \_ in range(self.n)]

self.shortestDistancesFromStart = [math.inf for \_ in range(self.n)]

self.shortestDistancesFromStart[start] = 0

def showAdjacencyMatrix(self):

"""Display adjacency matrix on the screen"""

print('\n======================')

print('== Adjacency matrix ==')

print('======================')

for row in self.matrix:

for val in row:

print('{:4}'.format(val), end='')

print()

print()

def process(self):

"""Traverse the graph using A\* algorithm"""

current = self.start

self.totalDistances[current] = \

self.shortestDistancesFromStart[current] + \

self.heuristic(current)

# Loop

while True:

unvisitedNeighbors = self.\_\_getUnvisitedNeighbors(current)

for neighbor in unvisitedNeighbors:

if (self.shortestDistancesFromStart[current] +

self.matrix[current][neighbor] <

self.shortestDistancesFromStart[neighbor]):

self.shortestDistancesFromStart[neighbor] = \

self.shortestDistancesFromStart[current] + \

self.matrix[current][neighbor]

self.totalDistances[neighbor] = \

self.shortestDistancesFromStart[neighbor] + \

self.heuristic(neighbor)

self.previousNodes[neighbor] = current

self.visited.append(current)

# Find unvisited node with the shortest distance from start

smallestUnvisitedNode = math.inf

smallestDistance = math.inf

for node in range(self.n):

if self.totalDistances[node] < smallestDistance and \

node not in self.visited:

smallestUnvisitedNode = node

smallestDistance = self.totalDistances[node]

# All nodes are visited

if len(self.visited) == self.n:

break

current = smallestUnvisitedNode

return [self.shortestDistancesFromStart, self.previousNodes]

def heuristic(self, node):

# Only to Sault Ste Marie node

h = self.\_\_getMatrixFromFile(self.heuristicFilename)

return h[0][node]

def \_\_getUnvisitedNeighbors(self, node):

"""Get list of neighbors for the current node"""

lst = []

for i in range(self.n):

if self.matrix[node][i] != 0 and (i not in self.visited):

lst.append(i)

return lst

def \_\_getMatrixFromFile(self, filename):

"""Read adjacency matrix from file"""

with open(filename, 'r') as f:

lines = f.readlines()

matrix = []

matrixInt = []

for line in lines:

matrix.append(line.replace('\n', '').split(' '))

# Convert string values to int

for row in matrix:

matrixInt.append([int(val) for val in row])

return matrixInt

def formatPathways(start, pathways, cities):

"""Get pathways, formated as a string"""

order = []

n = len(pathways)

# Get nodes order

for node in range(n):

if pathways[node] == []:

order.append([])

continue

temp = [node]

current = pathways[node]

while current != start:

temp.insert(0, current)

current = pathways[current]

temp.insert(0, start)

order.append(temp)

# Get path strings from orders

iterator = 0

output = []

for path in order:

current = path[:]

if path == []:

output.append("<empty>")

else:

output.append(" -> ".join([cities[val] for val in current]))

iterator += 1

return output

def main():

start = 11 # Sault Ste Marie

cities = ['Boston', 'Chicago', 'Denver', 'Duluth', 'Helena', 'Kansas City',

'Montreal', 'New York', 'Omaha', 'Pittsburg', 'Saint Louis',

'Sault Ste Marie', 'Toronto', 'Washington', 'Winnipeg']

dijkstra = Dijkstra(start, filename='USA-matrix.txt')

astar = Astar(start, filename='USA-matrix.txt',

heuristicFilename='USA-heuristic.txt')

d\_costs, d\_pathways = dijkstra.process()

a\_costs, a\_pathways = astar.process()

d\_pathwaysStr = formatPathways(start, d\_pathways, cities)

a\_pathwaysStr = formatPathways(start, a\_pathways, cities)

# Beuatiful output

# Dijkstra's algorithm

print(

"""\n

==========================

== Dijkstra's algorithm ==

==========================

"""

)

iteration = 0

for cost, path in zip(d\_costs, d\_pathwaysStr):

print("From " + cities[start] + " to " + cities[iteration] + ":")

print("\tCost: " + str(cost) + 'km')

print("\tPath:", path)

iteration += 1

# A\* algorithm

print(

"""\n

==========================

====== A\* algorithm ======

==========================

"""

)

iteration = 0

for cost, path in zip(d\_costs, d\_pathwaysStr):

print("From " + cities[start] + " to " + cities[iteration] + ":")

print("\tCost: " + str(cost) + 'km')

print("\tPath:", path)

iteration += 1

if \_\_name\_\_ == "\_\_main\_\_":

main()

