# CI342 - Applied Intelligent Systems

## Shrines – Final Report

Name: Jak Hall

Student Number: 15826673

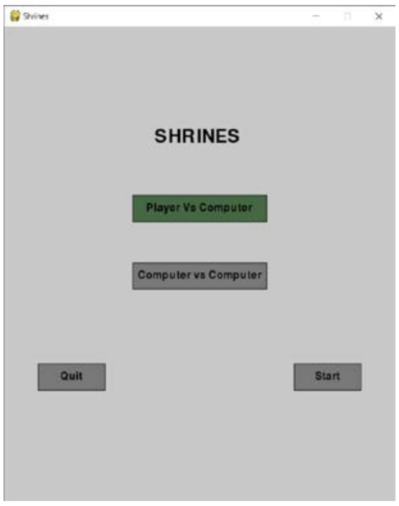
# Contents

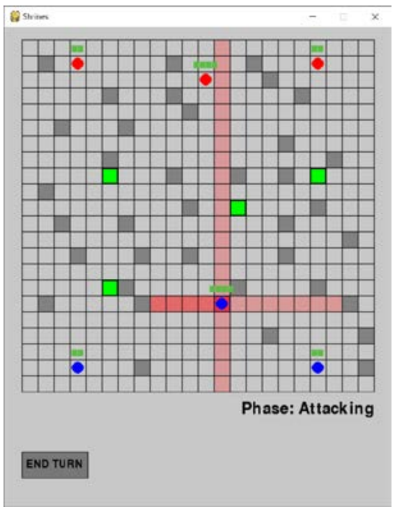
Features ..... 3
Implementation ..... 4
Testing ..... 7
Reflection ..... 9
Summary ..... 9
Literature Review ..... 10

# Features

Shrines is a turn based strategy the game/system that functions in the following way:

- Players take it in turn to move/capture shrines and attack.
- Modes include 'player vs computer' as well as 'computer vs computer'
- A shrine is capturable by a unit if, the unit is in range of a shrine and the shrine Is not being defended.
- A shrine is being defended if a unit of shrines colour is within it range.
- The game ends when a team has eliminated all enemy units or captured all the shrines.
- Implements A* pathfinding for navigation and movement restrictions.
- Uses Decision trees for AI movement, capturing and attacking.



