UMEÅ UNIVERSITET
Institutionen för Datavetenskap
Rapport obligatorisk uppgift

Datastrukturer och algoritmer (Python) 7.5 p
5DV150

Obligatorisk uppgift nr

5

Namn	Aladdin Persson
E-post	Aladdin.persson @umu.se
CAS	Tfy17api
Datum	2019-03-15
Handledare	Thomas Johansson Henrik Rosén

Innehåll

1 Introduction 2

2 Breadth-first search 3

3 Program description and usage 4

4 Time complexity analysis 5

5 Conclusions 6

1 Introduction

Graph problems can be found almost everywhere in computer science and hence there has been thorough analysis and clever algorithms invented. Just as sorting is used very frequently in the solutions to larger problems there exists fundamental algorithms for graphs that can generate solutions to more complex problems. One of such algorithms is breadth-first search frequently abbreviated as BFS, which is an algorithm for traversing or searching a graph. It is one of the simplest algorithms but an important one that is the archetype to the solutions of many graph problems.

In this report we wish to examine how a breadth-first search algorithm works in finding connected components from a specific node. Specifically if from a current location A, check with the help of a BFS algorithm if location B is reachable from our current location.

In section 2 we will review how the BFS algorithm works, and follow in section 3 with a program description explaining how the program works and how the user interacts and uses the program.

In section 4 we will determine the time complexity of the different parts of the program, reading or loading the graph and finding connected components with the usage of BFS. And lastly in section 5, we will go through a small testing procedure that was made to minimize the chances of mistakes.

2 Breadth-first search

The BFS algorithm is an fundamental algorithm for traversing a graph. It can be shortly explained by that it starts at an arbitrary node and explores all the neighbor nodes currently accessible by our start node. It then proceeds by accessing all the neighbor nodes to all of those neighbors and continues to do so until all connected nodes have been reached. For BFS it is quite convenient to use a queue to keep track of the neighbor nodes to traverse next. Pseuodcode for this algorithm is as follows

Algorithm 1 BFS(graph G, start node s)

```
1: mark all nodes as initially unvisited
2: mark s as visited
3: let Q = Queue (FIFO) initialized with node s
4: while Q ≠ ∅ do
5:     Remove first node of Q, call it v
6:     for each edge (v,w) do: // each edge that connects v to another vertex w
7:         if w not visited then
8:             mark w as explored
9:             add w to Q (at the end)
```

3 Program description and usage

The program we wish to implement in this project that utilizes the breadth-first search algorithm is one that the user inputs a graph, a start node or location A and our goal is to determine if there is a path that connects the start node to a final node B. Note that we do not need to output the path to do so, which would not require too much addition to the code but is a not a requirement of this program.

Specifically it is important to note that the input for the graph between nodes only follows the following pattern:

<startnode> <finalnode>.

Where we know that they are strings which could be represented by words or numbers. Hence we need a program that read through each line telling the start node to final node and store them as strings.

After loading the graph, we can now ask the user to input two of the locations and we can run the BFS algorithm to check if there exists a path that connects these two nodes.

The usage of the program is that the program is run from the command prompt as follows:
python isConnected.py graph_file.txt.

And the program then asks the user for which two nodes it wants to check connectivity between. As long as the user has not quit the program it will continue to ask for two nodes to check. After the program has been run an example to check connectivity can be the following:
NodeA NodeB.

This will check connectivity between NodeA and NodeB where nodeA is the starting node.

It is important to know that programs work as intended and continue to do so if changes occur later on. For this reason I've included a test program with this report to check if it passes a basic testing procedure. This can be called with the following command:
python test_isConnected.py.
Just make sure you have the included files for this report and that it is run in the same folder as these files.

4 Time complexity analysis

The analysis of the program can be separated into the part of reading the graph, running breadth-first search and also the complexity of asking the user for input to determine connectivity.

First let us make the notation clear, that we have an input graph: G(V,E) where the graph contains vertices V and edges E.

By examining the complexity of loading the graph, it can surely be read in many different ways, but the method chosen in this report was to first loop through every line of the text file describing the connection between nodes. And after that it is necessary to add each of these connections into a table, where the key is a specific node and the value is a list of all the connections. This can be done in constant time as insertions to table is constant and therefore the limitation is in looping through each line of text, making it $\mathcal{O}(|E|)$. This is considering that each line corresponds to an edge.

For the breadth-first search algorithm if we consider the worst case, we know that no node will be visited two times, and no edge will be visited two times either. This makes it $\mathcal{O}(|V| + |E|)$ depending on whetever there are more vertices than edges or vice versa.

Lastly we consider the complexity of reading the input and since the program would continue indefinitely if the user never wrote "quit", let us assue there is only a single line from the user, i.e for example NodeA NodeB. What we have to do before is to read the graph which we have argued is $\mathcal{O}(|E|)$ and then we also have running breadth first search which is $\mathcal{O}(|V| + |E|)$. Since reading the input is constant we will have

$$\mathcal{O}(|E|) + \mathcal{O}(|V| + |E|) = \mathcal{O}(|V| + |E|).$$

And therefor the time complexity for this program should be $\mathcal{O}(|V| + |E|)$.

5 Conclusions

In this project we saw how to use breadth-first search to be able to test if there is a connected component between two nodes or locations. We also saw that the time complexity of it is rather fast and can therefore be used on rather large graphs.

Improvements of this project is to include more robust testing procedures and to test the complexity experimentally of the claimed complexity of this program to verify that this holds.