# A* Pathfinding and State Machine Finite State Machine

COMP 452 – AI for Game Developers

April 18, 2024

Keith Mitchell

SID 3178513

# A* Pathfinding Visualizer

## Game Logic and Design

**Game Logic**

I decided to build this out like a paint program where the user could use a brush to place tiles within the map. Once the goal and start are placed, the user can press play to start the visualizer. Since it is more of a visualizer and less a game the logic here is pretty specific to showcasing A* and less about custom game logic. I included a reset button to allow the user to reset the visualizer after a run.

**Game Engine**

The only real engine updates that I did here were to add a sprite class which handles animations through sprite sheets. The rest was carried over from the first assignment. I did make sure to use the custom Vector class more regularly though some changes still need to be made within the GameObject class. I did attempt to make the AStar class reusable for any grid-based tile system that uses the tiling done here.

## Compiling/Game Setup

Before you can run the visualizer you must first ensure that the Java Runtime Environment (JRE) is installed on your system. If you are wanting to compile the program yourself, you will also need the Java Development Kit (JDK) installed.

**Compiling:**
In order to replicate the development environment, I used the IDE IntelliJ. You can load the AStarPathfinding folder as a project. Depending on your compiling environment you may need to target the assets folder as the resource folder. You can usually do this by right clicking the folder in the IDE and marking it as the resources folder, though each IDE may differ. You then can then use the IDE to compile/build/run the game. In IntelliJ, open the AStarPathfinding file (in the game package), and click play (Shift+F10).

You can compile without an IDE. Generally, this will involve navigating to the project folders and using command line prompts to compile all the .java files using the command line javac. This turns the files into .class files which can then be run using the java command line. (AStarPathfinding.java is the main entry point file).

If for some reason the assets cannot be found and you are getting null references, this might be due to miss targeting of the assets folder. The only location needed to adjust will be within the ResourceLoader class. Adjust the folder directory to match your systems directory structure to the location of the assets folder.

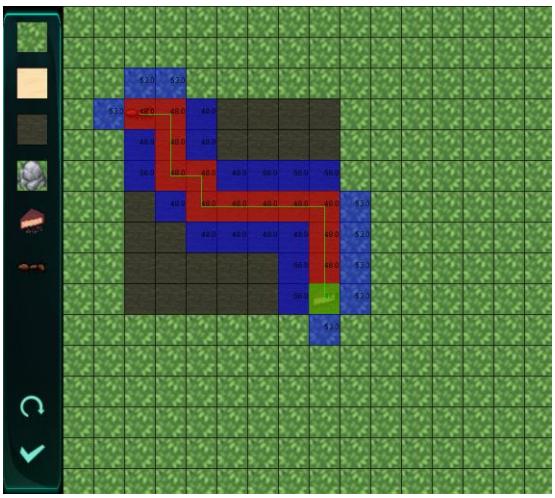
**Running from .jar:**
The .jar file is located in the AStarPathfinding folder. On most systems you can simply double click the .jar file and the game will launch. (Assuming you have installed the JRE).

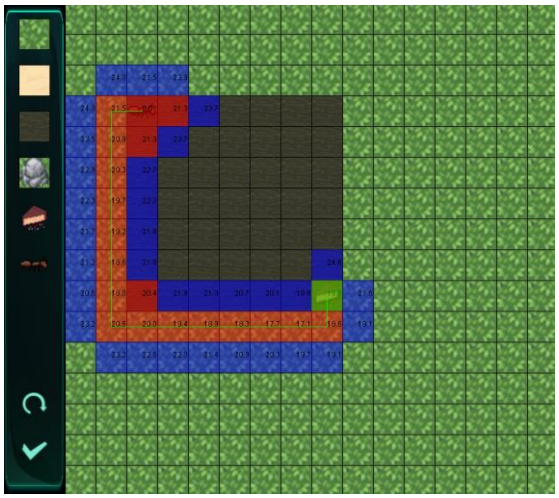
To run the game using command line you can navigate to the folder containing the .jar file and the enter the command: java -jar AStarPathfinding.jar

# A* Heuristic

I bounced around a bunch with the heuristic used. I settled on using the Manhatten Distance Hueristic as it works well with tile-based games. (Total tile count in a line between tile and goal). This is kind of similar to Euclidean distance but takes into account the actual number of tiles and not the direct vector which is less meaningful in a tile based game. I then multiplied the manhatten value by a variable. Adjusting this value would dramatically change based on the situation. When using a value of 4 (the max tile movement cost), the algorithm would often ignore cheaper grass movements in favour for the more direct path through a possible swamp. I instead tried using the average tile cost of all tiles in the map as the value and this seemed to perform better in most cases.

