FULL UNIT FINAL YEAR PROJECT WORK DIARY

Friday 29 September

25min initial project meeting with supervisor Farid

Thursday 5 October - Background

Began reading on multi-agent systems

Friday 6 October - Plan

Began project plan, through to Monday 9 October

Tuesday 16 January - Ideation

[First meeting with Emma, my personal supervisor]

Brainstormed some possible maze games to implement, bearing in mind personal intrigue, requirement satisfaction and suitable complexity. Settled on childhood favourite, family-created board game Labyrinth, which has a simple 2D design, has size scalability, entertainment value, and ability to add more characters and rewards for complexity. This will help differentiate my project because it is based on a unique work, but I will also not be bound to expectation from trying to replicate a well-known work.

Created a Gantt chart in Notion with expected milestones according to the project proposal/plan, including time for updating the project diary, writing unit tests and contingency planning, to keep me on track.

Thursday 25 January

[Second meeting with Emma where she wrote the colab stuff, see history and google meets]

Saturday 16 March – Environment setup, Visualisation, Graph

Set up repository in GitHub, cloned it locally and established its structure - [why is the structure good? All components are in one package like visualisation so easy for someone to come back to. Or separate packages: Different related components in one package, makes modular with self-explanatory names, makes reusable and readable.] Downloaded VSCode for development. I decided to use Python because of my secure familiarity with it. Downloaded Python and created virtual environment. Installed matplotlib and pillow. Made the simple graphical visualisation of the maze game: . I used matplotlib because it made visualisation appropriately simple, and the way Labyrinth works is each maze is composed of tiles, so each tile is a subplot on a figure; each tile has four sides which can be exits or walls, implemented as 4x4 grid on a tile where the corners are either black to indicate blocked, the centre is always white to indicate traversable, and the draw tile function takes as input the exits so the exits can be white and any side that isn't an exit can be black. The tiles will correspond to nodes on a graph, and the exits will correspond to edges. For now added crown and knight to indicate an item that can be collected and the player, respectively. Added type hints, docstrings and unit tests. I forgot to commit as I went along, I only realised after writing unit tests and making corrections to the code accordingly. Instead of attempting to reverse engineer I decided to commit what I had up to that point. Upon running the test I realised the plot was blocking the rest of the code from running, because the figures were staying displayed, so I added block = false.type

Making a wall wherever either of any two adjacent nodes want a wall (instead of where both do) results in way more paths than walls, which is a good thing for this continuous implementation unlike the original board game which is turn-based and has spells.

Remember what the decision made was (dead ends min 1 or 2?)

Test failed (was this because we ran it and though the walls joined up there was a dead end?)

Refactored (to make it easier to see why it didn’t behave like I thought?)

Realised should be class (because running same parameters into all those functions in graph.py for create_maze_graph(). Maybe later

Refactoring observationally/anecdotally seemed to have fixed the error of a dead end appearing.

Unit tests (for why there were dead ends when weren’t expecting, and the rest) [Did three but didn’t finish so committed the next afternoon and moved on as it was a mental block] (Still need type hints and docstrings

2pm to puttanesca dinner around 9.30 maybe 10, then resumed after dinner but emma went to bed and i didn’t work did I? Remember I was comparing it to 10:45 when I came back after burnley brentford on motd

Also bear in mind this was implementing what Emma put on colab so however long that took from January. Plus time to plant the Gantt chart (and whatever else!) counts as planning time. Plus meeting time maybe.

Sunday 17 March

Chose the small maze size to start with as 10 and made it a variable to start with.

Observationally/Anecdotally the deadends haven’t gone.

Got BFS to work by implementing the algorithm in a new file, using bottom left (size-1, 0) as start and top right (0, size-1) as goal and printing the path shortest available path.

Implemented DFS but it didn't work as it went on a huge diversion as it neared the end, and I'm not sure why yet.

Got the knight path to show eventually after using plt.ion() and plt.pause(1) to slow the figure down to drawing the updates every second but not block any subsequent code, and instead of having to close the figure when it blocked which meant we never saw the draw updates. We wanted to see this visualisation because we have to see the knight's updated position, and seeing it follow the shortest path found is helpful.

