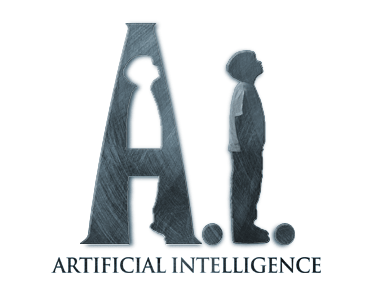
Applied Intelligent Systems

Final Report

SB684 - 12845703

Samuel Bowen

Table of Contents

[Introduction 1](#_Toc420253087)

[The Game 1](#_Toc420253088)

[The Maze 1](#_Toc420253089)

[The Player 1](#_Toc420253090)

[The key bindings 2](#_Toc420253091)

[Movement 2](#_Toc420253092)

[Debugging 2](#_Toc420253093)

[The Agents 2](#_Toc420253094)

[The Applied Intelligence 3](#_Toc420253095)

[All Agents 3](#_Toc420253096)

[Nose 3](#_Toc420253097)

[Ear 4](#_Toc420253098)

[Eyeball 4](#_Toc420253099)

[Spatial Awareness 5](#_Toc420253100)

[Analysis of feedback 5](#_Toc420253101)

[Findings 5](#_Toc420253102)

[References 6](#_Toc420253103)

# Introduction

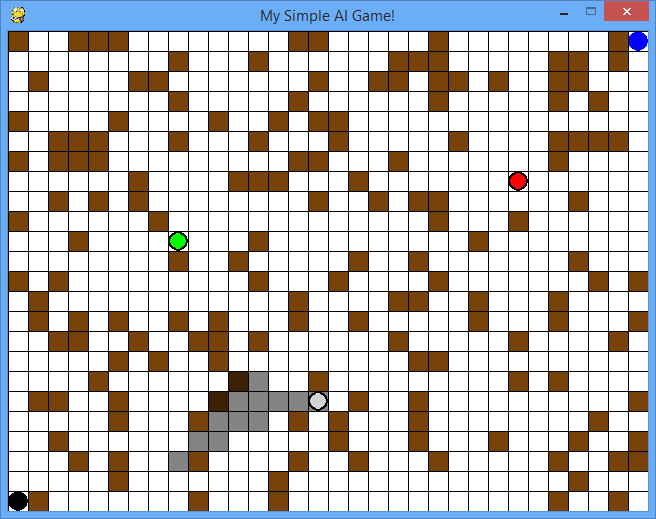
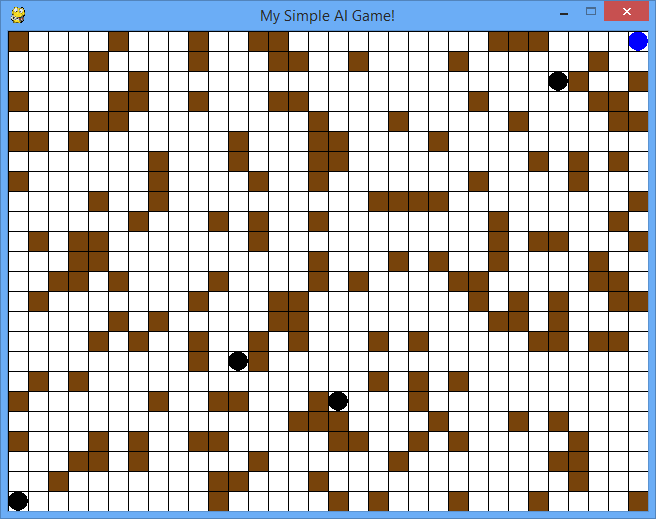
In this report I will be explaining what my program does, how it works and all of the features that I have included. I will then explain the artificial intelligence that has been implemented in the program; how the AI thinks and processes information, and the how it uses that information to try and achieve its goal.

