Fire of Eidolon

Josh Mecham
University Of Utah, Electrical Engineering
Palace Acquire Program, US Air Force
Salt Lake City, USA
u0920616@umail.utah.edu

Mark Sneddon
University Of Utah, Electrical Engineering
Palace Acquire Program, US Air Force
Salt Lake City, USA
markbsneddon@gmail.com

Travis Carlisle
University Of Utah, Electrical Engineering
Palace Acquire Program, US Air Force
Salt Lake City, USA
traviscarlisle@hotmail.com

Tye Foster
University Of Utah, Electrical Engineering
Palace Acquire Program, US Air Force
Salt Lake City, USA
tye.foster25@gmail.com

***Abstract*—Implementation of an adapted version of the board game "Fire of Eidolon" created by Magic Meeple Games. This adaptation makes use of Motion Planning Algorithms to navigate a grid-like dungeon to complete the game. This paper describes the methods, experiments, and results of adapting the board game virtually.**

Keywords—Breadth First Search (BFS), Depth First Search (DFS), A*, D*, heuristic, Random map generation

I. Introduction

The goal of our project is to design an autonomous version of the board game "Fire of Eidolon". The project will simulate "players" exploring a dungeon, unlocking the titular Fire of Eidolon, and escaping to the exit. The game has many levels of complexity ranging from simple to intricate, allowing for a unique and interesting application of motion planning. A key feature of the game is the dungeon exploration aspect. By implementing an ever-changing maze as our simulation graph, we can explore a large number of motion planning techniques.

Fire of Eidolon is played with 1-6 players in a grid-like dungeon. Players must recover the Fire of Eidolon by cooperatively exploring the dungeon and challenging the dangers in each Chamber to collect Tokens of Power. These Tokens are used to destroy the three Dark Relics sealing the Fire of Eidolon and escape the dungeon with the fire before the cultists of the Dark God Vorax complete their ritual to plunge the Fire of Eidolon into the Void forever.

Simulation will consist of 1-6 robots collecting tokens, searching for chambers, and then collecting and escaping with the Fire of Eidolon. Simulation design will include maze/dungeon generation, ranging rules of game complexity, changing goal states, multi robot simulation, and of course, lots of motion planning.

The first simulation would be to be able to complete a known maze without any barriers, specializations, or cultists who delete tiles. The second simulation will complete a known maze with specialization and barriers. The third simulation will complete an unknown maze with specialization and barriers. Our first stretch goal will be to complete a known maze with specialization, barriers, and cultists who delete tiles. Our second stretch goal will complete an unknown maze with specialization, barriers, and cultists who delete tiles.

Goal set: G(G0, G1, … Gn) Goals will vary in time during the game, ie multiple goals within a game. Goals will change according to which simulation is running

Simulation Variables:

- Known vs Unknown State
- Specializations vs General Abilities
- Barriers vs Free Movement
- Number of Players (1 to 6)
- Cultists who delete tiles vs No Tile Removal

Action set: A(Right, Left, Up, Down, Grab, Break, Trade, Attack)

State set: S(s0, s1,..., sn) 'n' = to number of available tiles "explored"

Transition functions: move from one tile to another according to action

Costs:

- Grab Token (1, 2, or 3 cost depending on specialization)
- Grab Fire of Eidolon, Movement, Trade, Break, or Attack (1 cost)

II. Related Work

A. *D* Lite*

D* Lite is an application of Dynamic A*, using a heuristic to determine the quickest path to the goal. Using this new method, a robot can calculate a new path to the goal when an obstacle is found in the way of the planned path. The difference between D* Lite and most other planning algorithms is the speed at which it can calculate a new path. Most algorithms (like A*) will recalculate a path from its current position to the goal when an obstruction is found. D* instead finds an alternative path with the same cost as the original without doing a complex start-to-finish calculation [1].

In our project we won't have much opportunity to utilize D* Lite but the concept is enticing. If we reach our stretch goal of implementing cultists that can appear and remove map tiles from

the dungeon, new paths must be made. Spending less time in path planning calculations will optimize our performance.

