Breadth First Search Algorithm in Finding Shortest Flight Route

Breadth First Search Algorithm Application

Kevin Ryan / 13519191
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): kevinryanwijaya@gmail.com

***Abstract*—This paper is written to describe the example of breadth first search algorithm (BFS) implementation in solving daily life problems. The problem tackled in this paper is finding the shortest flight route between an origin city to a destination city. This paper is written with the hope of helping readers understand the integration of algorithms, especially BFS algorithm, in solving daily life problems.**

***Keywords*—breadth first search, flight, shortest path.**

I. Introduction

The advancement of technology in this modern era has enabled people to travel vast distances in a short period of time. Comparing to the way of life then, the means of transportation nowadays is significantly faster than those in the past. Lots of days, months, or years even is needed to travel from one place to another in the past. A travel that took years to complete, now only takes several days. All this could be achieved due to public access to transportation technological marvel, air transportation (mainly planes or helicopters).

Airline companies offer everyone the access to board planes to travel from one place to another. They provided flight route available to everyone who wishes to board the plane. Often there is no direct flight path from one place to another. To solve this problem, people usually transits in another city to resume flight to the city of destination. People usually do this planning manually and may cause inefficiency in planning their flight route. To solve this problem, the author of this paper wrote a program to solve this problem by integrating and implementing breadth first search algorithm to find the shortest flight path from city of origin to city of destination.

II. Theories

A. Breadth First Search

Breadth First Search algorithm or BFS algorithm is a widening searching algorithm which preorderly visits every nodes in a graph. This means iterating through every nodes adjacent to previous nodes, and continues to do so until every node has been iterated through.

This algorithm utilizes a queue to store every nodes visited by the algorithm. These nodes are needed as a pivot to visit or iterate through all the adjacent nodes. Every nodes visited will be queued only once. This algorithm also uses a boolean table to store every visited nodes. This is done in order to ensure non-repetition in node visiting.

In breadth first search algorithm, visited child nodes will be stored in a queue. This queue is used to reference other adjacent nodes following the queue order to further reveal how BFS and its queue is used.

The steps of BFS algorithm :

1. Queue root node
2. Take the node from the head of queue and check if the queue meets the solution.
3. Returns the result if the node meets the solution.
4. Queue every nodes adjacent to the node (child node) if the node doesn't meet the solution.

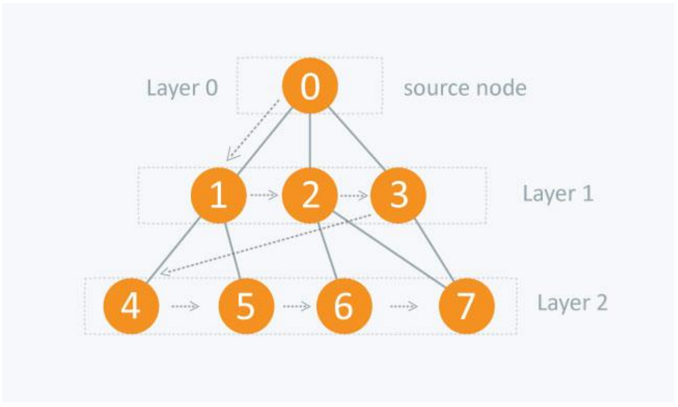


Figure 1. Visualization of Breadth First Search Represented in Graph

(https://he-s3.s3.amazonaws.com/media/uploads/fdec3c2.jpg)

B. Flight Route Finding with BFS

Suppose we have the data of every city and all its available flight route, we could iterate through every nodes (cities) preorderly with the help of breadth first search to find the shortest path from city of origin to city of destination. For example, suppose we wanted to travel from city of origin A to city of origin D. City A has a flight route to B and C, and city C has a flight route to D. We could iterate through every cities and its flight routes to find the path from city of origin D. The first step of this process is queueing A and iteration through all its available flight route. This results in A-B and A-C. Next, dequeue A from the queue and iterate through every flight route of the city B (because it's the last city of the head path of the queue). This results in nothing because B has no available flight route. After dequeueing this path, we iterate through every flight route in the city C. This results in A-C-D. Because the solution has been met, we stop iterating and return the result. This result is the shortest path from city of origin A to city of origin D.

