3CB113 Artificial Intelligence for Games

Daniel Comerford

Student Number: 169079462

Git Repository: https://github.com/dtocomerford/ai\_in\_games.git

Part 1.1: Introduction

In this report I will discuss the game which I created to exhibit a form of artificial intelligence. The game was written in C# programming language in the Unity game engine. The game that I based my game on was Pac-Man (1980). I chose this game as a template due to its simple game objectives and interesting pathfinding potential. The game involves a player in a maze with multiple enemies chasing after them. The objective of the game is to evade the enemies while collecting all the fruit in the level. Once all the collectables are attained the level is complete. The game requires AI agents which will chase the player through the maze, run away from the player and roam the maze with no objective. The plan is to implement the A\* (A Star) algorithm to find the shortest path from the enemy ghosts to the player. Aside from the pathfinding algorithm, the other type of AI in my game will be the use of a state machine. The state machine will be used to control the ghosts in the game. The ghosts in the original Pac-Man have multiple states which dictate the way they behave, and I aim to emulate this with my own state machine.

I chose these types of AI systems for a number of reasons. Firstly, the A\* algorithm was chosen for its efficient pathfinding capabilities. I needed an algorithm to implement to map the enemies movements through the maze. I wanted to do this without the help of Unity’s NavMesh system. I chose A\* because I wanted the game's enemies to be intelligent and difficult to beat. I felt like it was a valid option with the time available for the project. I had also been reading about the algorithm in my own time so I had prior knowledge of what was required to implement it correctly. The A\* algorithm is efficient when run on a map size such as in my game. Triple A games use A\* on much larger maps so I knew that the performance of running A\* would not be a problem. I chose to use the state machine due to the fact that the enemies required different behaviours at different times during gameplay. For example, the enemies need to chase, scatter and flee so it made sense to have a state machine which accommodates all of those behaviours. It also allowed me to cut out unnecessary code and duplicate code. The state machine is less complex than the pathfinding algorithm so time constraints and feasibility were not an issue. If A\* was achievable in the time frame available then so was the implementation of a state machine.

*Part 1.2: Implementation Report*

In this section of the report I will discuss how I implemented the AI system and describe the classes along with their functions. First of all I wanted the AI to be in its own class not attached to the enemy objects themselves, this way I could reuse the AI for multiple objects and uses. Doing it this way also isolated the movement of the enemy to the object of the enemy. This modular approach helped me understand the functionality of each piece of the game. The first step to create the enemies AI was to create a class called FindPath which has functions that calculates the enemies path. The class has a function which receives two points as arguments which would then plot the pathway between these points, this function is called *AStarSearch*. This function returns a path that will then be given to the enemy GameObjects, the path is of the type List. In the table below I will describe the classes I implemented using class diagrams and source code, I will also examine how the AI is accessed through the code structure.