I think there's more to be said about today's work and thinking behind it. But we want to crack on today while Emma's here.

I need times of working as well!

Started after nap, phone laptop notes, dinner with James

Monday 18 March

Started 10:45. Worked through with toilet breaks until lunch at 15:25.

While being confused as to why DFS result was so long compared to BFS, I found an error in my code that meant it adds a vertex to the list of visited nodes before exploring all of its neighbours, which I thought was the reason the DFS path was so long and winding. I corrected this, but the path was still long and winding, at which point I found out the difference between the path BFS will give you and that which DFS will give you. BFS will search more tiles and find the shortest path, DFS will return the first path it finds, which may not be the shortest as that isn't guaranteed in traversing the first possible path. This means typically DFS path will be longer than BFS, but DFS will search fewer tiles of the maze than BFS. We can draw a graph of this. So when the maze gets bigger and bigger, BFS searching so many tiles may become a problem. I surmised that if I set the maze randomness to a seed, we could run the two searches many times for each seed and plot the length of the path(y) and number of tiles searched (x) on a graph and see the distribution.

I also did research and found something called ID-DFS (https://www.quora.com/Why-cant-we-use-DFS-to-find-the-shortest-path / Jan Andres's answer).

Added different sizes (small, medium, large), let me choose in the command line, and made visualisation reflect whatever size it was. Chose 8, 16 and 32 respectively.

Large maze size updates vis seems to take a really long time (32x32) when medium (15x15) was handled fine, which may be to do with ineficiencies with how using matplotlib. To b e investigated and fixed but prioritsed getting requirements completed. 17% of CPU but same as medium

I'm giving every edge a cost. This will be randomised after graph creation with a nested dictionary. Later, when monsters are placed in the labyrinth, the cost will be decided by hero/current path node proximity to the tile the monster is on.

As a note, the cost function should be refactored because three functions in graph.py use it exactly the same way.

Need to test it still works now added costs to the edges (that - at this point - nothing looks at).

So now next step is implement the uniform-cost graph search using these costs I've given.

17:00 - (prob half an hour off after calling Emma) - 20:10

Tested, found a few errors (not adding value to dictionary key at bottom left visualisation, List() to list(), and reversed unnecessary alteration to checking if the neighbouring tile also has an exit in add_exits)

While looking at add_exits(), I spotted a potential logic error I'm dubbing the 'south, north, north' problem where it might not be aligning neighbour's matching exits correctly. The way to investigate this is to add a unit test to make sure you don't have more than entry for any given cardinal direction – len(current_tile) should be the same as len(set(current_tile.keys())) - if not, you have repeats e.g. north, north, south.

Tuesday 19 March

11:30, figuring out graph costs for uniform search, given I attached the costs to tile_exits/flattened_exits

Before uniform-cost search can be done on the graph, it has to have costs. Yesterday I generated the costs but they were attached only to tile_exits, with adaptations to visualisation.py to account for the additional dictionary. Today I took those costs and added them to the graph by updating graph from a dictionary - node:[connected_nodes] - to nested dictionaries - node:{connected_node:cost} - , changing add_edge to add one edge at a time with a cost, and calling add_edge at two points (exit creation in add_exits() and neighbour matching in update_neighbouring_exits()) instead of once and adding the pair of edges bilaterally because the exits aren't bidirectional anymore, they're now one-directional but in pairs.

In reexamining add_exits() and having to make the graph edges directional, this brought to a head a possible logic error I noticed yesterday, which was that it seemed to state 'if the current tile could have a south exit and the tile to its north has a north exit, then try to give the current tile a north exit' when it should read 'north ... south ... norsth'. This I changed in the next commit. I also noticed the possible redundancy of checking if the adjacent tile has an exit to the current tile before trying to give the current tile an exit that direction because neighbouring tiles are made to match exits later anyway. Another way of phrasing it is: do you even need to check if the tile to the north has a south exit for the current tile to roll the dice for if it creates a north exit of its own, because if the tile to the north has a south exit, update_neighbouring_tiles()

forces the tile to the south to have a north exit anyway. I have left this question for now as that is a wider design decision that can be implemented another time since the maze currently constructs well.