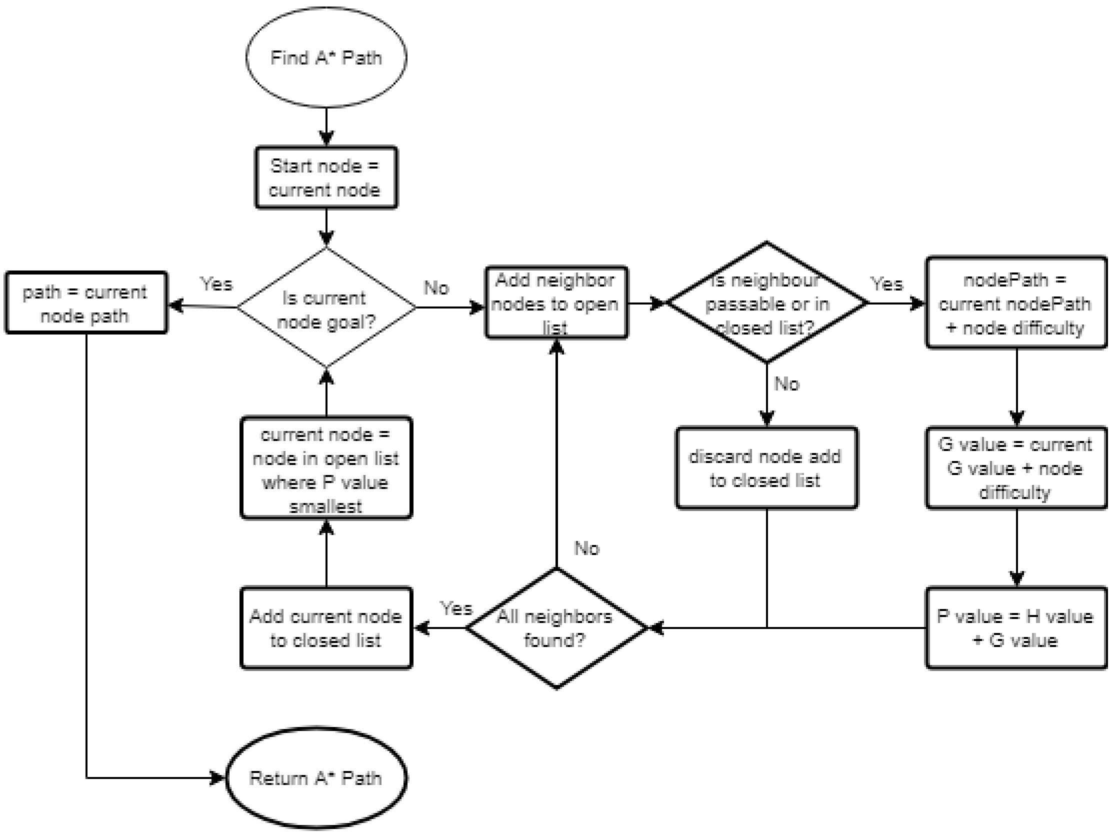


# Implementation

To create my AI game, I decided to use python. It is a language I haven't had much practise with, however I know it is popular with AI due to its rapid development time. This will give me good practise for future projects. Along with python I will be using the 'Pygame' library. With this library I will be able to easily create graphical objects and game elements whilst focusing on the AI parts of my system.

Since a lot of my system requires pathfinding, I created a grid of tiles. Each tile is an object within a two-dimensional array – the tile can be occupied by an entity such as a player unit, shrine or obstacle. The array will be used to store the current state of the game board, as well as for calculating the paths between the tiles.

After considering a number of different algorithms for pathfinding a decided to go with A* search. It is not the quickest in terms of O(n) however in the average case it will be better than alternatives such as Dijkstra's algorithm. Additionally, my research suggests it is the most popular option for game pathfinding in most cases due to its performance. I created the below flowchart for A* search to guide whilst implementing the system.


Find A* Path
Start node = current node
Is current node goal?
Yes
path = current node path
No
Add neighbor nodes to open list
Is neighbour passable or in closed list?
Yes
nodePath = current nodePath + node difficulty
No
discard node add to closed list
G value = current G value + node difficulty
P value = H value + G value
All neighbors found?
No
Yes
Add current node to closed list
current node = node in open list where P value smallest
Return A* Path

To save on computation time, the heuristic value for each tile can be pre-calculated when the game is first run, since the tile positions do not alter and the value does not consider if tiles are passable, it doesn't matter if this changes in the future. The Heuristic value can be stored as an array for each tile and reflects the distance between each all other tiles. I implemented two methods for calculating distances, since each will be more practical for different applications. The two distance formulas I went with can be seen below. By default, the H value uses Manhattan distance since units cannot move diagonally. However, this could be implemented easily in the future.

**Manhattan =** $|\mathbf{p1} - \mathbf{p2}| + |\mathbf{q1} - \mathbf{q2}|$

**Euclidean =** $\sqrt{(\boldsymbol{p1} - \boldsymbol{p2})^2 - (\boldsymbol{q1} - \boldsymbol{q2})^2}$

I implemented a game loop which allows each entity in the system to be updated continuously whilst the game is running. How and what elements are updated depends on what phase the game is currently in. The phases dictate what tasks the user and AI are currently able to perform, this makes it easier to separate operations and keep track of which players turn it is.

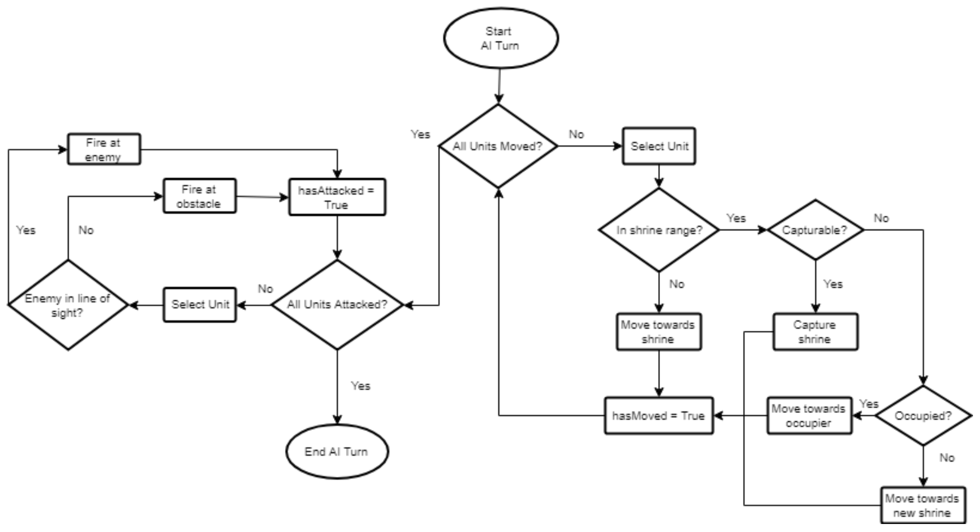
The first phase will be "moving" which the player has two options with each of their units; if they are within range of a shrine they can attempt to capture it or they can move their unit to a new location (within a predetermined range) giving them better strategic position to eliminate opposing units or capture a shrine in the subsequent turns.

