Autonomous Agents: AI Toolkit Report

Michael Stroughair
Student Number: 12302190

10th April 2017

1 Lab 1

I began by creating my State Machine code from the tutorial code provided. The State Machine as built to contain a generic type of object, and at this point consists of an Init() function, a ChangeState() function and an Update() function. It also contains a single State¡T¿ object, which, again, is designed to take a generic type of object. State is an abstract object, and contains 3 functions that an inheriting object must implement. Each state however is implemented with a static constructor, creating an instance of the state at runtime. This instance of the state is used for each Agent, in opposed to creating a new State each time the Agent needs to change state.

```
public class StateMachine <T> {

   private T agent;
   private State<T> state;
    private State<T> prevState;

   public void Awake () {
      this.state = null;
   }

   public void Init (T agent, State<T> startState) {
      this.agent = agent;
      this.state = startState;
   }

   public void Update ()
    {
      if (this.state != null) this.state.Execute(this.agent);
   }

   public void ChangeState (State<T> nextState) {
      if (this.state != null) this.state.Exit(this.agent);
      this.state = nextState;
```

```
        if (this.state != null) this.state.Enter(this.agent);
    }

    public void EnterGlobalState(GlobalState<T> nextState)
    {
        this.state.Exit(this.agent);
        this.prevState = this.state;
        this.state = nextState;
        this.state.Enter(this.agent);
    }

    public void LeaveGlobalState()
    {
        this.state.Exit(this.agent);
        this.state = this.prevState;
        this.state.Enter(this.agent);
    }
}
```

Within the lab are now 4 different agents, which implement the above objects within their execution.

- **Miner**

  The Miner mines for gold, deposits it in the Bank, and then will also choose to sleep after he has mined a lot of gold. For each location he moves into a new State. In his mining state, each Update() function will see him mine for gold for a while, and then check whether he's mined enough for that cycle. When he has, he will move into the Bank State, in which he goes to the bank and deposits his gold. At the Bank Stage, he then checks if he's deposited enough gold for the day. If he has, he will enter the Sleep State and go home to sleep for a while. Otherwise, he returns to the Mining State and continue working.

- **Bandit**

  The Bandit will lurk between both the Cemetary and the Bandit camp, moving between both after a random period of time has passed. However, every so often, he will instead enter the Rob State and go rob the Bank. This is implemented slightly different to that of the Miner, as the Cemetary State and the CampState are regular states, but the Rob State is a global state, which means that the Bandit will return to the location he was previously after robbing the bank, in opposed to entering the same state each time.

- **Sheriff**

  The Sheriff has a single state that he works within, the Patrol State. During this state, the Sheriff moves between each of the locations on the board, and checks to see whether the Bandit is there. This is achieved by executing the OnArrival() event. When the Bandit recieves this event, if he is in the same location as the Sheriff he has a chance of being able to run away. If he fails he is killed. When he is killed, if he has any gold on him that he has not stashed inside his camp, the Sheriff can recover it, and trigger another event to tell the Miner that his gold has been returned to the bank.

- **Undertaker**

  The Undertaker also works with a single state, in which they stay within the confines of the town waiting to be needed. When the Bandit is killed, he is attached as an object to the Undertaker. When this happens, the Undertaker is then under control of of the movements of the Bandit. He will proceed to carry the body to the cemetary to bury him. When the body has been buried, he is removed from the Undertaker and regains control, at which point he is resurrected and returns to his regular actions.

2 Lab 2

I chose to follow the tutorial found at the following address to complete this particular lab, changing the name of the object TileEngine to GameController, as I later added more functionality to the object. https://gamedevacademy.org/how-to-script-a-2d-tile-map-in-unity3d/

The code allows for an entirely different layout to be created each time the game is launched, and can be extended to allow for the map to change over time, for example, giving the Mine a limited lifespan, and requiring the Miner to create another one somewhere on the map. It could also be used to hold multiple maps, and choose which one to load in real time.

Simply put, a 2D array of TileSprites is created with a mix of mountains and plains, and the unique locations are placed in a unique random spot within the map. Then, each update() call refreshes the tiles on the map. This allowed me to create code that would allow for tiles to be shaded grey, for the subsequent A* code that would be later implemented (this functionality however was never used, due to the fact that overlapping paths would cause the Tile to be unshaded when the first agent passed over it, and before the second agent had done the same. This could be fixed by adding a counter within the Tile, which could cause it to become progressively darker the more agents are due to pass over it).