B. *D* Lite Based Real-Time Multi-Agent Path Planning in Dynamic Environments*

The algorithm proposed in this paper employs a modified D* Lite algorithm. The modifications added are a dynamically changing obstacle graph and a prioritization list for robot paths. The robots with higher priority tasks plan first. Each node occupied by the robot while traversing the path is updated to appear as an obstacle to any other robot trying to plan. This condition is only temporary. Robots can have overlapping plans, but the condition ensures robots will not collide with one another [2].

For our project, we can use the task priority list and modify the dynamic obstacle map approach to meet our needs. Instead of turning paths into temporary obstacles, we can have a planned path update the cost function for other agents. This will disincentivize but not disallow other players from going down the same path as another player at the same time.

C. *DARP: Divide Areas Algorithm for Optimal Multi-Robot Coverage Path Planning*

DARP splits a predefined area with defined obstacles into multiple sections for multi-agent path planning. The DARP algorithm divides the terrain into several equal areas each corresponding to a specific robot, so as to guarantee complete coverage, non-backtracking solution, minimum coverage path, while at the same time does not need any preparatory stage [3].

Our group can utilize the DARP algorithm after the goals and subgoals have been discovered to coordinate and path plan for equally for the multiple agents. DARP cannot be used to coordinate path planning in the map discovery phase of the project because the algorithm requires defined obstacles.

D. *Optimal Multi-Agent Map Coverage Path Planning Algorithm*

This algorithm improves and optimizes DART to reduce the planning time and the number of times where no plan is found. The study runs the two-pass algorithm or connected components algorithm before running DARP. [4]

The paper found that using the connect components algorithm first made the planning for multiple agents more efficient and resulted in less failed path finding attempts.

Our group will utilize this hybrid version of DARP to optimally path plan for the multiple agents after finding the subgoals and goals.

III. METHODS

A. *Map Generation*

The first task was to generate a map of using the Fire of Eidolon game tiles. We elected to use Pygame software platform that would allow us to code our Fire of Eidolon emulation in a simple manner. Pygame is freeware that reduces the amount of code required to make a video game. Pygame allows for modular coding in python runs in the main function of the python.

Pygame gave us an easy way to create a map using pictures of the game tiles and a way display the path taken by players and the accompanying number of steps taken. Overall, we chose Pygame to create the map and the game due to its simplicity to create a clean looking way to display the map and the characters.

All the game tiles were scanned and saved. Fig. 1 shows the different classes of the Fire of Eidolon tiles. Each tile attributes were then added to a dictionary. The dictionary included important attributes such as the name of the tile, the color of the associated token, the orientation of the passages, and whether the tile was a special event tile. The event tiles included start tile, the blue, green, and red event tokens, the secret passages, the Fire of Eidolon tile, and the end tile. The back side of the tiles were scanned to act as a placeholder inside of the grid of the map.

Figure 1: Fire of Eidolon Game tiles and tokens

We used the dictionary of game tiles to generate a random map that would connect the passages in a grid gameboard structure using the following pseudocode:

ALGORITHM 1: RANDOM MAP GENERATOR

```
    Input: Dictionary of game tiles
    Output: complete map grid with connected tiles
1   Get the list of the game tiles
2   Make a large grid map of filler tiles
3   Replace center of map filler tile with start tile
4   while (game tile list not empty)
5       while (checked all cardinal direction = FALSE)
6           if (door exists in cardinal direction)
7               if (tile space is empty)
8                   grab a random tile from game tile list
9                   replace the filler tile
10                  remove the replacement tile from tile list
11                  while (replacement tile does not have a
                    door in opposite of previous direction)
12                      rotate replacement tile clockwise
13                  if (working tile has unchecked cardinal
                    directions)
14                      save tile to open spots list
15                  set replacement tile as working tile
16      if (no tile has been placed)
17          set working tile = a random tile from
            open_spots_list
```

The python implementation of the algorithm 1: random map generator produced a game map that connected all the passages

of the game tiles. Fig 2 shows a generated Fire of Eidolon map displayed with Pygame.