When a user clicks on a shrine they can see an area highlighted in green, this is the capture zone for that shrine. This is dictated by which tiles are within a predetermined range and calculated using Euclidean distanced. A unit can spend their move capturing a shrine if they are in range of it, the shrine is neutral or belongs to the enemy and an enemy unit is not also in range. (they must eliminate the defender first before capturing). Once a shrine has been captured, its owner will change to the team of the capturing unit.

If the unit is not in range of a shrine, or the shrine is being defended then it's likely the player will desire to move the unit. When they select the unit, a highlighted area will appear showing all tiles that are in moveable range. This is calculated as a pre-determined number of tiles the unit is allowed to move, this is then compared to the A* search path of each tile around the unit to see if they can reach that tile in the given number of steps. To increase computation time, only tiles that are within a given Manhattan distance are considered as they are the furthest they would be able to reach without any obstacles.

Once all units have moved or captured a shrine, the player can enter the attack phase. This will allow them to fire projectiles and cause damage to enemy units. Units can fire on in a linear direction from the units current tile. The highlighted area that is displayed when a unit is clicked in this phase shows the predicted paths of the projectile depending on which direction the unit shoots. This is calculated by getting each tiles neighbour tile in the selected direction until it hits tile that is impassable. The projectile will travel in the selected direction and destroy itself when it hits the impassable tile or the board boundary. When a unit or obstacle is hit, its health will be reduced by the determined damage of the projectile, if its health is now less than or equal to 0 then the unit/obstacle will be removed.

Once all units have used their attack the player must end their turn. This will begin the last phase which is the enemies turn. The enemy player is always the AI therefore it can be automated; however, the AI will go through both the move and attack phase as the human player would, the system will wait for each unit to move and attack before moving onto the next decision. The decision tree/flow can be seen below, it demonstrates how the AI operates to complete a turn.



The AI will prioritize moving towards the nearest capturable shrine if the unit is not already in range of one, unless it has already selected a unit to move towards that shrine, then it will choose another to spread it resources. If they are trying to capture a shrine and it is currently being defended then the unit will attempt to seek out the defending unit and eliminating it. In both cases the unit will create an ideal path towards the goal tile, if they cannot reach the goal tile in the current turn due to range limit, then they will move to the tile that is furthest in the ideal path that can be reached. This means each turn the path will change depending on the current state of the board.

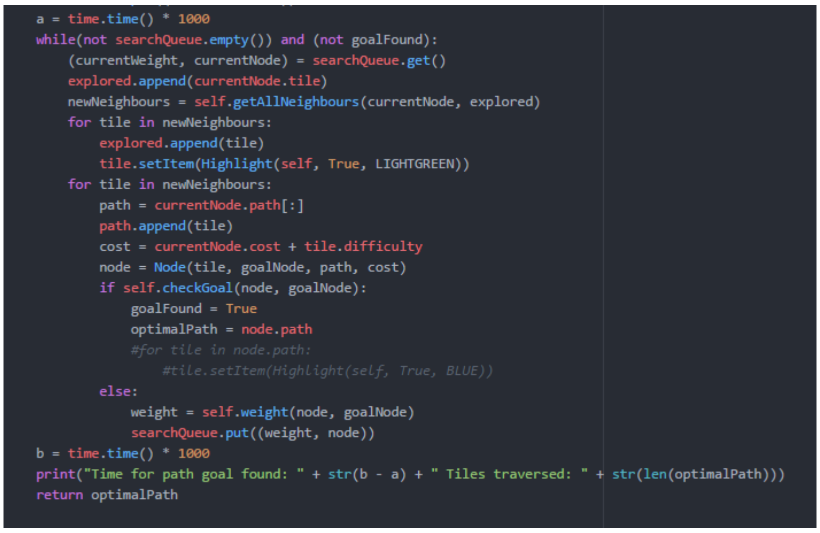
For attacking the unit will locate all directions in which an enemy unit is intercepting, then chooses the unit which is closes to the unit since this is the most threatening enemy to the units own safety. Once all units have attacked the enemy must end its turn. This will end the phase and start the cycle again, if the AI is playing another AI then the phase will immediately go back to the 'enemy' phase however the current team will be the opposing AI player. This means they will continuously alternate until a winner is found.