Within this class, I also added a value called characterMovement. In my Update() function, I also chose to increment another variable called framesPassed. When framesPassed is equal to zero, the agents are able to take a turn of movement. However, any other value will cause the agents to skip their turn. This reduces the movement on screen significantly. I chose to implement the code in this way to make my later sensing code a little easier, since I wanted to be able to move in discrete grid steps rather than continuously. By doing it this way, other options such as the Sheriff's OnArrival() event can be triggered by a physical transform location, and I don't need to worry about the sprites overlapping on the screen since each sprite is drawn within a segment of the tile area (ie, the Bandit is always in the top left of the tile, the Sheriff in the bottom left, the Miner in the top right and the Undertaker in the bottom right).

For drawing each Agent on the board, I created a function called GetGlobalPosition(), which will take the eLocation of the Agent, and find the corresponding tile within the 2D grid in the game controller. This location, ie, the grid location, is then converted into world coordinates, and returned for use as the agent's transform position.

3 Lab 3

I chose to follow another tutorial to help me with the initial parts of this lab. http://blog.two-cats.com/2014/06/a-star-example/

This time however, I needed to alter the code given to me by this tutorial, as I needed more complex data structures for my implementation. On top of this, the basic implementation of this that the tutorial gave me does not factor in differing terrain costs, so I needed to create a function called GetWeightedTraversalCost that changes how easy it is to move to a Tile based on the location depicted in it. For example, a mountain has a x1.5 modifer, the plains had no modifier, and all other tiles had a x1.2 modifier. However, I chose to only use this on the G-value of the Node, and left the H-value to be a simple distance calculation. I did this because sometimes the agents chose to take quite peculiar paths when there were clearly shorter paths they could take, and I determined that this was due to the H value being too dependant on the current square's Tile type.

The structure for my Pathfinding code ended up being simply a Solver class and the Node class, since I determined it was better off that the Solver class worked like a Factory Object, returning a stack of Nodes for a given start and end location. Since I had already created a grid of TileSprites within my GameController, I passed that to the Solver as a constructor object, which I then used to fill my new grid of Nodes each time I gave the function a new start and end point.

The path is created recursively, starting at the start location. The program first closes the current node, then gets a list of all the adjacent nodes and checks whether any of them are untested. If they are, they are added to the list to check later. If there are any nodes that are already open, their previous F value is checked against their current F value. If the current value is lower than the previous value, it is added to the list as well.

This list is then ordered by F-value lowest to highest, and the first node is checked to see whether it is the end location. If it is, the global Node "end" is set as that Node, and the function returns true. If is isn't, the node is passed into the search() function recursively. If an end point isn't found, the next node is checked, and then the next, and so on until every node in the grid has been searched or a path has been found.

When each node is added to the list of adjacent nodes, its parent is set to be the current node. This means that when the end node has been reached, the path of nodes from the start to the finish can be found by recursively adding a node that the agent must reach to the stack, and then adding its parent, and so on. This stack is then returned to the Agent that asked for it.

Since I didn't want agents to really be in a State as such while they are travelling, I chose to implement my movement code so that an agent is unable to access its State Machine while it is moving. Within the DoMovement() function, if an Agent's destination is equal to its location, it will return true, allowing the State Machine work to occur. The location will become equal to the destination when there are no nodes left within the path.

I chose to allow for diagonal movement as it gives Agents the option to do so, even though it may not be faster than going vertical or horizontal. However, the code is written to make this very easy to change, as I would simply need to comment out 4 lines within my GetAdjacentLocations() function.

I also chose to write a function called GetPathLength() which returns the length of the shortest parth between two points. This was written to be integrated later with my sensing functions.

```
public bool DoMovement()
    {
        if (first)
        {
            first = false;
            transform.position = GetGlobalPosition();
            destination = location;
            return true;
        }
        if (dead || !controller.characterMovement)
            return false;
        if (location == destination)
            return true;

        if (path.Count == 0)
        {
            path = controller.pathfinder.solve(GetGridPosition(location),
                GetGridPosition(destination));
        }
        if (path.Count == 0)
            Debug.Log("Crashing...");

        Node nextSquare = path.Pop();
        Vector2 MapSize = controller.MapSize;
        var viewOffsetX = MapSize.x / 2f;
        var viewOffsetY = MapSize.y / 2f;
        var tX = (nextSquare.location.x - viewOffsetX + 0.5f) * 1f;
        var tY = (nextSquare.location.y - viewOffsetY + 0.5f) * 1f;
        transform.position = new Vector2(tX, tY);
        if(path.Count == 0)
        {
            location = destination;
            isMoving = false;
            return true;
        }
        return false;
    }
```

4 Lab 4

I was limited in how I could implement this final lab based on time constraints, but I have functions that implement each sense in a different way than the others. This is not to say that I couldn't have had an event that all agents were able to listen in on, and respond to differently based on whether it was a sight sound or smell type event, but I wrote this code a little more quickly than I would have liked and so was unable to unify all the different ways into a single event.