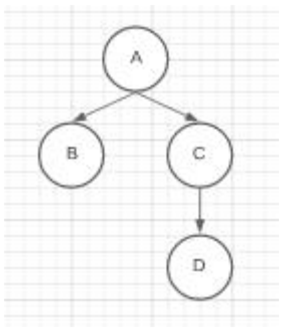


Figure 2. Graph Representation of The Cities

This methods is guaranteed to return the most optimal results based on flight count. This could happen due to how BFS iterates preorderly through every nodes. BFS checks through every possible path starting from the root node and all its adjacent nodes. However, this program only ensures that users take the least amount of flight and does not guarantee the shortest distance of a path.

III. Implementation

This program is written in Python, a high-level programming language. The way this program works is by utilizing a class city which has attributes of its name and all of its flight route stored in a list

classes.py

```
# Class City
class city :
    # Constructor
    def __init__(self, name) :
        self.name = name
        self.routes = []

    def addRoute(self, cityToAdd) :
        self.routes.append(cityToAdd)
```

Next, the program will initiate a list read from a txt file called "input.txt". The program will parse each line and split the string read by the char ','. For every cities in input.txt, the program is going to make a city object of its own and store every routes in the attribute "routes". All this are done in a function called makeListCities which returns a list of objects of all cities and its flight routes.

functions.py

```
from classes import *

def makeListCities() :
    listCities = []
    with open("input.txt", 'r') as file :
        for line in file :
            tmpline = line.strip()
            line = tmpline.split(",")

            # Check first item in txt
            existed = False
            for item in listCities :
                if item.name == line[0] :
                    existed = True

            if not(existed) :

listCities.append(city(line[0]))

            # Check second item in txt
            existed = False
            for item in listCities :
                if item.name == line[1] :
                    existed = True

            if not(existed) :

listCities.append(city(line[1]))

            # Add to route

listCities[getIndexOfCity(listCities,
line[0])].addRoute(listCities[getIndexOfCi
ty(listCities, line[1])])

    return listCities

def getIndexOfCity(listCities, cityName) :
    for item in listCities :
        if item.name == cityName :
            return listCities.index(item)
```

Now in the main program, after listCities has been initialized, the program will initialize a new list acting as queue which was first initialized with class city of origin. The program keeps track of the iteration index through initializing a variable called iterationIndex. The program will keeps on increasing the iterationIndex while the solution is not met and stops if solution can't be met. The program is going to add every route to the path in queue at the element iterationIndex and add it to the queue. This process is done until program has successfully found the solution or if the city at the end node has no more routes. All of these are integrated and implemented in main.py

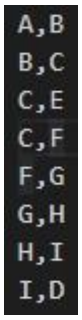
main.py

```
from classes import *
from functions import *

listCities = makeListCities()

originCity = str(input("Origin city : "))
destinationCity = str(input("Destination
city : "))

originCityClass =
listCities[getIndexOfCity(listCities,
originCity)]
destinationCityClass =
listCities[getIndexOfCity(listCities,
destinationCity)]

bfsPath = [[originCityClass]]
iterationIndex = 0

found = False

while not(found) :
    for item in
bfsPath[iterationIndex][len(bfsPath[itera
tionIndex])-1].routes :
        tmpBfsPath =
bfsPath[iterationIndex].copy()

        tmpBfsPath.append(item)

        bfsPath.append(tmpBfsPath)

        if item == destinationCityClass :
            path = tmpBfsPath
            found = True

    iterationIndex = iterationIndex + 1

for item in path :
    print(item.name, end = " ")
```

