INF-2101 Algorithms — You are likely to be eatern by a Grue!

Bjørn Ludvig Langaas Johansen <bjorn@octanium.net>,
Johannes Arctander Larsen <johs.a.larsen@gmail.com>

University of Tromsø

2012-10-05

Abstract

The assignment was to implement an efficient "walker" for a given labyrinth, set to the movie Labyrinth. This was done using Dijkstra's algorithm to be able to find new, unexplored areas when needed to, in addition to prioritizing adjacent unexplored tiles before using Dijkstra's algorithm. The whole implementation of the "walker" has been contained within the class `AiPlayer`. At the end of the assignment, the resulting implementation walks all fairly normal labyrinths tested, with reasonable "intelligence" and manages to find Toby before the time runs out.

1 Introduction

The assignment was to implement a way to traverse a given labyrinth, using methods from the field of graph theory. The theme of the assignment is set somewhat loosely to the movie Labyrinth[1](1986), where a girl, Sarah, unfortunately manages to "wish away" her brother Toby. She then have a change of mind, and is given a chance to get her brother back by finding her way through a labyrinth filled with traps and monsters before the time is up.

1.1 Requirements

- "Walk" a given labyrinth in a reasonable time
- Use graph theory to determine where to go next
- Try to find Toby?

1.2 Technical background

1.2.1 Graph theory

Graph theory is the study of connections or affiliations between entities. A graph is then a collection of entities(or objects/nodes) and the connections between them(edges). Graph theory is the study of the properties of the graph. This can be the "cost" getting from one node to the other, the amount of edges needed to traverse to get from one object to the other.

1.2.2 Dijkstra's algorithm

Dijkstra's algorithm[2] is a graph searching algorithm that traverses a graph, finding the shortest path from a given node to all nodes in the graph, resulting in a shortest path tree[3]. It can alternately produce a shortest-path route from one given node to another, by stopping the algorithm when the algorithm reaches the requested node.

2 Implementation

The implementation utilizes its own `python` class called `AiPlayer` for all tasks needed to walk the labyrinth. To walk a labyrinth, one instance of the class is initiated, with the client side labyrinth as a parameter. The `play` method of the `AiPlayer` is then called to begin walk the labyrinth.

2.1 `AiPlayer` class

The class itself holds the position of the "player", a dictionary with information about tiles it has seen, and the labyrinth it "walks". The dictionary of seen tiles is keyed on position, and contains `None` if the block has not been visited, and a list of weights of the surrounding blocks, if the block has been visited. This means that the dictionary will only hold keys that are tiles that are possible to walk on, but associated with the key, information about walls can be stored.

2.1.1 `play` method

The `play` method runs in a "infinite" loop until an extraordinary event occurs(Toby is found, time runs out, ...). The loop starts with an empty, temporary list of adjacent tiles that not has been explored. It then adds the unexplored tiles around the current position to the temporary list of adjacent, unexplored tiles.

[1] http://www.imdb.com/title/tt0091369/
[2] http://en.wikipedia.org/wiki/Dijkstra's_algorithm
[3] http://en.wikipedia.org/wiki/Shortest_path_tree

If there is adjacent unexplored tiles the "player" walks to the first(just a good of a choice as any) in the list. If there are no unvisited adjacent tiles it calls the method `go_to_closest_unexplored_tile`. At the end of each loop iteration, the method looks around with the `look` method provided by the `labyrinth` class, parses the output, and adds the tiles around its new position to the list of seen tiles.

2.1.2 `go_to_closest_unexplored_tile` method

The method first runs Dijkstra's algorithm to get a list containing a path to unvisited tiles, and a list of distances to the tiles. The method then iterates through the list of distances to find the closest unvisited tile. A path to the tile is then constructed by "backtracking" the path until a path from the unvisited tile to the current position has been established. The method then "walks" the path to the unvisited tile.

2.1.3 `parse_look` method

A separate method for parsing information from the server when using the `look` method provided by the `labyrinth class`. This uses the `weights` method to give the four adjacent tiles it sees weight based on what kind of tile it is(a tile, a wall or Toby).

2.2 `dijkstras_algorithm` method

The method takes a source node, s, as its argument. It starts with two dictionaries, `dist` (distance from s) and `previous`, both keyed on position. `previous` is initialized with `None` for every node and the distance dictionary is initialized with 0 for s and `sys.intmax`($\approx \infty$) for the rest.

The algorithm makes a queue, `Q`, out of the nodes and iterates until this queue is empty. The node, $n \in$ `Q`, closest to s is extracted from the queue, and its adjacent nodes, $a \in A$, that also is in the queue are iterated through. If the distance from n to s plus the distance between a and n is greater than the distance from a to s, then the distance of a is updated to the shorter distance and its previous node is set to n.

The function returns both dictionaries.

2.3 Weight

A set weight is set to Toby, for easy checking early in the loop that does the walking. The Dijkstra's algorithm used for finding close, unexplored tiles, does not use weights for any calculation of importance, even though there is set a hardcoded weight of 1 for traveling from tile to tile.

3 Discussion

3.1 Problems, limitations

As with all ways of walking a labyrinth, a wrong turn, or large open spaces can cause the walker to spend a lot of time exploring unnecessary or wrong areas. In addition, if Toby is located in an open space, there is no way of actually finding Ttoby without "scanning" the space, and using a lot of time.

3.2 Special properties

In the parts of the code where a adjacent unexplored node are picked for exploration, a inherent prioritization is made between which direction to go. This comes from the sequence in which directions are parsed. In our implementation the prioritization is: **North**, **South**, **West**, **East**, and can comes from the sequence of which the tiles comes from the server(`labyrinthServer.py`) when looking. This is also replicated in all the methods that parses through all four directions, which also uses this particular sequence.

This may change between implementations and can make two otherwise identical implementations behave completely different(i.e. turn left instead of right, or similar choices).

3.3 Inconcistencies between assignment text and precode

It should be noted that the description of the assignment by the assignment text given in the file "`INF-2101 oblig 2.pdf`", differs from the precode given in several areas. These inconsistencies mainly concern specification of the cost of looking, and the cost of "walking into a wall". Were the assignment text says there is no cost in time by looking and "walking into a wall" but the server still deducts time from the time left when doing these things.

4 Conclusion

The implementation "walks" a given labyrinth with reasonable efficiency, although it may seem "stupid" to the observer when it makes a wrong turn even close to the goal(there is no way for it to see the goal from "far away").