- **Smell**

  This was implemented by a simple event structure, owned by the Agent class but currently only triggered by the Miner. Upon leaving the mine, the Miner emits an odor, whose strength is random. When one of the agents subscribed to the event is required to handle it, they check their distance to the Miner, and if it is less than the radius that was randomly generated they can respond properly. The Sheriff simply responds with how unimpressed he is upon smelling the Miner, but in the case of the Bandit, this automatically causes him to enter the Rob Global State, to steal the Miner's newly earned gold.

- **Sight**

  Sight is implemented as a function within the GameController, which can be accessed by any of the agents. It is passed the agent calling it, the maximum distance the agent can see, a 3D vector and a reference to a list of 2D vectors. The function then obtains a list of all agents within the program, and runs through them individually, checking if they are within the maximum distance and in the direction specified. If they are, they are added to a list of Agents that will be returned later, and their location is added to the list of locations.

  I have also added code that will convert the calculations to work as a sight bubble if the direction vector passed in is equal to (0,0,0). This is the way that I have it implemented on the Sheriff. If he sees the bandit within a 2 unit radius it will trigger the Showdown event, in which the Sheriff has a chance to kill him.

- **Hearing** Hearing is also implemented within the GameController.However, while it is passed a maximum distance, this distance is calculated by the A* code. Like the Sight code, all agents are pulled from the program, and checked for their proximity to the listening agent. The shortest path is calculated by first calling the pathing function to get the stack of Nodes. This path is then reversed and fed into an array format instead, and the G-value for the Node in the first position is returned. This value of course is the G-value of the final Node in the stack, which is length of the path.

  If the shortest path to them is shorter than the maximum distance, the agent is able to detect them. This code is used with the Bandit. If he notices the Sheriff is within his hearing range, the variable turnsLurking is set to MAX_TURNS_LURKING, which will cause him to move locations during the next cycle of his state machine.

```
//Code triggered when the Miner leaves the Mine
int stench = Random.Range(1, 3);
Agent.TriggerSmellEvent(agent, stench);

//Direction is the direction in which the agent is looking. If its
    set to 0, it means that the agent can see in all directions.
public List<Agent> AgentSee(Agent a, int maxDistance, Vector3
    direction, ref List<Vector2> locations)
{
   List<Agent> agentsOut = new List<Agent>();
   GameObject[] agents = GameObject.FindGameObjectsWithTag("Agent");
   Vector2 agentLocation = a.GetGridPosition();
   foreach (GameObject agent in agents)
   {
      Agent target = agent.GetComponent<Agent>();
      if (target == a)
         continue;
      Vector2 targetLocation = target.GetGridPosition();
      if(direction == new Vector3(0.0f, 0.0f, 0.0f))
      {
         if(Vector2.Distance(agentLocation, targetLocation) <=
             maxDistance)
         {
            locations.Add(targetLocation);
            agentsOut.Add(target);
         }
      }
      else
      {
         Vector3 distance = targetLocation - agentLocation;
         distance = Vector3.ClampMagnitude(distance, 1.0f);
         direction = Vector3.ClampMagnitude(direction, 1.0f);
         if((Vector2.Distance(agentLocation, targetLocation) <=
              maxDistance) && distance == direction)
            {
                locations.Add(targetLocation);
                agentsOut.Add(target);
            }
        }
    }
    return agentsOut;
}

public List<Agent> AgentHear(Agent a, int maxDistance, ref
     List<Vector2> locations)
{
    List<Agent> agentsOut = new List<Agent>();
    GameObject[] agents = GameObject.FindGameObjectsWithTag("Agent");
    Vector2 agentLocation = a.GetGridPosition();
    foreach(GameObject agent in agents)
    {
        Agent target = agent.GetComponent<Agent>();
        if (target == a)
            continue;
        Vector2 targetLocation = target.GetGridPosition();
        if(pathfinder.getPathLength(agentLocation, targetLocation) <
             maxDistance)
            agentsOut.Add(target);
    }
    return agentsOut;
}
```

4.1 Task 5

- **Question 1**

  To begin with, I could very easily change the ability of the Bandit to hear the Sheriff if he is carrying gold at the time, due to perhaps the rattling of it in his pockets.