# The Game

My program is a simple maze game where the player needs to reach a goal. The map contains a randomly generated maze. While traversing through the maze the player will need to avoid three different enemy agents. The three agents have different artificial intelligence, they will search for and chase the player in different ways. The program also tracks how many games have been played and the results of each of the game, recording who managed to win.

## The Maze

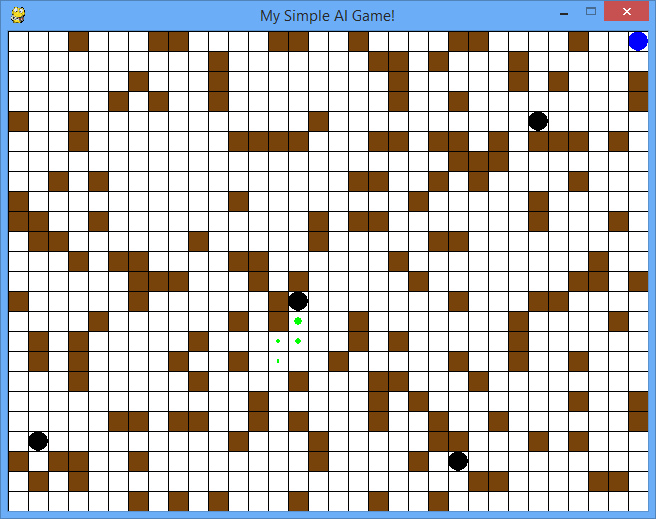
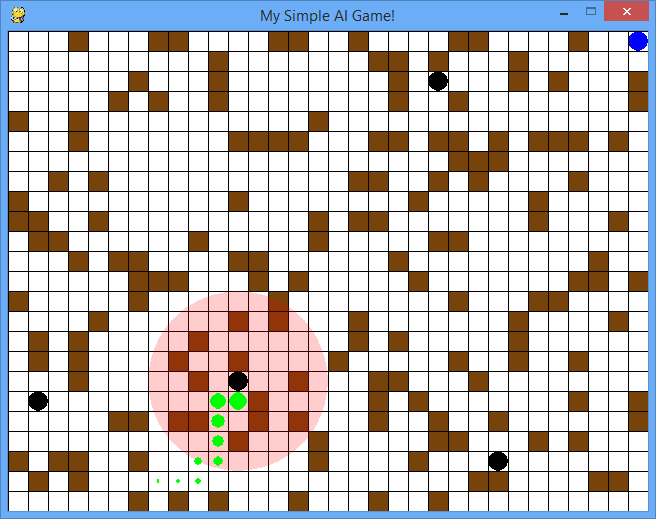
The maze is a 32 by 24 coordinate grid, which means there are 768 different coordinate positions. One quarter of the maze contains walls, players and agents will not be able to move onto the positions which contain a wall, also the enemy agent Eyeball will have his line of sight impeded by the walls. The goal state is marked on the map as a blue circle (top right).



*Figures 1 & 2*. In these two pictures you can see that the walls have been generated randomly each time, also the goal state in the top right. The first image is how the game should be played, where the player cannot see which agent is which, however in the second image you can identify the different agents. The green agent is Nose, the red agent is Ear, and the grey agent is Eyeball, Eyeball has his line of sight (grey blocks) blocked by the walls.

## The Player

The player’s goal is to move through the map without being caught by the agents and get to the blue circle in the top right of the map. The player has 5 movement options; up, down, left, right and to stand still. Moving will produce sound as he moves and will leave a smell where he has moved from, where standing idle will cause the smell to reduce in potency and the sound will disappear.



*Figures 3 & 4*. In these two images you can see the “sound” that the player is producing whenever he moves (the red ring), you will also notice his “smell” trailing behind him (the green circles). The smell decreases in potency as the player makes moves, causing his smell to disappear from parts of the maze. In the second image you will notice that the player has stood idle for a few turns, causing the sound ring to disappear and smell to be greatly reduced.