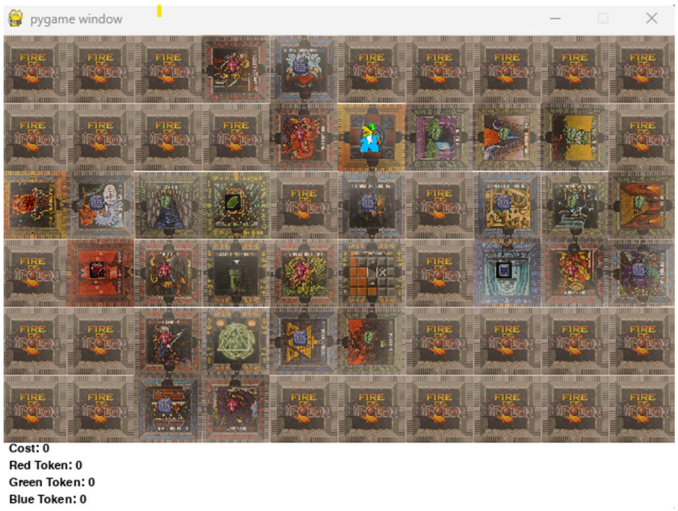

Figure 2: Game map randomly generated with algorithm 1.

B. *Manual Game Play*

The initial form of the game is played by a single player. The player is placed on the start tile of the randomly generated map. Using the keys 'w', 'a', 's' and 'd' for moving up, left, down and right respectively. The player can pick up a token using 'e', use 'q' to take the secret passage, and 'p' to perform a single A* search. The planning button 'p' is not necessary for manual gameplay but is available. The lazy button 'l' allows the player to skip their action without adding cost to the total cost.

The user inputs valid moves and the player moves accordingly. The user moves the player around the board until enough tokens of a color are collected, then breaking that colors seal. Eventually allowing access to the Fire of Eidolon which is collected and moved to the end tile. This ends the program, and the player has now won the game.

C. *Motion Planning Algorithm*

For the game to transition from manual gameplay to a program which plays itself, a motion planning algorithm needed to be utilized. There are numerous methods available that would solve a simple maze including BFS, DFS, UCS, A*, etc. We decided that the method best suited for our task would be A* based on the speed in which A* can produce a valid path, the limited number of nodes considered, and the benefit of heuristics with a known goal state. This planning method is the starting point for the rest of the project and provides a good base for any future developments.

The pseudocode for our A* implementation is as follows:

- Initialize current player location as a 'SearchTile'
- Create frontier as a 'PriorityQueue'
- Create set of visited tiles, 'visited'
    - While tiles exist in frontier, remove tile with cheapest cost
        - If not in visited, check if tile is the goal state
            - If yes, return path, actions, and visited
            - If not, add each valid tile neighbor to visited and generate cost for each

A SearchTile consists of cost, how much did it cost to get to this tile; parent, which tile came before; parent action, how did we get to the current tile from the previous; state, where is the tile located on the map; and actions, where are the valid neighbors. The PriorityQueue is a sorted list. The sorting is based off cost and the cheapest item in the queue is always the next tile to be checked. The cost is computed by summing both the cost-to-come and the cost-to-go. Cost-to-come is a forward accumulation of the distance a player would travel to go from their current location to the tile in question. Cost-to-go is a heuristic cost based off Euclidean or Manhattan distance that does not accumulate.

D. *Heuristic*

We tested both the Manhattan and Euclidean heuristic models in our program. From the player's position we could determine the heuristic cost to any given tile on the map by use of previously assigned tile id numbers. If the tile number was a zero, that indicated that tile was one of our obstacles tiles the player can't pass through. We would give the obstacle tiles such a large heuristic cost the program never even considers that tile a movement possibility.

We didn't find much difference between the two heuristic methods, probably due to the small size of our map. Ultimately, we decided to stick with the Manhattan Heuristic since our players won't be moving diagonally at any point in the game.