IV. Test Cases

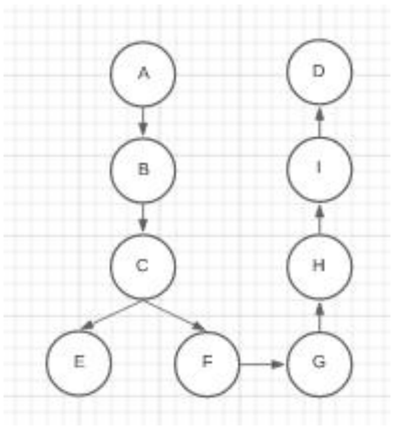
I. Test Case 1

Suppose you are trying to find a path from city of origin A to city of destination D given the routes available are as follow :

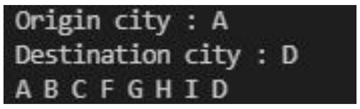


```
A,B
B,C
C,E
C,F
F,G
G,H
H,I
I,D
```

The graph of this routes are as follows :



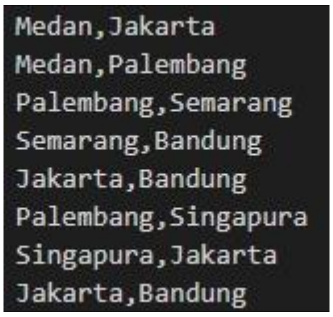
We could see that the path are A-B-C-F-G-H-I-D. We could check this by using this program. The result of the program are as follows :



```
Origin city : A
Destination city : D
A B C F G H I D
```

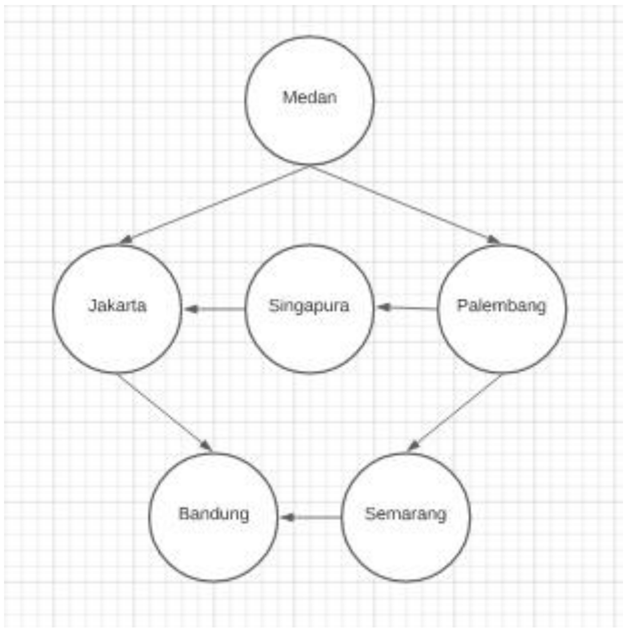
II. Test Case 2

Suppose you are trying to find the flight path from city of origin Medan to city of destination Bandung given the routes available are as follow :



```
Medan,Jakarta
Medan,Palembang
Palembang,Semarang
Semarang,Bandung
Jakarta,Bandung
Palembang,Singapura
Singapura,Jakarta
Jakarta,Bandung
```

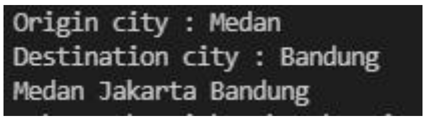
The graph of this routes are as follows :



From the graph, we could see that there are several routes from Medan to Bandung. Those are :

1. Medan -> Jakarta -> Bandung
2. Medan -> Palembang -> Semarang -> Bandung
3. Medan -> Palembang -> Singapura -> Jakarta -> Bandung

The supposed result are the first one, which is the shortest path from Medan to Bandung. To further confirm the results, we are going to use the program to help find the path.



```
Origin city : Medan
Destination city : Bandung
Medan Jakarta Bandung
```

V. Source Code

https://github.com/kevinryann/flight-route

References

[1] Delima Zai, Haeni Budiati, Sunneng Sandino Berutu, Simulasi Rute Terpendek Lokasi Pariwisata di Nias Dengan Metode Breadth First Search dan Tabu Search.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 26 April 2021

[illegible]

Kevin Ryan, 13519191