Maze Challenge

Orestis Tourgelis

January 6, 2017

1 Introduction

In this document we are going to describe briefly the problem we are trying to solve, the general idea behind our approach, as well as the implementation and the design principles we followed.

A bot is trying to find a goal in a maze. It doesn't have any previous knowledge of the maze or the position of the goal. It gradualy builds it's knowledge by inspecting the cells that are visible to it. It is allowed to move in four directions North, South, East and West and also to "see" in the same directions.

2 General Approach

The bot starts from an arbitrary position. The first step is performed by randomly choosing one of the available cells, in one of the four directions. In the consequent choices we prefer to maintain the same direction if there is an available cell, as this increases the bots visibility on the grid (a move in the same direction will result in three newly seen cells). If the cell in the same direction is not available it will choose at random one of the remaining choices, given that there exist some newly seen cells.

If there are no newly seen cells in the bots neighbors this means that we have reached a dead end and the bot is required to retreat to a previously seen cell. The selection process consists of two parts. We store the newly seen cells in a data structure and we remove items from the structure until we get a cell that we haven't yet seen all of its neighbors. The reason for discarding the cells that we have already seen all its neighboring cells is that visiting this cell is not going to increase the visibility of the bot and thus is not going to point us to an unexplored area of the grid.

Once we determine the cell to which our bot will retreat, we use the bots previous knowledge to construct a grid, and run a path finding algorithm to determine the path of our retreat.

3 Implementation

In our implementation we tried to apply the single responsibility principle and useful design patterns to achieve separation of logic and flexibility.

3.1 Grid and GridBuilder

Taking into consideration that the functionality for the base *Grid*, and the *Grid* constructed by the bots memory should be essentially the same, I extracted the parsing of files into a separate *GridBuilder* class, based on the logic of the builder pattern [1].

The *Grid* class is responsible for returning the available neighbors for a position, and responding to whether a position is the goal or not.

3.2 Retreat algorithm

Our bot has the flexibility to let the client deside on the algorithm it is going to use to find the retreat path. We have created a parent class *Algorithm* that forces, the child classes to implement the needed methods that the bot is using. So even though duck-typing is in general the prefered approach, using this simulation of an abstract class, we can ensure that our algorithms have a consistent interface. We have implemented two such algorithms, the bfs and the $A*$. This implementation was based on the strategy pattern [1].

The A* algorithm is a well known path finding algorithm. It is actually an alternation of the Dijkstra algorithm using a heuristic. In our case, of the two dimentional grid, the manhattan distance is a metric with low computational cost.

We are using a min-heap priority queue to speed up the algorithm. The priority queue is using an abstraction of a binary tree implemented as an array, where a node k is the parent of nodes $2k$, to its left and $2k+1$, to its right. When inserting or deleting a new node we need to traverse the tree and exchange any nodes that do not satisfy the min heap property. This process is $O(logn)$. We also need to heapify the whole structure when the cost to a cell is updated since the min-heap property is violated, which is $O(n * logn)$ [3].

3.3 Bot and BotMemory

The *Bot* class is a composite object. It consists of a *retreat_algorithm* which can be determined on run time, and of a *BotMemory* class. The reason for this design is to keep the *Bot* class to a minimum and maintainable size.

The bot is responsible of using the information provided to it by the *get_neighbors* method of the *Grid* class to determine its next move and to build up its memory. By calling the move method, the bot repeats the logic we described previously, until it either finds the goal, or sees all free cells in the grid.

The bots memory responsibility is to store any information related to the grid and the bots movement. It is capable of creating an instance of the *Grid* class based on progressively collected data. This grid is then used by the *retreat_algorithm*. It is also responsible to decide which is the best available cell to retreat to.

BotMemory class also has the flexibility to change on run time. We can provide through the constructor, the data structure that it will use to find the cell to retreat to. By changing the data structure *frontier*, from a *Stack* to a *Queue* the bot will not retreat to the last seen cell (which is the default behavior) but to the first seen available cell. This change in the data structure is essentially the difference between the depth first search and the breadth first search algorithms. We expect that the usage of a *Queue* will result in poor performance comparing to the *Stack*.

3.4 MovementView

The *MovementView* object is using the file containing the grid data and the bot's path array, after the *move* method concluded, to display the course the bot followed.

References

[1] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John.
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley. ISBN 0-201-63361-2.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms.
MIT Press. 1990. ISBN 978-0-262-03384-8.

[3] `http://theory.stanford.edu/~amitp/GameProgramming/`