| Class Diagrams | |
| --- | --- |
| The FindPath class is where the A\* algorithm is called from.  Here are the properties of the class; It contains three list types which store the openset, closedset and path nodes. The openset is a list of nodes which are yet to be visited by the algorithm, the closedset is a list of nodes that have been visited by the algorithm but have been discarded for a better node. There are two Node type variables called start and goal which store the Node script for the beginning and end of the path. The class contains a Walls type variable which is a class I created to store the GameObjects for the walls in the maze, FindPath takes in a Walls object to plot the best path using the positions in the maze. The final variable in FindPath is a two dimensional array of the type Node, this stores all the Node scripts in the formation they appear in in the game world. |  |
| Here are the methods of the FindPath class;  **void Setup(Walls, walls)**  The first method is the *Setup* method which is called from the GameManager class, it takes a Walls class object as an argument. From the Walls object which contains a two dimensional array the *Setup* function uses a nested for loop to go through each GameObject in the two dimensional array storing the maze to get the Node script that is attached to that GameObject.  //array.size == this.wallsInfo.nodes.GetLength(0)  //Couldn’t fit the code in the grid without chopping it off so I’ve abbreviated  public void Setup(Walls walls)  {  this.wallsInfo = walls;  this.workableMap = new Node[array.size, array.size];  for (int i = 0; i < this.wallsInfo.nodes.GetLength(0); i++)  {  for (int j = 0; j < this.wallsInfo.nodes.GetLength(0); j++)  {  Node nodeScript = this.wallsInfo.nodes[i, j].GetComponent<Node>();  test[i, j] = nodeScript;  }  }  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////  **double ReturnDistance(Node current, Node neighbour)**  The function *ReturnDistance* returns a double value of the distance between two nodes  public double ReturnDistance(Node current, Node neighbour)  {  double distance;  distance = Math.Sqrt(Math.Pow(current.xPosition - neighbour.xPosition, 2) + . Math.Pow(current.zPosition - neighbour.zPosition, 2));  return distance;  }  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////  **void ReturnPath(Node, goalNode)**  This function takes in a single Node object and using a while loop it traverses through the Nodes in the path that the algorithm has found, similar to how a linked list is navigated.  public void ReturnPath(Node goalNode)  {  Node currentNode = goalNode;  while (currentNode != start)  {  path.Add(currentNode);  currentNode = currentNode.previous;  }  }  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////  **List<Node> AStarSearch(int startX, int startZ, int goalX, int goalZ)**  This function;   * Takes in four integers as arguments which it uses as coordinates in the two dimensional array to get the start Node and goal Node for the path. * The start Node is then added to the openset, a while loop then starts which will run while the openset is not empty. * On the first loop when there is only a single Node in the openset the variable currentNode will be set to that Node, later when there are multiple Nodes to consider in the openset currentNode will be set to the Node with the best fValue. The fValue is the Nodes distance from start value + distance from goal value. * currentNode which is the best Node at this point in the search is removed from the openset and added to the closedset. * An if statement then checks to see if the currentNode is the goal, if it is then the *ReturnPathFunction* is called and the *AStartSearch* function concludes. If currentNode is not the goal the search continues. * A for loop then searches through every neighbour of the currentNode, if the neighbour is in the closedset the loop skips to the next iteration. A calculation is done using the *ReturnDistance* function to get the distance from the neighbour Node to the start Node going via the currentNode. If the sum of this calculation is lower than the gValue of the neighbour that is currently being looked at then that neighbour has its gValue updated to the new figure, its hValue is assigned and the variable that all Nodes have called previous is set to the currentNode. * One final if statement that checks to see if the neighbour is in the openset or not, if it isn’t then the neighbour is added. * The loop then repeats * Once the path is found the nodes are returned as a List of type Node   public List<Node> AStarSearch(int startX, int startZ, int goalX, int goalZ)  {  ResetPath();  //Set the start and goal nodes  start = workableMap[startZ, startX];  goal = workableMap[goalZ, goalX];    //Add starting node to list  openSet.Add(start);  //Loop while the open set is not empty  while (openSet.Count > 0)  {  Node currentNode = openSet[0];  Node nodeScriptOfCurrent = currentNode.GetComponent<Node>();  //Getting the best node from the openset, first loop through there's only one to choose from which is the start node  for (int i = 1; i < openSet.Count; i++)  {  //if two nodes have the same F score we pick the one which is closer to the goal, with the better H score  if (openSet[i].fValue < currentNode.fValue || openSet[i].fValue == currentNode.fValue && openSet[i].hValue < currentNode.hValue)  {  currentNode = openSet[i];  //path.Add(currentNode);  }  }  //remove the current node from the open set as we're now done with it and add current node to the closed set  openSet.Remove(currentNode);  closedSet.Add(currentNode);  path.Add(currentNode);    //If it's the goal you know what to do  if (currentNode == goal)  {  ReturnPath(currentNode);  return path;  }  //Loop through every neighbour of the current node  for (int i = 0; i < currentNode.neighbours.Count; i++)  {  Node neighbour = currentNode.neighbours[i];  //Check the closed list to see if the neighbour is in that list, if so the loop breaks the current iteration and goes again  if (closedSet.Contains(neighbour))  {  continue;  }  //Calculate the distance from the neighbour to the start  double newMovementCostToNeighbour = currentNode.gValue + ReturnDistance(currentNode, neighbour);  //Check to see if distance from the start to the neighbour through the current node is a shorter distance than the G score of the neighbour currently  //Add check if the neighbour is in the open set  if (newMovementCostToNeighbour < neighbour.gValue || !openSet.Contains(neighbour))  {  neighbour.gValue = newMovementCostToNeighbour;  neighbour.hValue = ReturnDistance(neighbour, goal);  neighbour.previous = currentNode;  if (!openSet.Contains(neighbour))  {  openSet.Add(neighbour);  //neighbour.label = 'O';  }  }  }  }  return path;  }  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////  There are other functions in the FindPath class but they are for debugging.  void DrawMap()  This function would draw the nodes which make up the path from the enemies to the goal.  void ResetMap()  This function is called once a path has been found and returned to the GameManager class. It empties the lists and retsets some variables which will be used again when a new path needs to be found. | |
| This picture shows an early class diagram and how each class connects to each other. | |