### Примеры работы

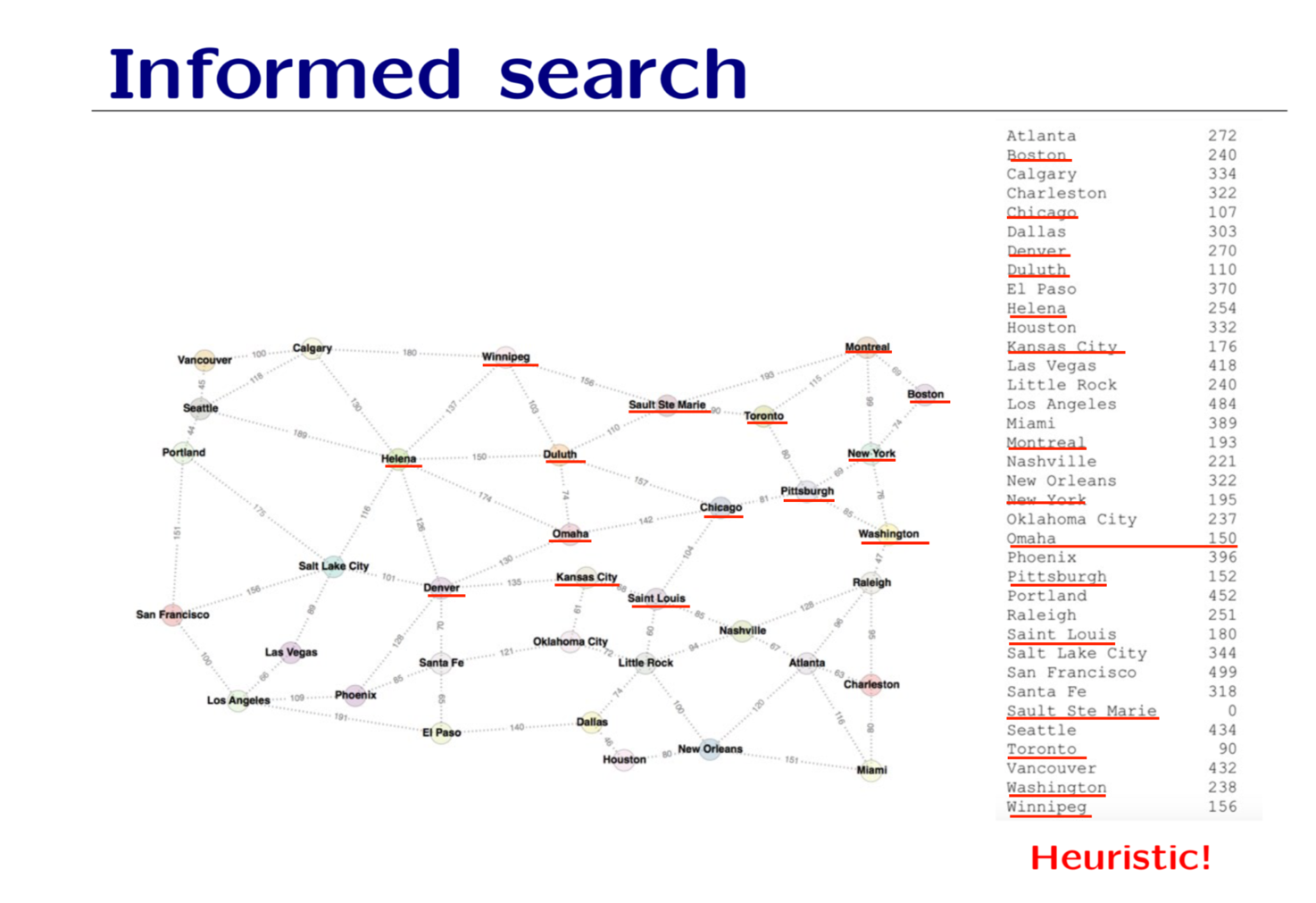
На рисунках 3.1 и 3.2 показаны примеры работы программы для разных алгоритмов поиска. Также на рисунке 3.3 показана сама карта с городами.

Рисунок 3.1 – Жадный поиск (Дейкстры)



Рисунок 3.2 – Поиск А\*



Рисунок 3.3 – Карта с городами

Выводы

В рамках данной лабораторной работы в качестве алгоритма жадного поиска был выбран алгоритм Дейкстры, так как он является довольно простым с точки зрения понимания и реализации, а также, собственно, по определению является жадным.

Результаты двух алгоритмов совпали, что позволяет удостовериться в правильности результатов.