As part of the game, you can move from one secret tunnel to the other for only the cost of one action. As such it was advantageous for us to use them in cases where taking the tunnel would overall reduce the cost to the goal, even if that meant moving away from the goal for a couple of moves. Using the basic heuristics formulas rarely gave us this result in our A* adaptations, especially in maps that were spread out. We adapted the heuristic so that after getting the heuristics values for each tile, it would determine which secret tunnel entrance was closer. The farther secret tunnel entrance tile heuristic cost would become equal to the closer secret tunnel entrance tile heuristic cost plus one. We would then create a second heuristic map based on the farther secret tile and add farther secret tile heuristic value to each of those values. We then compare the two heuristic maps and at each point and kept the smaller heuristic value for each given tile.

We ran into a problem where the program would prioritize taking the tunnel to rather than continuing its path. We solved this problem by first running our Manhattan heuristic from the player's position with secret tunnel heuristic part included. We would then run our Manhattan heuristic from the goal point to the player's position with the secret tunnel heuristic part included. We would then add the two heuristic maps together. This would make the tiles that were far away from both the starting point and the goal tile less likely to be used, even if they have a connected tunnel.

E. *Automatic Game Play, multiplayer, and D**

For the game to play itself, we implemented a function that can determine a player's next move and then move the player

without user input. At the beginning of the players turn, after every 3 actions, A* is ran. The autoplay function determines the closest goal, whether that be a token, seal tile, Fire of Eidolon tile, etc. The shortest path and actions needed to complete the task is returned to the player. After the autoplay completes 3 actions, A* is run again and a new set of actions is provided back to the player. Since most tasks take more than 3 actions to complete, the actions returned in between turns are often unchanged minus the 3 previous actions completed.

To increase the number of players, we manually create new instances of the player class. Each of the new players is given an individual color. Each player is given access to the next player and has an attribute of 'active'. Player 1 starts out as active and during a turn transition can deactivate itself and activate the next player, Player 2. Just like in single player, each turn consists of 3 actions, after which the next player can begin their turn. During autoplay, each player uses A* to begin its turn. Each player is provided a different goal, otherwise at the start of the game, all players would move to the same closest token. Gameplay commences by input of 'u' by the user and multiplayer autoplay continues until the game is won.

As referenced above, one of the future goals is to create a version of the game that utilizes D*/D* lite. In the game's current form, A* is run again and again at the beginning of each player's turn to determine the next set of actions to take. This is a fine approach for our simplified version of the game. But as we incorporate more rules of the game, we will need better methods for replanning. For instance, in more complicated versions of the game, cultists appear on the map as play occurs that can delete existing tiles. Players would then need to replan according to cultists on the board to eradicate and avoid tile deletion. If the players do not eradicate cultists, the state of the board could become unwinnable. As tiles are deleted, we also run into the case where the currently planned path is unusable and must be accounted for. This is the reason why D* is necessary going forward.

IV. EXPERIMENTS

A. *Version 1: Manual Single-Player Testing*

Our first iteration of testing was moving a single character around our generated map to play the game. The game was setup with goals and subgoals:

- The player must have the Fire of Eidolon to escape and win.
- The player must have all 3 Seals of Vorax to be broken to collect the Fire of Eidolon
- The player must have at least 6 tokens of the same type to break the corresponding color Seal of Vorax.

To keep track of cost we made this single player stronger than the typical character in the game, taking 1 action point to pick up any token.

This version was put in place to verify the basics of our program were in place. We wanted to be sure pygame had been implemented correctly, and the functionality of the goals and sub-goals allowed the game to function correctly. See Fig. 2 for an example of single player testing.

B. *Version 2: Automatic Single Player Testing*

We then added automatic pathing using A*. Implementing a "goal-updater" function we would have a simple AI identify what task needed to be done. Once the AI knew where it needed to go, we used A* to find an efficient path to the goal. This proved to be effective and demonstrated our actual motion planning was working in our program. See Fig. 3 for the python implementation of the Automatic Single Player test.