Paused for lunch 14:20. Resumed 15:50

Having made this change to graph, BFS and DFS on medium size both don't return a path at least half the time. This requires investigation and unit tests, but I want to give it time and implement UCS first, then I shall return to it. I also discovered my BFS and DFS algorithms can be optimised for efficiency slightly (to do with popping from the queue and when the node is added to the visited set, respectively) but again I will come back to this as it works for now and efficiency is currently unimportant (until larger sizes are important at least).

Moving on to finalising my UCS algorithm now that graph has costs added at the moment of cost and edge generation.

I've implemented the algorithm, but it has a key error for different keys because it tries to access a key that doesn't exist in the graph, but it should exist as there should be a key for every tile in the graph. I'm also now getting no path found for BFS and DFS for small too. And when it does work, BFS now gets close to short but observably not the shortest anymore.

Stopped at 20:10 for dinner

Resumed 21:45

Now done the three things Emma spotted to graph.py. "Ran temp.py with the debugging feature in VSCode and looked at the graph variable, and I could see it had dead ends (one single exit) even though the visualisation didn't." Ran it and it got all three searches to work on both small and medium as observationally expected! Committed it as UCS worked and the graph was fully matched up to tile_exits with no dead ends – had forgotten dead_end_handling()'s additional exits, and created a check that graph and tile_exits match and that every tile has a proper number of exits (minimum 1, or 2 if avoiding dead ends).

Then created monsters.py, which dictates how many monsters based on maze size and places them in the graph, while increasing (making bigger) the cost of traversing into the tile from any neighbouring tile. Made the visualisation draw the monsters' image in their locations in the figure. I also updated the practicality of the cost, since previously I had randomised it throughout for the purpose of implementing the uniform-cost search – now every edge has a base cost of 1 except those traversing into a tile with a monster in, which has a cost of 5. This satisfies the milestone of specific areas being more expensive than others in the cost function. And finally in looking at the maze graph creation again, I realised dead_end_handling() could be used to totally supersede add_exits() since ("add_exits() was completely redundant because the dead-end handling did the same and more") DEH() only goes and fixes any mistakes add_exits() brings in not creating enough exits by creating more exits to prevent a dead end or tile with no exit, so graph.py was refactored to make the two functions one. This also goes to resolve one of the redundancies I had noticed yesterday that add_exits() before making an exit for a tile in a direction, checks if the neighbouring tile also has an exit in that direction, even though the script will go on to enforce neighbouring tiles reciprocate exits.

Ran the program and it all works as expected! And thus the early deliverables are complete!

Stopped for bed 01:30 02:10

Wednesday 20 March

Started 13:00. Researching the A* algorithm. Wrote code to implement it and print result. Ran it on small and it worked. Began running for medium size at 14:55: took a long while for DFS to find and visualise its first path found; by 15:05 DFS had finished, the figure went to 'not responding', BFS,DFS&UCS had printed in the terminal so it was the first A* search it was executing; so I opened the task manager and found Python 3.11 to still be averaging 18% of CPU but eventually increased memory use from (around 400 MB for DFS to) 4GB to over 10GB, maxing out my laptop's memory, was classed as very high power usage, and my laptop froze at 15:15, forcing a restart by 15:30

Crashed at 15:30. Took time to get over it, lunch, and start laptop up again. Resumed 16:40. Break from 18:15 to 18:35. I've checked what I've written and it pretty reasonably follows the textbook A* algorithm and adapts it to my graph and the goal being all corners, so I need to find a way or ways to optimise it. I've looked at implementing a new heuristic, a minimum spanning tree (MST) of the unvisited corners. The algorithm would be more accurate and explore fewer nodes, but it would have a high computational overhead because it recalculates the MST for the remaining unvisited corners every time it evaluates a node. This would be a problem is larger mazes, and my current implementation struggles with medium 16x16 size. So a custom, simpler but well-designed heuristic might offer a more practical balance between search efficiency and computational overhead.