## The key bindings

### Movement

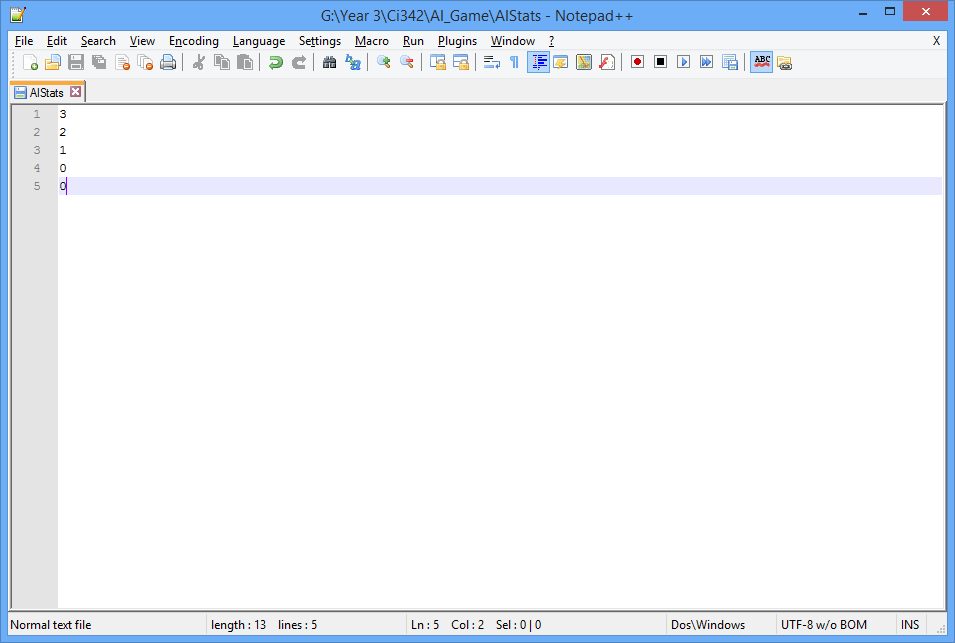
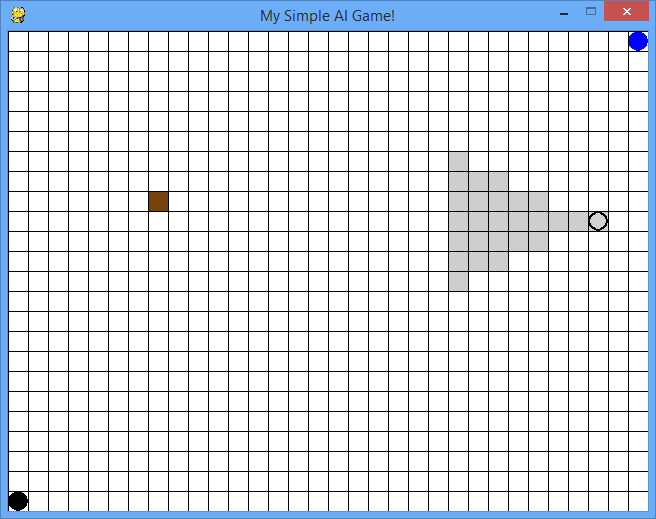
Up Arrow – Move up one space  
Down Arrow – Move down one space  
Left Arrow – Move left one space  
Right Arrow – Move right one space  
Spacebar – Stay idle for a move (see figure 4)

### Debugging

Escape – Closes the game  
‘Z’ – Disables Agent “Nose”  
‘X’ – Disables Agent “Ear”  
‘C’ – Disables Agent “Eyeball”  
‘V’ – Enables “Colour Mode” – Colours the agents and display’s Eyeball’s line of sight (see figure 2)

## The Agents

There are three different enemy agents that have been implemented; Nose, Ear & Eyeball, each agent will track the player by their different “sense”. Nose uses smell, by searching for and following the player’s scent Nose is able to hunt down and track the player’s movements around the maze. Ear uses hearing, when the player moves into close proximity of Ear he will know the location of the player, and will be able to move towards him. Eyeball uses sight, whichever way Eyeball is facing he is able to see 7 tiles in front of himself, which spreads out in a triangular fashion (see fig 5). When the player enters Eyeballs sight field Eyeball is able to track down the player.



