University of Tromsø

INF-2101 Algorithms

- You are likely to be eathen by a Grue!

By Morten Groennesby & Ruben Maeland

October 3, 2012

Contents

1 Introduction

In this assignment the program have to get through a labyrinth in order to find Sarah's sister, Toby. Toby is stolen by 'tussekongen', and well protected by the labyrinth. The job, is for the program to find a way to toby through the labyrinth. If it does, you can choose between marrying the 'tussekongen' or just go home.

1.1 Abstract

The problem in this assignemt are how to represent the graph. Since the labyrinth isn't a already defined graph, the program have to build the graph as it goes, in order to know where it have been and where to go next (to new unvisited nodes in the graph). So the goal is to solve the labyrinth in shortest amount of time.

2 Technical Background

Basic knowledge for Depth-first[Wikipedia.org(2012b)] search, breadth-first search[Wikipedia.org(2012a)] and Dijkstra's algorithm[Wikipedia.org(2012c)] and general knowledge of graphs and graphrepresentation, are required to read this report.

3 Design

Since the problem in this assignment is representation of the graph, the first thing that were made was a class Vertex. That class have one setter, and one getter methods. The getter get's the current position for Sarah in the labyrinth, and the setter set the position to a given position. This is just to represent where in the labyrinth she is at any time.

When the search starts, the current position of Sarah, is set to (0,0). This is just to get a reference point to work from. The program should move depending on how the sorroundings look like. Then the program should use some algorithm (either BFS, DFS or Dijkstra's algorithm) to find the shortest path to toby. Ofcourse, there should be some helping functions to discover corners and such, so it don't get 'stuck' at those nodes.

The backtracking part of the program should also use some of the same algorithms as mentioned above, to find the shortest path to a unvisited node.

To determine if Toby is found or not, the program should have some boolean variables that is set to False, and get True when you find Toby in the list of sorroundings.

4 Implementation

The assignment is implemented using a depth-first search, this means that Sarah will go as far as she can in a specific direction until she gets stuck, and then backtrack through the same pattern until she finds a new unvisited tile to move to. In essence, Sarah will check out all tiles that can be moved to until she finds Toby.

4.1 Moving Sarah

Sarahs movement is based on looking around and finding an unvisited tile (walls are ignored), and moving to that tile, then repeating the procedure. Before we start moving Sarah, we need something to keep track of where she is in the labyrinth, a coordinate system. We think of each tile in the labyrinth as a Vertex, with a tuple to represent its location in the labyrinth. This way we can keep track of where Sarah has been, but only using tuples means that we will have to manually update Sarahs position. The labyrinth pre-code has no method for keeping track of coordinates. The way this is implemented is to use a Vertex class to hold our coordinates, update the Vertex instance for each move, and add the instances coordinates to a visited list for all coordinates we have been to.

The movement method is iterative, so for each iteration we use the pre-codes look method to determine which tiles around Sarah we can move to, and choosing the first direction we can move to based on the order of the conditional statements (in the order: south, north, east, west), as opposed to choosing a direction at random. After we move Sarah, we skip to the next iteration to open the possibility to move in the same direction again. We also use a stack structure (using list as a stack) to keep track of the most recent moves that has been made, to backtrack later if we get stuck.

4.2 Checking if we are stuck

The code has a help method to check if Sarah is stuck in the labyrinth. The method returns a list of two arguments, one directional string, and a Boolean value. This way we can determine if Sarah is stuck and we need to backtrack, or in the other case, which direction would be a valid move. Checking which direction has an unvisited node ensures that Sarah always will move in a new direction while backtracking, if there is one.

4.3 Backtracking

When Sarah becomes stuck in the labyrinth, the stuck check method will return a True Boolean value (it is true that Sarah is now stuck) and a None variable, ergo we cannot find a new direction for Sarah to move in. The movement code will call on another helping method to backtrack Sarahs movement up to now. This is where the stack will come into play, up to now the code has been adding Sarahs movement to a LIFO stack using the form:

```
Stack = [[(0,1), 'north'], [(1,1), 'west'], [(1,0), 'south']]
```

The stack is a list of lists, and each of the list elements on the stack holds Sarahs previous tuple coordinate visited, and the direction in which Sarah need to move to get back to it. The backtrack method pops of the last element of the stack, and sets the current position to the tuple contained in the stack element. It then moves in the direction given by the second variable of the list (string with a given direction).

4.4 Finding Toby

For each iteration a third helping function checks if any of the look lists arguments is 'toby'. If one of the tiles is toby, the loops statement is no longer false, and the method ends.

5 Discussion

Now that the program is finished, and the result looks quite different than the intention was, the first thing to mention is that neither BFS or Dijkstra's algorithm were used. Since we thought of the whole map as a graph, we used DFS, which means that Sarah moves as far as possible before backtracking. DFS is not the most effective algorithm for finding the shortest path to toby, since that is what BFS do, and DFS find's a random path to Toby. That is ofcourse, under the condition that there is a path to Toby.

Since the only way Dijkstra's algorithm could be used is in backtracking, and the fact that we don't build the graph as we go we were unable to use Dijkstra's algorithm in the assignment. Dijkstra's algorithm would ofcourse improve the efficiency of the program a lot, but that would require a total different approach for the whole assignment.

The way this could be done with dijkstra's is to have two dicts, one adjacency list and one dict to hold vertex information. By 'tagging' the last vertex which had a possible new route in the vertex info dict, you could easily search for it in the adjacency list and utilize dijkstras algorithm to find the shortest path back to the last vertex with unvisited adjacent nodes.

6 Conclusion

The program does work properly for all maps, but not for all gamemodes. The problem is the backtracking, which could be implemented with Dijkstra's algorithm to get it more efficient.

References

[Wikipedia.org(2012a)] Wikipedia.org. Breadth-first search, September 2012a. URL `http://en.wikipedia.org/wiki/Breadth-first_search`.

[Wikipedia.org(2012b)] Wikipedia.org. Depth-first search, September 2012b. URL `http://en.wikipedia.org/wiki/Depth-first_search`.

[Wikipedia.org(2012c)] Wikipedia.org. Dijkstra's algorithm, September 2012c. URL `http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm`.