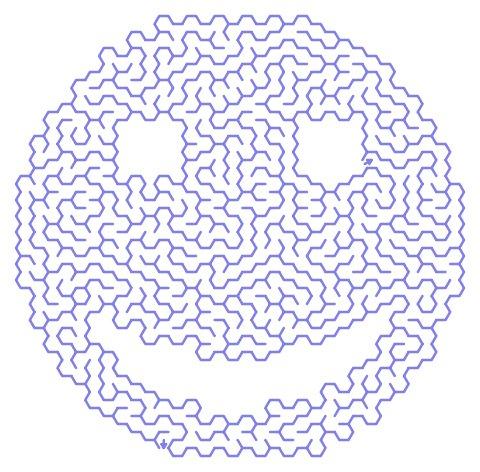
Assignment 1: Smiley Goes Home Like A\*

by

Martin Cousineau - 6562766

Nicolas Eveillard - 5632336

Scott Hassoun - 6061968



Submitted to

Professor Nour El-Kadri

for the course

CSI4106 - Intro. to Artificial Intelligence

University of Ottawa

October 10th, 2013

[Introduction](#h.oi42zasbwwv0)

[Procedure](#h.3sgiyyvquuby)

[Screenshots](#h.4wh1ctrzuxg1)

[Conclusion](#h.nosw7oibd6ii)

# Introduction

The goal of our project was to implement the A\* algorithm given a heuristic function to make our agent (the smiley) traverse the graph in an efficient manner while avoiding obstacles. Our algorithm would attempt to apply these heuristics to a predefined game board in order to find the "optimal" solution for all four agents located at different positions on the board.

# Procedure

A consensus was reached in the group to deploy this application in Java for the following reasons. Java is object-oriented. There weren’t many objects in our implementation and we could have implemented our program in a language such as Scheme or Prolog, but the familiarity with Java and general object oriented programing was greater amongst our group. Although our implementation took ~500 lines of code compared to a Prolog implementation of possibly 30, it was easier for us to understand and model this problem in the object-oriented paradigm.

Furthermore, the A\* algorithm needed to be fully understood in order to implement it. Hand drawn solutions were helpful in understanding how the algorithm worked. This included drawings of the grid, grid-traversals, etc.

Our program incorporated the following components: a heuristic method, an A\* tree generation, a board class, and a main class to run the program. Our heuristics method is included in *Solver.java* and the method is *getHeuristics(Cell c)*. This returns an integer value based on the distance that was to be calculated as specified by the assignment instructions. Our *checkCell* method, which is in the same class we use this *getHeuristic* function, decides what is the best cell to move towards its goal. The A\* tree was built by using the *Node.java* class that defines its parent, current position, heuristic value, and can calculate its overall weight by adding the heuristic value and the distance from the starting cell, which is calculated recursively through the node’s parent. We built the Board and Cell classes to keep track of the information on the board, since the tree is generated based on the setup of the cells on the board. The A\* tree decides which is the route to take and it will always find an optimal solution given that the heuristic method is any good.

In the board object, cells are stored in a bidimensional array of cells. Since in Java elements are stored in an array starting at position 0, the coordinates of the top-left cell is (0,0), the top-right cell is (width, 0), the bottom-left cell is (0, height) and the bottom-right cell is (width, height). Since the assignment required us to center our board on the home cell, we convert those coordinates after the fact when printing the result of our algorithm in the main method.

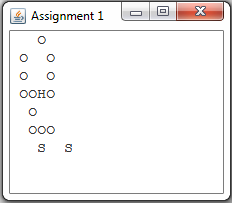
The project started using an agile software development technique in software engineering called pair programming. This is beneficial because team members are able to work together, collaborate and not get as frustrated by bugs, since there is a team member there to help. When the team started feeling more comfortable with the implementation of this program, classes and methods were divided and a certain autonomy was achieved. This included consulting each other for any requests for methods or changes to the implementation of our parts. After about 3 hours of work, our greenfield project had the most essential parts coded. Afterwards, our team put these parts together and our algorithm printed out the paths taken by all the smilies.

In order to make our implementation more visible and understandable, so that the user would not have to trace a grid with (x,y) coordinates, our team implemented a GUI displaying the position of the different smileys at every step of the creation of the A\* tree nodes. The smileys do not move smoothly on the board since the algorithm tries multiple paths at once, but they do end up home no matter what.

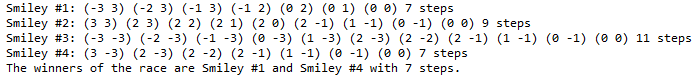
In further testing, we gave our algorithm differently sized boards with varying amounts of smileys and obstacles and the A\* algorithm continued to perform as expected.

# Screenshots

The following are screenshots from the runtime execution of our program. You can see (without animation) that our smiley “S” in the middle of the bottom row is traversing around the obstacles “O” to get home “H”.



The output is also produced at the end of execution for each smiley. To test out the application in more detail, feel free to change the board.txt file and watch if our smileys find their way home. Please note that our program requires a matrix with a consistent amount of columns and rows, and will produce an error otherwise.



# Conclusion

In under 5 hours of work, our team was capable of producing a Java implementation of the A\* algorithm using specific heuristics to reach its goal. Without the A\* algorithm, this implementation would not have been “neat” and likely to be less efficient compared to ITI1121 where some members of our team had a similar assignment using inefficient rules (such as turn left in no obstacle, otherwise turn right, unless that path has been previously visited, etc). That algorithm had lead to agents getting stuck and repeating the path in loops. As for our application, we are grateful to see that it will always find the best path to take, and will always succeed.

It is interesting to mention that even after having studied and mostly understood the A\* algorithm, through the GUI output we were able to find bugs in our solution algorithm. Notably, our algorithm was doing a slightly more efficient depth first search. This behavior was fixed upon further inspection of the A\* algorithm and our code.

It should be noted that in different instances of the game board that were tested (with obstacles in different places) sometimes the algorithm proved poorly. Eventually it would get to its destination although not as efficiently as possible. The reason for this is that the given heuristic looks vertically and then horizontally, but in different game setups the AI might get a better choice checking horizontally then vertically or going in the straightest lines as possible. It is difficult to suggest a way to improve this behavior without allowing the algorithm to know or predict the obstacles between it and the goal.