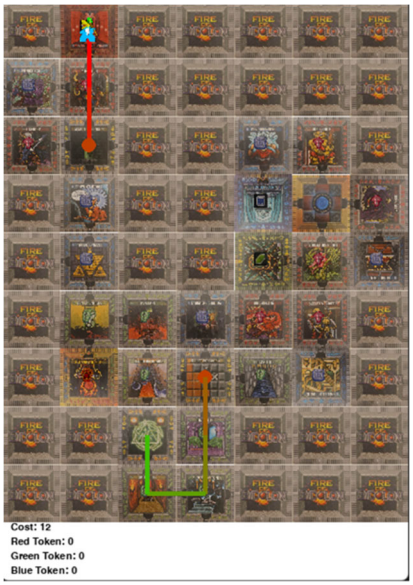

Figure 3: Automatic Single Player Test

C. *Version 3: Multi-Player Testing*

After a single-player version was working automatically, we added multiple players on the same board. All players had the same strengths as the original and were able to pick up any token for 1 action point. Once any character had spent 3 action points moving or picking up tokens, the game would automatically update to the next player's turn. Each player's path and total cost was tracked separately, and the AI was able to operate with each player on their turn. See Fig. 4 for the implementation of the multi-player game.

Figure 4: Beginning of a Multi-player Game.

Figure 5: Final Display of a Multi-player Game

For this experiment we wanted to make the game more realistic. While many engineers are forced to play games alone and without company, most people play with friends. This not only makes the game easier but more enjoyable.

D. Version 4: Character Specializations

Our final version had character specializations that would be better or worse at grabbing certain tokens. For example, one character would spend 1 action point to grab a blue token, 2 action points to grab a red token, and 3 action points (their whole turn) to pick up a green token, while another player would spend 1 action to grab a red token, 3 actions to grab a blue token, and 2 actions to grab a green token. Having strengths and weaknesses spread out among the players makes much more diverse and interesting gameplay.

If the AI was in charge, it would direct any player to the closest needed token that related to that player's strength. If there were at the tile location but did not have enough actions to pick up the token, the player would pass their turn.

The purpose of this final version was to create a version of the game close to the original. Players would have to rely on each other to collect the tokens, using their strengths and accounting for their weaknesses. We ensured that the program did account for picking up tiles and each player's action points were accurately tracked. Refer to Fig. 5 for the final implementation of the multi-player testing.

V. ANALYSIS

A. Tile Map Generation: Depth First Search

When developing the map generation, we wanted to be able to test out our program against multiple types of maps. We first figured out how to place each tile individually but quickly learned that took too much time to develop multiple types of maps. To best mimic how human players would develop the map during gameplay, we wanted the tiles to be connected in as much of a singular path as possible, only changing path locations when the current path could no longer be extended. Depth First Search proved to be extremely effective at this endeavor. Using doors as connecting points allowed the program to quickly create a map where it was possible to reach every tile and allowed for deep paths to be taken.

We first made it so that when a path could not continue it would go back to the first possible open path. This created rather uninteresting maps where the most efficient path to a goal was also the only path to the goal. After that we tried it so that the new path would start and build from there. Again, though we would find the same problem of the most efficient path to the goal was also the only path to the goal.

We realized that it would be more realistic to simulate multiple players creating the map at the same time, both using other players tile placements and creating their own deep paths. To do that, we implemented a random element, where once a depth iteration got stuck, it would take a random tile that had door openings and would begin to build the next depth first search from that tile. This proved to be very effective at outputting a complete map in a timely manner.

It cannot be said that this method is perfectly complete as due to the natures of many of the tiles only having one door and the random placement of tiles, it is possible to completely block

off any chance of victory by having the first four paths end with no open doors as shown in Fig. 6. This is, however, extremely unlikely to happen and rarely happened in the experiments.