**Left - Multiplier of 4 showing full trudge through the swamp**
**Right – Average Tile Cost of map.**





The main issue with 4 is that it is often an over estimation or worst case (Unless maze walls exist), but the problem with the average is that it can often be an underestimation (especially when maze walls exist). The main issue here is the Manhattan distance does not take into account walled areas that could drastically increase the distance and so large areas will be checked. Both a value of 4 and average tile will struggle. However, I did find that the average tile cost was either equally efficient or better in most walled mazes. In the map shown in the photo below, both versions touched all the same nodes, but the average cost retread already closed nodes less frequently and found a more efficient path through the desert (2 tiles), instead of 4.

One issue of note is that the average tile cost considers potentially unused space which will skew the average. In the photo below all the green space outside the maze is skewing the average movement closer to 1 even though those tiles are not apart of the maze.

**Average Cost Path through intentionally awkward maze**

I debated calculating the average cost for each tiles Manhattan distance. Instead of doing an average for the entire map the heuristic would instead add the average of the actual tiles in the Manhattan distance path for each tile explored. The issue here is that it still ignores the problem of walled paths, so I decided to leave it as is since the point of the assignment was not to find the perfect heuristic. I did enjoy tweaking this though to try and min max my solutions.

# How To Play

Upon launching the visualizer, the player is presented with a 16x16 gird and UI on the left. By clicking the tile types on the left the user switches their cursor to a brush of that type. The user can then click anywhere within the grid to replace that locations type with the held brush. The goal is the piece of pie, and the ant is the start. Once both are placed within the game, a checkmark button appears. When it is clicked the visualizer starts. After the visualizer is completed, a reset button appears above the checkmark that the user can use to reset the map to start again. (This completely reloads the game scene which was a fast-coding solution but does mean garbage collection will go bonkers here) During visualization red tiles represent closed nodes, blue represents open nodes, and green represents the current node. The green line showcases the path taken to the current node. Once completed, if no path could be found a UI display will show saying NO PATH.

# Bugs/Future Builds

**Bugs**

I believe I got rid of most of the bugs and am currently unsure if there are any. A bug from the first assignment was fixed here. I figured out why sometimes clicking would not work. The reason is that the click method only works if the mouse is pressed and released quite rapidly and so using pressed and released checks is a much cleaner solution. It did create some edge case checking like clicking in the screen and releasing outside the screen. However, I am currently unsure of any remaining unhandled bugs.

**Sound/Animation Systems**

There is still no sound system built and though sprites/animations are implemented it was a rushed/hacky version.

**Hit Detection**

Because the tile system is array based and the drawing/mouse location is vector based I had to use some awkward vector location to array math along with int casting to get array positions out of a vector. It may have just been easier to store an int x and y value for array locations. However, the solution still works well and the grid-based snapping works as a result while keeping coordinate values as Vector2Ds. Perhaps creating a Vector2D(int) and Vector2D(float) would be valuable though casting would still happen regardless.

**Game Engine**

Very few updates were made to the engine itself. Animations were the only addition though technically the AStar class could quickly be adapted to run once without waiting for updates and then return the final path for AI objects to then follow for movement.