*Figure 5 & 6*. Eyeball (grey circle) with his triangular line of sight (light grey tiles). A text file holding the statistics of the game; how many games have been played and who has won. Games played, player wins, nose wins, ear wins, and eyeball wins (in order from top to bottom).

# The Applied Intelligence

I have managed to implement multiple different forms of applied intelligence into my program, ranging from the different ways the agents track the player, to the different path finding algorithms that are used to move to the player.

## All Agents

All agents have been implemented with a state system, where they can only be in one state at a time and will change between the states depending on the situation. I have only implemented two states, the patrol state, and the chase state. While in the patrol state all of the agents will move around the maze randomly, they can also stand still for a turn. While in the chase state each agent uses different their unique path finding algorithm in an attempt to find the player.

## Nose

Nose is one of the enemy agents that has been implemented into my game. Using the player’s scent he is able to find and track the player. Nose uses an algorithm similar to that of Dijkstra’s, where Dijkstra will look for the most valuable move (Mostly used for travel algorithms, trying to find the cheapest route), Nose will look for the next most valuable move.

Each tile in the maze will have a different level of scent, most tiles will have a value of 0, but the tiles that the player has recently visited will have a larger value. Nose looks for these “smelly” tiles and once he moves onto one, he looks at the tiles around him (north, south east & west) in an attempt to find the next smelliest tile. Just like Dijkstra’s algorithm, Nose is looking for the next most valuable move in an attempt to reach his goal, the player.

The player’s scent is stored in a list (smellArray) that gets updated every time the player moves. The position the player was last standing on gets appended to the list of smelly tiles, along with the strength of the smell (generally 10). After the new value has been added to the list the all of the values in the list are reduced by one, the weakest smell will then be reduced to “0” and will be removed from the list.

At the start of each update the agent checks its position against the whole of the list of smelly tiles, if the agent is on one of smelly tiles he enters the chase state, if not he stays in the patrol state. In the patrol state the agent will move around randomly.

While in the chase state Nose will look at the tiles north, south, east and west of himself. Nose keeps track of the stink value of the tile he is currently standing on, and then assesses the stickiness of the tiles adjacent. Using a loop from 0 to 4, and by using a “Compass” list (where Compass = ([North, South, East, West)), Nose loops through the adjacent tiles, when a tile has a higher stink value than his tile that tile becomes his next move and his personal stink value is updated to the new stink value.

## Ear

Ear is another enemy agent that I have implemented into my game. Ear uses the players sound in an attempt to catch the player. Ear uses a greedy best-first search algorithm that uses a simple heuristic generating function to determine the tile which will take it closest to its goal. Greedy best-first search algorithms are generally the fastest in simple searches but become increasingly slow when dealing with large graphs of data.

A ring of sound is generated around the player every time the player moves. The positions are created by using two for loops between -4 and 5, in the x and y directions. These positions are then added to a list (soundArray) which simply contains the position coordinates of the tiles which will contain the sound. The two loops create a square block of sound so some of the array values are .pop()’d to make it circular!

At the start of each update the agent checks its current position against the list of sound tiles, if the agent is on one of the sound tiles he enters the chase state, otherwise he will stay in the patrol state. In the patrol state the agent will move around randomly.

While in the chase state Ear will create a dictionary which will hold all of his possible moves and the value of each move. Just like Nose, Ear will loop through the compass checking each of his adjacent tiles, by using a simple heuristic function, which returns the distance between two tiles, a value is given to each new possible move. After Ear looks through all of the possible moves he can make he compares the value of each move and decides which move would take him closest to his goal, the player!