| **State Machine** | |
| --- | --- |
| This is the diagram of the state machine which I had initially envisioned for the enemy behaviour. There are three active states which the enemies can exhibit, these behaviours are called chase, flee and scatter. The states would change based on the player's distance to the enemy, if items were collected like the fruit in the original Pac-Man or on a timer. |  |
| In the game I made this is the state machine which I implemented. There are three states called chase, scatter and scared. The enemies begin in chase and transition to the scatter state after a set amount of time. From the scatter state the enemies will transition back to the chase state after a set amount of time too. The enemies will transition to the scared state if the player collects a cherry. They will return to the chase state once they have reached the tile furthest away from the player. They will then continue to switch between chase and scatter states, unless another cherry is collected. |  |
| **State Machine Code** | |
| Void GhostBehaviour   * This function exists on each enemy GameObject. * The function takes in the players GameObject. * A switch statement is used to change between enemy behaviour. * Transitions between the chase and scatter state are triggered by a timer, the states are set to change quickly this is from the testing phase where the behaviour was being monitored. * In the scared state the function *GetScaredNode(player)* is called and the player GameObject is passed in. The function then calculates the node furthest away from the player and returns that node. There are four nodes which are considered to make this a more efficient search and they are the four corner nodes. A path is then calculated to that node. * The scatter function uses a similar method, *GetRandomNode()* generates a random number and that number is used as the index for the list of all walkable nodes on the map. A random node is returned and from that a path is calculated.   public void GhostBehaviour(GameObject player)  {  timer++;  this.xPosition = (int)this.gameObject.transform.position.x;  this.zPosition = (int)this.gameObject.transform.position.z;    //Check to see if the player is in the game  //If not then the timer is frozen so states don't change and the enemy will continue in scatter mode  if (GameManager.playerInGame == true)  {  pathToFollow = findPath.AStarSearch((int)this.gameObject.transform.position.x, (int)this.gameObject.transform.position.z, targetX, targetZ);  }  else  {  this.enemyState.currentState = EnemyState.State.Scatter;  }  if (GameManager.cherryEaten == true)  {  this.enemyState.currentState = EnemyState.State.Scared;  GameManager.cherryEaten = false;  }  //State machine which dictates the enemies behaviour  switch (this.enemyState.currentState)  {  **case EnemyState.State.Chase:**  Debug.Log("Chase");  Run(pathToFollow);  pathToFollow.Clear();  if (timer > 1000)  {  timer = 0;  this.enemyState.currentState = EnemyState.State.Scatter;  }  break;  **case EnemyState.State.Scatter:**  Debug.Log("Scatter");  Node target = GetRandomNode();  pathToFollow = findPath.AStarSearch((int)this.xPosition, (int)this.zPosition, target.xPosition, target.zPosition);  Run(pathToFollow);    if (timer > 1000)  {  timer = 0;  this.enemyState.currentState = EnemyState.State.Chase;  }  break;  **case EnemyState.State.Scared:**  Debug.Log("Scared");  target = GetScaredNode(player);  pathToFollow = findPath.AStarSearch((int)this.xPosition, (int)this.zPosition, target.xPosition, target.zPosition);  Run(pathToFollow);  if (this.transform.position.x == target.xPosition && this.transform.position.z == target.zPosition)  {  this.enemyState.currentState = EnemyState.State.Chase;  }  break;  default:  Debug.Log("Idle");  break;  }  } | |

The design decisions that I made took into consideration the time available to me and the complexities of the individual task. These decisions were subject to change if the development of the game was taking longer than anticipated. The game type was decided upon due to the fact that I wanted to implement the A\* algorithm and I felt that a maze game was one of the best ways to show off the algorithm. I then thought that Pac-Man would be the ideal game to replicate, this was for a couple of reasons. Pac-Man is over 40 years old therefore it has been analysed and talked about more than modern games, this increases the likelihood that literature has been written and made available to the public regarding its source code and AI. If I ran into any problems I would have ample support to fix any bugs. Another reason for choosing Pac-Man is that it is one of the most recognisable games in the world so users will be familiar with the behaviour that the game should display. When I started creating the game I had a choice of 2D or 3D, either would have worked well for what I wanted but I have more experience making 3D games so decided I would use Unity’s 3D environment.