**Functionality**

There are SOOO many mini updates that could be done to make this even better. For example, the reset button could reset the visualizer without fully restarting. This would allow the user to tweak an already built map with a few minor changes to test again and see how switching a single tile would change things. Even typing this I am SOO tempted to go back in and just reset the state flags, but I need to let myself move on to the next project. It would also be great to allow the user to set the heuristic so they could try a variety of values and see how the different heuristic effects the same map. This would require adding UI elements and input checking to set the heuristic, which would add a much more time intensive inclusion and was outside the scope of the project. Once the visualizer is done it could also pass the path off to the ant so that the ant can perform the path. Again, this was outside the project scope, so I let it be. Another feature would be when to allow the user to “paint” tiles down. So, if the user presses and holds they can drag the mouse around and any tile they cross gets painted instead of having to individually click on each tile. This would require additional flagging for states (Press and hold while using in Brush type) and would not require a ton of time but it was something I thought of only after making a ton of maps for testing and realizing it was a bit tedious to have to manually click over and over.

**Overall Thoughts**

I did hit some major snags during the development of the AStar class. This primarily occurred because I was trying to layer nodes on top without tracking their associated array position. This meant it was complicated to figure out what was a neighbour. On top of that, if I did get info on the tile to create neighours, issues arose when checking to see if the neighbour already existed since Set objects .contains() checks to see if the exact object is already in memory, not if there is an object with the same data values. I was going to create custom equality checking but since I had to resort to looking at the tiles anyways to figure out what was even a neighbour, I figured it was easiest to just copy the map layout and use an array to pull the reference out directly. This does mean that 16*16 references are created in memory and pointed to null right off the bat, but it does mean that the only memory that gets allocated past the memory needed for the references is when a node is discovered. By checking for null first I can instantiate nodes as the path moves through the space meaning memory is only allocated when a tile is first explored in the pathfinding.

# Finite State Machine

## Game Logic and Design

**Game Logic**

Since this is a simulation it really does not involve much “game” logic. The main decision here was to provide the user with a way of inputting the starting colony size. For this I wanted to limit input errors so I chose to use a number input with arrows for increment and decrement. The assignment was a bit unclear if food, water and poison should disappear when interacted with and then a new one gets placed. I debated building it that way but chose to stick with a static map that gets randomized on load. It would be straightforward to add a method that the state calls when it changes to swap that tile to grass and prompt for a new item of that type to randomize in. I also chose to ignore ant on ant collision. Part of the reason for this was that pathing around would have been incredibly difficult with even 20 ants as ants would get stuck on a tile between ants and paths back to the colony could constantly be barred. Not only did this simplify things, but it also makes it feel busier and more like an ant colony as the ants criss cross and overlap as they move around.

Towards the end I decided to add a thought bubble above each ant indicating what their current state was. This was a nice touch as it allows the user to track what each ant is doing at any given moment.

**Game Engine**

No changes made to the engine for this one. Added state classes but those are incredibly specific to this simulation.

## Compiling/Game Setup

Before you can run the visualizer you must first ensure that the Java Runtime Environment (JRE) is installed on your system. If you are wanting to compile the program yourself, you will also need the Java Development Kit (JDK) installed.

**Compiling:**
In order to replicate the development environment, I used the IDE IntelliJ. You can load the FiniteStateMachine folder as a project. Depending on your compiling environment you may need to target the assets folder as the resource folder. You can usually do this by right clicking the folder in the IDE and marking it as the resources folder, though each IDE may differ. You then can then use the IDE to compile/build/run the game. In IntelliJ, open the FiniteStateMachine file (in the game package), and click play (Shift+F10).

You can compile without an IDE. Generally, this will involve navigating to the project folders and using command line prompts to compile all the .java files using the command line javac. This turns the files into .class files which can then be run using the java command line. (FiniteStateMachine.java is the main entry point file).