This is determined when the board only contains shrines occupied by one of the players, or all of one players units have been eliminated and they have no other moves. The winner will be selected as the player in possession of the shrines or remaining units.

## Testing

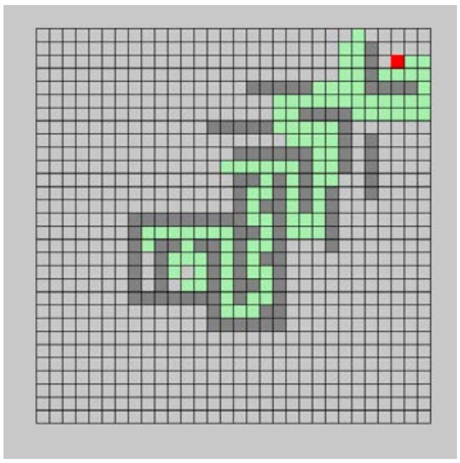
### Method

The main performance test for my pathfinding was done by measuring the time it taken to execute each algorithm. I tested an implementation of dijkstras, depth-first and A* search algorithms to make a decision on which to use in my final AI design. To do this I recorded the time in milliseconds before and after the path had been found by the algorithm.

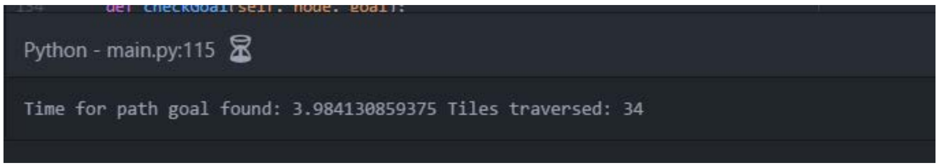


```python
a = time.time() * 1000
while(not searchQueue.empty()) and (not goalFound):
    (currentWeight, currentNode) = searchQueue.get()
    explored.append(currentNode.tile)
    newNeighbours = self.getAllNeighbours(currentNode, explored)
    for tile in newNeighbours:
        explored.append(tile)
        tile.setItem(Highlight(self, True, LIGHTGREEN))
    for tile in newNeighbours:
        path = currentNode.path[:]
        path.append(tile)
        cost = currentNode.cost + tile.difficulty
        node = Node(tile, goalNode, path, cost)
        if self.checkGoal(node, goalNode):
            goalFound = True
            optimalPath = node.path
            #for tile in node.path:
                #tile.setItem(Highlight(self, True, BLUE))
        else:
            weight = self.weight(node, goalNode)
            searchQueue.put((weight, node))
b = time.time() * 1000
print("Time for path goal found: " + str(b - a) + " Tiles traversed: " + str(len(optimalPath)))
return optimalPath
```

To give each method a change I tried three different paths for each at varying lengths and obstacle density. An example of one of the paths can be seen below along with all searched tiles highlighted in green.



The results of each are printed as shown below, the time is the milliseconds to find the goal path and 'tiles traversed' is the how many tiles the found path takes. I can then compare it to the results of all the other two algorithms.



```
Python - main.py:115

Time for path goal found: 3.984130859375 Tiles traversed: 34
```

## Results

### Path One

Description: Long distance from goal node, many obstacles in path.

| Algorithm | Time (m/s) | Tiles Traversed |
|---|---|---|
| Dijkstras | 7.39293 | 34 |
| Depth-First | 3.03314 | 40 |
| A* Search | 3.98413 | 34 |

### Path Two

Description: Short distance from goal node, few obstacles in path.