After a lot of looking into it and composing a custom heuristic, I still can't get the algorithm to work at all on medium in acceptable time or memory requirement. It works brilliantly on small. It works no matter which corner you start in, or even if you start anywhere in the maze not just the corner. I did notice the visualisation getting slower as it went on, as it started going from the third corner to the final. After this long trying to get three heuristic functions for the A* algorithm working on a medium-sized maze, I am giving up (or pausing this for now) and moving on, because it works well on the small size. I might make medium smaller because 16x16 is quite large, but I will ask my supervisor and wait to make these adjustments before exploring other efficiency optimisations.

Stopped for dinner 20:10. Will finalise what I've written for A* and commit after dinner, then move on to next deliverable. Resumed 21:25. Distracted for a while from 21:50 to 22:50. Made a wrapper function in a_star_search.py to account for my custom heuristic needing one more parameter (N, being size of maze) than the standard two heuristics – I chose to wrap the heuristic call in a lambda function because the discrepancy needs solving and this solution is modular, flexible and scalable since it inspects the signature of the heuristic function instead of just identifying 'if heuristic == custom_heuristic" which is direct and not versatile. Ran a couple times on small to check all six paths are found correctly – they do. I even found an edge case where the way the maze set itself up paths all paths were drawn to the crown but the A* algorithms returned no path found, because even though all four corners were accessible it would have to go back on itself at at least two points. I think I should start outputting the seed so when I screenshot examples, I am able to recreate it to explore when it's interesting like this edge case. Paused at 23:25

Thursday 21 March

Started 11:25. I begin work on the next deliverable: "formulating a search problem for collecting all items, implement A* and greedy search to do that with minimal steps, visualise, and compare on three maze sizes." For the comparison, comparing effectiveness (path length) and efficiency (steps, searching) could do. If objects placed around aren't a good enough proxy for PacMan's food, I could implement PacMan food-like stuff being 'supplies' or 'mana'.

So I'm going to need an array that is a list of locations where there is an item, to be passed to the A* and greedy searches. For now I have chosen the items' locations in each size grid by my own random dictation. I can now apply one of the A* algorithms to return the shortest path that goes through the currently five items. Will refactor A* algorithm script to take a list of locations when temp.py calls it, it can feed it either a list of the corners or the list of items.

I have today implemented items in the graph, and refactored a_star_search.py (a_star_search()) to take any list of locations not just corners, in order to collect items. Not visualised items on the graph or the path taken yet.

Moving on to greedy search for collecting all items. Done and committed! Not that much to it; it's similar to A* but simplified, as it disregards the cost so far and focuses solely on the heuristic for decision-making.

Earlier I arranged a meeting with supervisor Farid for Monday. Gone for a walk at 17:10. Resumed 19:10.

In thinking about removing the drawn knight's image once the path visualisation is complete so that the next path could be, because Matplotlib can't directly remove the image once it's been drawn but you can clear the tile and redraw it without the image, but if the knight walked over a path with a monster or item on you would want to redraw it with them too, it occurred to me another reason it makes sense to refactor the visualisation as having a class of tile and you could add or remove images to be drawn onto it. This way you could remove the knight from the list but keep any monster or item when redrawing in the simplest way.

Paused for dinner 19:55.

Friday 22 March

Continuing with visualisation. Finishing up on erasing the path last drawn and redrawing the tiles to have the next path drawn. For the purposes of starting earlier than this but took a couple breaks, say started 11:45. Paused at 13:00 to have meeting with my personal tutor. Resumed 15:00. Took all A* searches out (all corners and all items) and the other four searches (including greedy all items) worked pretty instantaneously on small, but on medium and large BFS,DFS&UCS all worked but greedy collecting all items didn't. Greedy didn't take loads of memory consumption (750MB), but I stopped it after almost five minutes. Wondering if it's to do with going in a cycle forever in the maze because of not being able to tread on a tile it has previously gone over.