I chose the A\* algorithm because of its fame among path finding algorithms and its use in large console games. I also wanted to test my knowledge of A\* by writing it myself to get a better understanding of how it worked. Due to picking A\* I reduced some visual aspects of the game, things like UI, icons and menu features. This happened as a result of issues implementing the algorithm, and therefore time was restricted. I anticipated that this might happen as it was by far the most intricate element of the game. The other AI element of the game is the state machine that I was intrigued by, like the pathfinding algorithm I had never implemented one so I wanted to try something new in this game.

When it came to choosing software to create my game I decided to use the Unity game engine, it is an engine I have used before which allows games to be written in the C# (C Sharp) programming language which is familiar to me. I chose Unity so I could begin development immediately without having to learn a new programming language and game engine. Alongside Unity I used Visual Studio to write the C# code. In a number of classes I use the C# List type which I use to store Nodes, GameObjects and the path for the enemy to follow. This required me to import the System.Collections.Generic library, I imported this so I could create a strongly typed collection, this meant I could have a list of Node type objects or GameObjects.

*Part 1.3: Discussion*

In this section I will discuss the overall result of the AI systems I implemented and how well they accomplish the goals I set out, the limitations of the AI and alternative ways I could have achieved the same results or better results.

The behaviour of my game achieves some of my initial goals but misses out in a few areas. The enemy agent uses the A\* algorithm to plan a route and successfully finds its way to the player. There are two enemy agents which is maybe too few for the size of the map, the challenge is not that great. The state machine also works correctly moving between states and altering the enemy’s behaviour. I had planned for the state to change from the scatter state to the chase state if the player got within a certain distance but I didn’t have time to implement that feature. If the player is eaten the user can respawn back into the game by pressing the ‘R’ key but the enemies state machines be locked into the scatter state where they randomly explore the map.

I had wanted the player to collect power ups which allowed the player to eat the enemy, I created the scared state for this purpose but I didn’t have time to add the feature to eat the enemies. I also wanted the enemies to change colour if they were in the scared state just like in the original Pac-Man so that there is a visual cue that something has changed. The state machine cutbacks are due to time constraints so perhaps I was too adventurous in terms of scope in this project, I spent the majority of the time working on the enemy’s movement and pathfinding along with the player’s movement. Once these features were complete I didn’t have as much time as I had planned for to develop other areas of the game such as the power ups and other states for the enemy.

The AI had some limitations which I noticed during testing, the first of which is the way that the enemy plans and executes its route. The enemy will complete its path even if the player has moved. This is obvious when the player turns at a junction and then the enemy moves past the functions only to return back in the direction it came to then turn at the junction the player went down. This is due to the enemy only receiving a new path once it has completed its current path. On reflection I could have taken different approaches to the game which may have allowed more time to create a better game. I could have used Unity’s NavMesh to navigate the enemy to the player.

While testing the game I also noticed that the enemies jump from time to time. They will be following a path but will jump ahead, the movement isn’t smooth, I think this is from the pathfinding algorithm not finding a complete path before the next frame needs to be drawn. If I found a more efficient way to access the nodes or store the paths this could be reduced.

There are some minor issues which are unrelated to the AI that I implemented for this game. For example there is no way to win the game, there is no user interface to display a score or number of lives. There isn’t a menu or pause menu which would have made the game feel more polished. The player and enemies spawn quite close together in the same positions every time which I would like to alter.

Overall I am happy with the resulting game. There are multiple enemies which can change behaviour between three states using two methods to transition the behaviour. The enemies use the A\* algorithm to find the shortest path and they do so successfully. Another positive is that I have only experienced the game crash a few times during testing. Away from the gameplay I implemented the map and nodes so that the map can be scaled and the game can be run without any further modifications, meaning that my code is dynamic.

| Here is a screenshot of the final game. The two enemies in red and pink due to the material not loading for the second enemy agent. The spheres are the cherries in game which once collected put the enemies into the scared state. |
| --- |
| The cherries spawn on four random nodes which doesn’t prove to be great positioning. A change I would make would be to spawn the cherries on the four corner nodes. That way there is more of the map to traverse to reach the cherries, increasing the challenge |