| Algorithm | Time (m/s) | Tiles Traversed |
|---|---|---|
| Dijkstras | 0.90342 | 9 |
| Depth-First | 0.78222 | 9 |
| A* Search | 0.98403 | 9 |

### Path Three

Description: Medium distance from goal node, medium number of obstacle.

| Algorithm | Time (m/s) | Tiles Traversed |
|---|---|---|
| Dijkstras | 1.69833 | 22 |
| Depth-First | 1.73489 | 29 |
| A* Search | 1.64003 | 22 |

## Evaluation

From the test results I recorded I could use this data to compare the performance of algorithm. In general, Depth-First search took less time however returned with a less than optimal path. This is because it will naively select the first path it finds which may not be shortest route. Dijkstra's search is different in that it will calculate every possible path and then choose the shortest – This finds the shortest path every time, however is far more time consuming the more possible optional routes are available.

A* search takes the best of both worlds in this situation, it will return the first path it finds. But it will ensure that the current route is always the best path (as well as considers heuristic value unlike breadth-first search). That means the route it returns is also the optimal path, as this is reflected in the results I have obtained; my AI will use A* search for all pathfinding. However, if results change in the future, or new algorithms are available, the system is very extendable to allow for new search methods.

## Reflection

The system I have created completes my primary requirements in building a turn-based strategy game which allows a human user to play against or observe the computer. It features almost instantaneous and effective pathfinding, as well as intelligent decision making for complex automated moves that act to challenge the user.

I had however proposed a card system that would allow for random affects to be applied to units, the random element would improve the AI's dynamic functionality. It could have also included automated enhancement, by measuring the success of the AI over each turn, then cross validating it against the optimal/test training data. This might have provided a greater challenge, learning strengths and weaknesses of the player overtime and generate a more unique experience each game.

Given more time a few things that I may add to improve the functionality of my AI would be:

- Additional search algorithms to compare different paths.
- Random card generation which can be applied to affect units – testing the AIs performance with random elements.
- Implement linear regression to predict possible player unit moves.
- Improved aiming system, allowing units to fire at angles. Could implement fuzzy logic for optimal targeting.
- A neural network system that would improve decision making over time, by scoring resulting moves and comparing them to optimal outcomes.

## Summary

It was challenging to building the AI system. Before starting the project, I was unfamiliar with python and game design techniques which made it difficult to learn how to best generate the environment to implement and test my AI. However, I became better at using the language over the duration of the project and it gave me a lot of experience with building such system.

I was also able to learn a lot about AI decision making due to the functions and algorithms that I applied to complete the requirements of my system. I particularly learnt a lot about the A* search algorithm and how it can be utilized for optimal pathfinding as well as how decision trees can be used to improve the functionality of an intelligent system. As intelligent systems are becoming used for so many purposes in the IT industry, I tend on continuing to research AI algorithms and techniques that I can use to improve my skills as a programmer in the future.

# Literature Review

NILSSON, N. J.

**1. Principles of artificial intelligence**

**In-text:** (Nilsson, 1986)

**Bibliography:** Nilsson, N. (1986). *Principles of artificial intelligence*. 1st ed. Los Altos Calif.: M. Kaufmann.

**Review:** A very helpful book for AI techniques in general, in my research I used the resource primarily for its guidance on A* star search algorithm and how it can be best implemented into a system.

KAMIŃSKI, B., JAKUBCZYK, M. AND SZUFEL, P.

**2. A framework for sensitivity analysis of decision trees**

**In-text:** (Kamiński, Jakubczyk and Szufel, 2017)

**Bibliography:** Kamiński, B., Jakubczyk, M. and Szufel, P. (2017). A framework for sensitivity analysis of decision trees. *Central European Journal of Operations Research*, 26(1), pp.135-159.

**Review:** An extensive journal on decision trees, addressing strategies to maximize result values as well as probabilistic analysis and self-reflection that can be used to improve the system overtime.

JAMES, G., WITTEN, D., HASTIE, T. AND TIBSHIRANI, R.

**3. An introduction to statistical learning**

**In-text:** (James et al., n.d.)

**Bibliography:** James, G., Witten, D., Hastie, T. and Tibshirani, R. (n.d.). *An introduction to statistical learning*.

**Review:** A great book detailing mathematical side of machine learning and pattern recognition, very technical but great source of information. I used it for my research and possible improvements to my system.