  The Sheriff's sight could perhaps change based on how long its been since he's been back in town, perhaps taking a quick teabreak in his office before continuing on his rounds.

  The strength of the Miner's smell could increase based on how many times he's been down in the mine since he last slept.

- **Question 2**

  All global weather and time would need to be contained within the Game Controller. I could very easily implement a time system within the Update() function, perhaps based on how many frames have been rendered. With this, I could then decrease vision range incrementally based on the time (midnight being lowest, midday being the highest). In terms of weather, the main two would need to be rain and wind. Obviously sight and smell would be hampered by rain, but sound and smell would increase in range when the wind was blowing in the direction of the recieving agent, and decrease when blowing in the opposite direction. The best way to implement this would probably be by using the Dot product to calculate the angle between the direction of the wind and the direction to the agent from the source of the sense event.

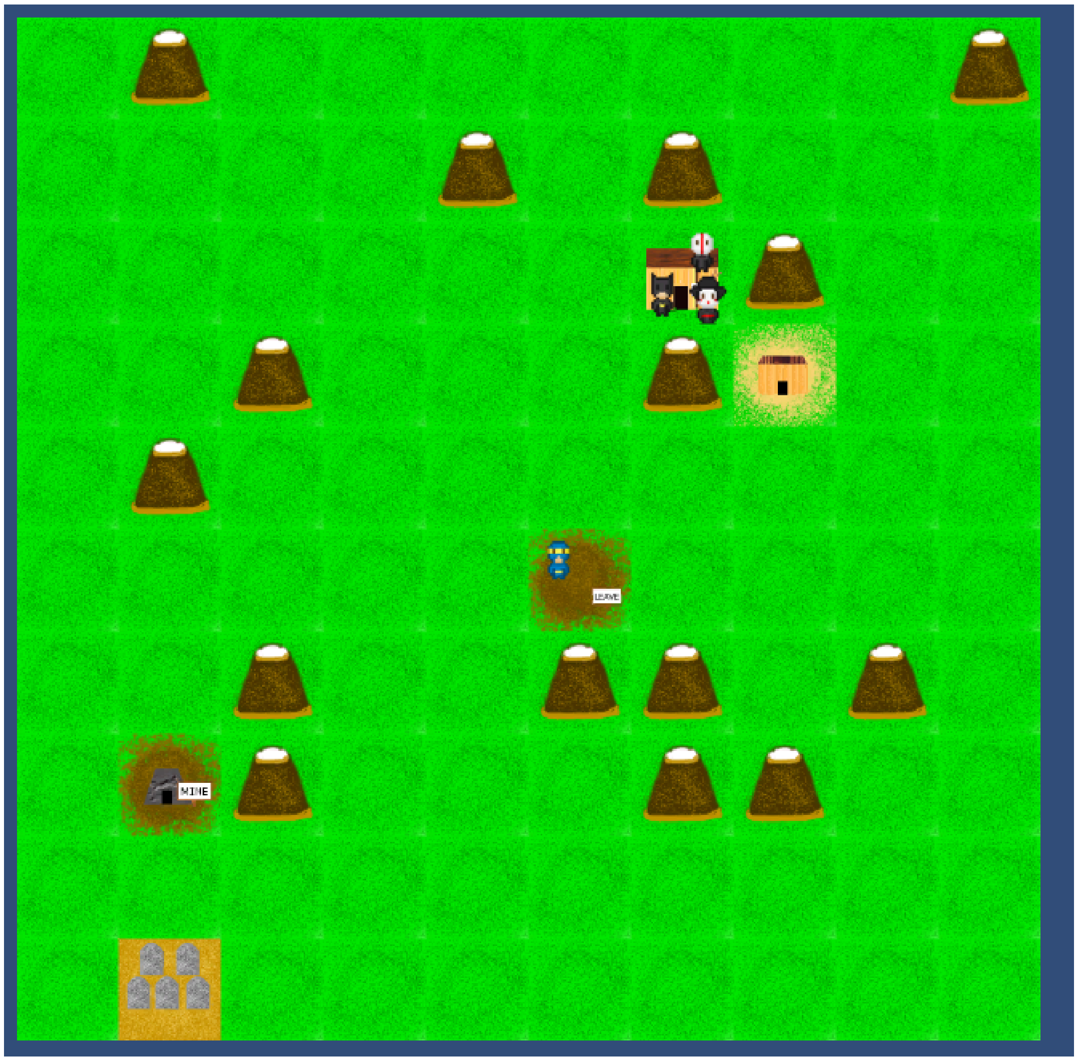

Figure 1: *The Gameboard*