## Eyeball

Eyeball is the third and final agent that has been implemented in my game. Eyeball uses an artificial sense of sight in an attempt to try and locate the player, once the player has been located Eyeball uses an A\* path finding algorithm to determine the best possible route towards the player, taking into consideration all of the walls in the maze. A\* search algorithms are generally the best type of search algorithm to implement, they’re much more reliable than greedy best-first search algorithms but in some cases can be slower.

After every move that Eyeball makes a field of view is generated, creating a triangular array of tiles that Eyeball can now see, and acknowledge the contents of (see figure 5). The field of view is created in the lineOfSight() function that is defined in the Eyeball class, and uses a simple loop to generate the positions of the field of view, appending them into an array list. Once the field of view has been generated it needs to be fixed so that Eyeball is unable to look through the walls of the maze. I have managed to do this by using a lot of different logical statements, which take into consideration the way Eyeball is facing, where the walls of the maze are, and where Eyeball can currently see.

At the start of each update call Eyeball checks the player’s current location against his current location and field of view, if the player is in Eyeball’s field of view he enters the chase state, otherwise he will stay in the patrol state.

While in the chase state Eyeball will use his A\* pathfinding algorithm to determine the best possible route towards the player, he uses a heuristic value for each possible move to make sure that he takes the fastest route.

The A\* algorithm that I have found, and adapted to work for my game uses a priority queue and two dictionaries. The priority queue holds all of the different moves that the agent can make, along with a heuristic for the value of the move, the most valuable moves are the ones that bring the agent closer to the player. The two dictionaries consist of one that holds where the agent has just come from, allowing it to compare the next move to its previous move, as well as a dictionary to hold the cost of the next move (in other mazes I would have implemented tiles which would take up two movements instead of one, meaning some routes would take longer!).

Eyeball checks all four compass directions, adding them to temporary variables which are used to compare the value of each move. If the next move isn’t already in the moves that Eyeball has already checked, and the move is cheaper than the previous the move will continue on to be examined. If the next move doesn’t move the agent into a wall then it is added to the priority queue and the dictionaries are updated. Once all of the routes to the player have been analysed the list which held all of the moves that the Eyeball has taken to get to the player is reverse engineered to produce the path it should take to the agent. The next movement in the list of moves is then used to update Eyeball’s position.

## Spatial Awareness

This may seem simple, but I consider it as some form of applied intelligence. Only one of the three enemy agents has spatial awareness implemented into its pathfinding algorithm. Eyeball is the only agent which considers walls as part of an obstacle, creating routes around the walls in order to catch the player. When playing you will notice that nose will walk into walls if a wall is between him and the player!

# Analysis of feedback

I didn’t really receive much feedback at all, all that the supervisor did was look at the game and acknowledge the artificial intelligence. Hopefully you will be able to play the game a few times and properly analyse the AI.

# Findings

I previously stated that I added a score tracker stored in a text field which keeps track of which agent is winning the most games, or if the players are winning more. I have received some feedback from the people that I have asked to play my game, as well as some personal analysis that the outcome of the game greatly depends on where the agents spawn (as they are spawned into the world with random coordinates), as well as the layout of the maze.

By looking at the statistics text file and all of the results it has gathered, I have concluded that Eyeball is the most efficient at finding and catching the players. Ear and Nose are generally waiting for input from the player whereas Eyeball can locate the player from far off, giving him an advantage. I also believe Eyeball’s path finding algorithm to be far superior to the others when chasing a player.

# References

[1] - Redblobgames.com,. 'Introduction To A\*'. N.p., 2015. Web. 23 May 2015. (Research)

[2] - Interactivepython.org,. 'Shortest Path Problems — Problem Solving With Algorithms And Data Structures'. N.p., 2015. Web. 24 May 2015. (Research)

[3] - Mechahuggermr.tripod.com,. 'The "A.I.: Artificial Intelligence" Fanfiction Online Anthology'. N.p., 2015. Web. 24 May 2015. (Image)