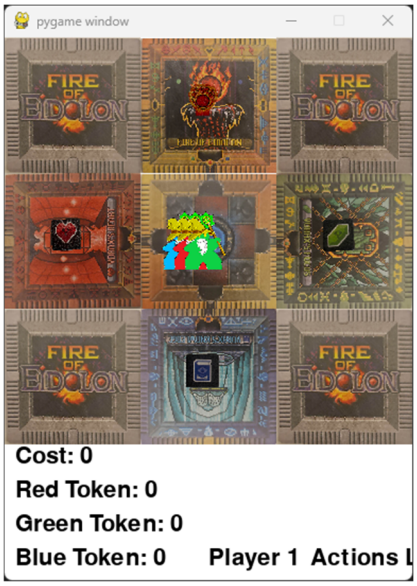

Figure 6: Unsolvable Map

B. A Search*

Our implementation of A* is optimal in terms of path length for any single path generation. Meaning, while playing the game manually or during automatic gameplay, any generated path is the shortest length from start to goal.

There is however no guarantee during automatic gameplay the entire path taken by all players is the shortest path to win the game. With automatic gameplay, paths are updated as different players take their turns, events occur, and tokens are picked up. In each of these scenarios, each player is playing the game independently in the sense that no data is shared between the players. Two players could be going along the same path when spreading out may be more efficient in terms of minimizing total actions needed to win. This is something we plan to work towards moving forward. The solution to this problem involves better task allocation using AI.

VI. DISCUSSION

This project was an excellent exercise in motion planning. We were able to take information we learned early in class and apply them to an interest that all members of the group shared. Playing board games has always been enjoyable for the four of us and adapting one of our favorite board games into a digital version was very engaging.

In the process of working through this project we learned how to apply algorithms we had used in homework in a similar way but now used in a way that we decided instead of how the problem description required it. Using heuristics in A* and defining the start and end goals were good practice on concepts we had only briefly worked through in the homework.

Additionally, we learned how to better work with python in general including but not limited to: file management, imports, function implementation, and plugins such as pygame, pynput and more.

Coming away from this project and report, we hope the reader gains inspiration and confidence in coding. All four of the members of this group are not particularly strong at coding but we were able to implement a hobby that we enjoyed together into a virtual space. We firmly believe that if there is enough determination to see your end product you will always be able to push through the roadblocks that may come up.

Finally, if we were to extend this work, there would be so much more to add. The version of the game we have now is a somewhat simplified version of the full game. We hope to continue to work together on this project after this class is over to implement the following:

- Map Generation by exploration.
- AI coordination between different players.
- Player token trading.
- Cultists that can appear to remove tiles from the map.
- Adjustments to the AI to account for these cultists, eliminating them before tiles are removed.
- Implementing other game modes that include shades and other special events.

ACKNOWLEDGMENTS

We would like to officially acknowledge Magic Meeple Games and especially the co-owner and co-founder of the company Ian Stedman with whom we were in direct contact with about adapting his game into a digital version. Ian was very helpful and enthusiastic about our efforts and he and everyone at Magic Meeple Games our gratitude.

We would also like to acknowledge and thank our professor Dr. Alan Kuntz and the Teaching Assistant Mohanraj Devendranshanthi. They both helped with the conceptualization of this project and provided advice on how to best tackle any issues we came across.

REFERENCES

[1] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," in IEEE Transactions on Robotics, vol. 21, no. 3, pp. 354-363, June 2005, doi: 10.1109/TRO.2004.838026.

[2] K. Al-Mutib, M. AlSulaiman, M. Emaduddin, H. Ramdane and E. Mattar, "D* Lite Based Real-Time Multi-Agent Path Planning in Dynamic Environments," 2011 Third International Conference on Computational Intelligence, Modelling & Simulation, Langkawi, Malaysia, 2011, pp. 170-174, doi: 10.1109/CIMSim.2011.38.

[3] Kapoutsis, A. C., Chatzichristofis, S. A., & Kosmatopoulos, E. B. (2017). DARP: divide areas algorithm for optimal multi-robot coverage path planning. Journal of Intelligent & Robotic Systems, 86, 663-680.

[4] Y. Zheng, X. Tu and Q. Yang, "Optimal Multi-Agent Map Coverage Path Planning Algorithm," 2020 Chinese Automation Congress (CAC), Shanghai, China, 2020, pp. 6055-6060, doi: 10.1109/CAC51589.2020.9327261.