I'm going to change monster and item placement to randomise locations each time dependent on the size of the maze, so I can change how big each size is and this change is automated throughout. I will then reduce medium and large sizes considerably to see if this makes a difference to A* and greedy searches (since I know all works well 8x8), and so that the upcoming automated comparison of A* and greedy for collecting items can actually complete.

Added items to visualisation, all as the key image. Updated erase_path() to redraw all crown, monster and item images in case they were on tiles that had just been redrawn to remove knight image for the previous path.

Take 15 minutes off my time. Beginning script for comparison of A* and greedy algorithms for collecting all items. Agreed something like worked 15:00 – 17:00.

Saturday 23 March

11:45.

You want a seed to give both algs a fair chance by giving same set (same mazes to both algs); also makes sure each run different since i is different as it loops through; reproducibility in science is really important in making claim about efficiency and effectiveness of algorithms, so other people can come along and verify it (how exactly?)

Half an hour diversion around 13:30.

In making the comparison, because medium at 16x16 just doesn't work, I am finding the largest size that will work. Small has always been fine so 8 is good, now I know 9, 10 and 11 are good, but 12 is unworkable (is that because there were two corners where the path would have had to go back on itself?).

And for the first tests of the comparison function, I am running only 3 times each and only for small and medium with this new size for medium. Once it works will expand to large (once I know what size that should be too) and 10, 50, 100 times. You'd scientifically like it to be 1,000 each but that would be 6,000 runs per comparison which would take hours.

In observing the A* search, and I may have made this point earlier because I certainly notices it days ago, its algorithm may not allow it to tread on the same tile more than once in its solution, which is a logical fault as the shortest path to collect items should be able to complete a figure-of-eight for instance. I will investigate my algorithm to see if it does forbid this and if that is why it is impossible to complete (without unavailable time or without crashing) for maze sizes over 16.

Be mindful of the error the program just pulled wherein if the path trying to be erased is ‘None’ because for example there is no UCS path from the entrance to the crown (because of walls in the way making a separate room perhaps) a loop can’t iterate through None. So catch that the path isn’t None, perhaps.

Now I’ve randomised monster and item locations, I need to think about making so monsters and keys can't be on each other (could they be on the other though?) or on the entrance or crown.

Stopped 15:15. ... check notes app

Removed from commit message: “Makes medium 12x12 instead of 16x16.”

Monday 25 March

35min final project meeting with supervisor Farid

Friday 29 March

18:00 – 23:00

Working on making monsters move to the hero. I had the thought that predictive modelling would be good to have the agents predict and pre-empt their adversary’s future behaviour, but that this is too complex for now.

Flirted with a D* Lite implementation to dynamically route the monsters’ path to the hero as the program runs, and got all the way with implementation, but this turned out to not achieve what I wanted so scrapped it in favour of the existing A* algorithm but only running it for the end part of the monster's first routed path to the hero that would become out of date as the hero moves around.

# Initiate maze with knight and monster positions

# Monster runs A* search to find best path to knight # Knight decides a path for itself (? use a simple one for now)

# For now, use a fixed number of turns (later you will need a while-loop until the game is over) n_turns = 20 for turn in n_turns:

# Monster changes its position one place along its path

# Knight change its position one place along its path

# Monster updates its path according to knight's position

//

1. Add function to items.py that adds an item to every tile

2. Amend knight's search algo so that it aims to visit every item/tile, not just the crown

3. Instead of a for-loop for the knight's path, make a while loop (while not game_over:). At the end of each iteration, check if any monster is currently occupying the same location as the knight; if so, game_over = True. Also when there are no items left, game_over = True

4. When updating the knight's position, make him run through these steps:

    i) Get the next point on the path

    ii) Check if there is a monster on that next point

iii) If no monster, move there and continue as normal

iv) Otherwise, check what other exits there are from this tile according to the graph

v) Iterate over those exits; for each one check if there is a monster on the adjacent tile; if no monster, move there

vi) Redo the A* search for collecting remaining items starting from new position? Not sure

Also return the list of uncollected items

Had the thought that it'd be better to have the monster path adjustment function stop the loop that finds the closest point in the current path to the hero's new location when the Manhattan distance for a point if 1, so it can stop there instead of cycling through the rest of the path that won't be any closer. Or even making it cycle through the monster's path backwards from the end, since that is more likely to be nearer the closest point to the hero than the beginning of the monster's path. But this loop through the path isn't very costly so it's not a huge inefficiency to worry about.