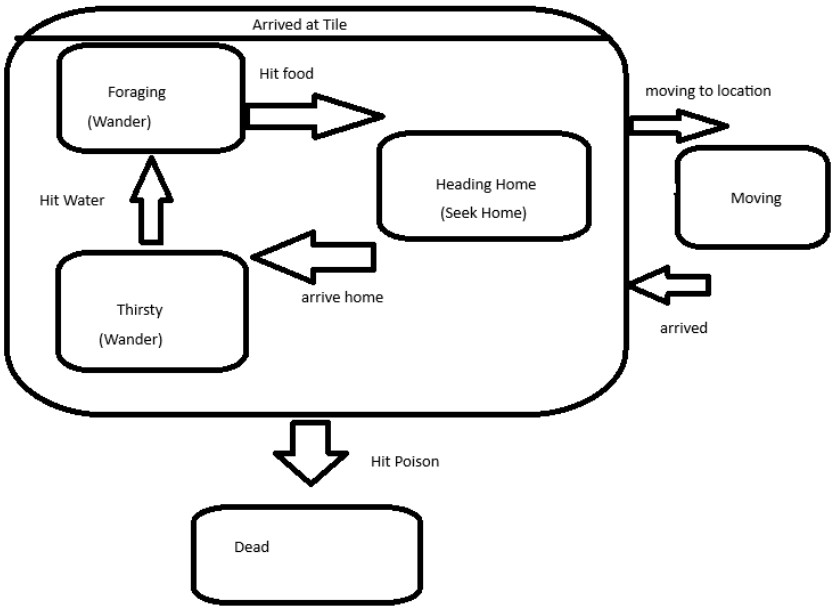
If for some reason the assets cannot be found and you are getting null references, this might be due to miss targeting of the assets folder. The only location needed to adjust will be within the ResourceLoader class. Adjust the folder directory to match your systems directory structure to the location of the assets folder.

**Running from .jar:**
The .jar file is located in the FiniteStateMachine folder. On most systems you can simply double click the .jar file and the game will launch. (Assuming you have installed the JRE).

To run the game using command line you can navigate to the folder containing the .jar file and the enter the command: java -jar FiniteStateMachine.jar

# Finite State Machine

With such a small-scale assignment this would have been incredibly simple to do a hard coded switch over an Enum within the ants update as described in the text. However, I decided to try and build it out with each class being its own state. It would not be incredibly complex to add a hierarchy to the state system used. I did draw up visual of what the state machine could have looked like, however, I did treat movement and death within each state instead of using hierarchies.




Since all states relied on checking the ant against a tile I used the “collision” detection to figure out what the ant needed to do at each step.

# How To Play

Simply set the number of ants you want to start in the colony and hit the checkmark to start. The up arrows above the numbers increase the value and the down arrows decrease (10s and 1s). The max value is 50 and the minimum is 1. If all the ants die the simulation will stop. If the ants reach a colony size of 70 the simulation will end. Pressing the reset button will then restart the game.

# Bugs/Future Builds

**Bugs**

No bugs noted. Input values are caught by the dial system so there is always a minimum of 1 and max of 50 for starting colony size. There were some concurrent issues with removal and adding of adds while cycling through the for loop during an update so the removal and adding now happen after the update cycle.

**Sound/Animation Systems**

Still no sound but the animation system is working fairly well though the game object class does still need to be updated to only use sprites instead of maintaining flexibility for both sprite and static image.

**Game Engine**

I should have used a UI class to overlay information but went with a more quick and dirty solution here. The game core is getting a bit cluttered in its draw sections. Ideally I would overlay UI based on the current state of the game but using basic Boolean works fine.

**Functionality**

The last minute addition of “thought bubbles” really helped add to the simulation. I think adding the functionality to set the starting food, poison and water amounts would be nice touch. It would also be interesting to have the option to flag respawns of food and water instead of having static locations. I treated a food tile as a piece of pie that the ants were grabbing crumbs from instead of returning the entire pie.

In addition, there should be a reset/pause/end simulation buttons during the game. The user currently has to force close the game or wait until all ants either die or hit the max amount. This is a bit of an oversite on my part but added those UI elements during the simulation felt a bit overkill as I am running out of course time.

I also debated adding some checks during movement to allow ants to sense food if they are adjacent to one instead of randomly walking away when right next to a food source. This could make the ants feel a bit more real.

**Overall Thoughts**

I didn’t have much issue with this one, though forcing the use of state classes instead of just hard coding it into the ant took a moment to troubleshoot. The randomized movement had some

debugging and hurdles to overcome. Currently when an ant is against an edge they technically have a 50% chance to move inwards and 25% in either direction along the wall as the direction off the screen flips inward. Not much more to say on this one though.