Saturday 30 March

Started 11:30. Stopped 18:45.

Committed changes to A*-greedy comparison that adds success rate since they don't always return a path (because not every maze layout will allow you to travel to all the handful of items), in addition to making the A* search more efficient (by getting sets involved in the list comprehension for its computation speed quicker than a list, and creating a limited priority queue as a modified heap which pops highest-cost items so it doesn't store more than a thousand and run up memory) so it can run on the larger sizes of maze.

Now then; looking at the project description again, the deliverable for comparing A* and greedy searches to collect items had items in every cell or tile of the maze, rather than the handful dotted around like I had implemented. So I will make that change, before moving on to the next milestone of getting the monsters to chase the hero with a game-over possibility, and making the hero dodge the monsters while it collects items from every tile.

Had trouble in that I was implementing the monster chase under the wrong search algorithm essentially, as well as losing sight of what the hero's goal for this milestone is, so I have to carefully unpick and reapply what I've written to a new search or the item collecting I achieved previously that just got the number of items wrong.

Sunday 31 March

Started 10:30. Stopped 14:45. In the evening call it 2.5–3 hours.

The seeds don't work (they don't reproduce the same maze) and I'm not sure why.

After working concurrently, I managed to commit to finalise the A* and greedy search comparison for collecting items from every tile (made them efficient and improved path return rate), and to make the player a reflex agent avoiding the chasing monsters. I got the monsters to recalculate its path to the hero as it will move during the run, by redoing the A* search to the player but only the part of the path that's no longer any good, so it reuses the part of the path that still is the best route towards the hero. And now that monsters are chasing, there are game win and loss conditions; win if the player collects all the items, lose if caught by a monster. I also refactored the path visualisation to apply for any character (ie monsters too) and to not show the trail anymore.

Moving on to minimax!

Monday 1 April

Started 11:45. Break 14:45 – 16:05. Just five or ten minutes. Resumed 18:30. Dear god. 21:00-21:30 dinner. Distracted 22:00-22:50.

Feeling great about minimax linking together, getting into object-oriented and making the labyrinth a gamestate.

Now A* done for later deliverable it makes the touching corners one from before double touch. A new bug introduced but it worked before perfectly. No time to investigate but interesting. Still achieves task but not most efficiently on one of the heuristics not both!

Tuesday 2 April

12:50. Frick! Finally linking the best node through to the top-level node's child. I'm just a bit tired of working on this every day. My fault for leaving so much over so little time, but it is hard. Almost two hours gone for the toilet. 18:20 I started in the evening, having gone for a nap 16:20 after having lunch. I reckon 1.5 hrs work before then. Stopped for dinner 21:20. Resumed 22:30. Realistically did half an hour's work until 00:40.

Finishing minimax at long last, debugging A*-greedy comparison, quick alpha-beta pruning. Shan't be time for expectimax, but no worry.

For deliverable 6 (A* and greedy for collecting all items) in the monster section I changed path[1] to path[0] since _that_ is what holds the position it is being moved to, and no problems on small but medium doesn't make a single character move. Task Manager isn't reporting GBs of memory, the usual 730MB +350MB, but the figure goes to 'not responding'. This isn't running the comparison yet either. The solution might be found by debugging, using breakpoints or with print statements, but I should leave this for now but it possibly is important that this deliverable is completed. Gah, why is A* so troublesome!

EDIT: After five or ten minutes, the execution completed! It just took that long to establish there wasn't a path that collected all items for the hero using greedy search. Understand why it'll take all night to run 100 times on small medium and large for both searches. Testing that now before I run it for good while I go off for bed

Wednesday 3 March

Report writing. 11:55.