Technical Achievements

PROJECT 1: MARS PATHFINDER

Team Space_Invaders

Akshaya Karthikeyan, Avani Gupta, Dipanwita Guhathakurta and Tathagata Raha

---

Introduction

This project was created as part of the Microsoft Engage 2020 Mentorship Program. Two options were offered to us, and we chose to develop Project 1: NAVIGATE THE MARS ROVER.
In this project, we helped the Mars Rover find the shortest path between two points while avoiding obstacles on the way. We exhibited good teamwork and thus, were able to incorporate many additional features such as maze generation algorithms, message bot TARS, multiple checkpoints etc. We also followed good coding principles and structure. In this document, we will list our technical achievements in this project, from the programming paradigm to state diagrams and more.

Running the Project and Local installation:

The project is deployed in Azure. You can access it here:
https://tathagataraha.z13.web.core.windows.net/

Local running:

- At first clone the repository
    - git clone https://github.com/Akshayakayy/Space_Invaders
    - cd Space_Invaders
- You can open the visual/landing.html to test it in the browser.
- For development and changing files in the src folder, follow these steps.
    - npm install (install the required modules)
    - gulp compile && mv lib/pathfinding-browser.min.js visual/lib (to compile the src folder)

Tech Stack

Gulp - Used to compile the src folder to form a .min.js file
Bootstrap - Bootstrap for decoration of the UI.
Sweetalert - Used for the guide and showing the stats in the end.
Raphael - For generating the grid on which the whole thing happens.
Npm - node modules.
Git - version control.
HTML, CSS, JS and other basic web dev tools are used to write the whole project.
Jquery - JavaScript library for making for HTML document traversal and manipulation, event handling, and animation.

Programming Paradigm: Object-Oriented Programming

Objects

Agent

Our agent is defined in the visual/js/agent.js file. Based on the user inputs from the panel and the grid, the agent performs the job required.

- If any maze algorithm is selected, the agent clears all the obstacles and renders that particular maze algorithm.
- If any search algorithm is selected, the agent searches for the destination accordingly.
- On changing the speed, the speed of rendering changes.
- The user inputs in the grid-like the start position and the end position and checkpoints (map of the environment). While finding the path it considers the checkpoints and the obstacles placed on the way.
- After the path is found, dragging the checkpoint or start or endpoints renders the path in real-time.

These are some of the functions of the agent. We can call the agent a rational agent because here the user inputs and the grid state functions as the precepts of the agent and the agents take the decision and perform the actions based on the perceived environment.

Mazes

The mazes are defined in the src/mazes folder. Each of the 3 mazes has its classes.
Input: Depending on the maze user chooses, the agent initializes the class of that particular maze.
Action: After the maze is initialized, each class takes the grid and other things as input and forms the maze on the grid. In the case of the recursive mazes, it takes the density as input too.

Finder

The finders are defined in the src/finders folder. Each of the 10 finders has its own classes.
Input: Depending on the finder the user chooses and the parameters of the finder, the agent initializes the class of that particular finder.
Output: The classes take in the current state grid and other parameters as input and tries to find the path to the destination. If a path is found, the finder returns the path and the order in which the nodes were visited or tested. After that, the agent start rendering the path depending on the speed and the stats are shown in the right bottom corner. If any path is not found, it is also shown in the right bottom corner.

Bot

The Bot class is defined in the visual/js/bot.js file.
Input: It receives the action performed by the agent as input and sets the display text
Action: According to the bot state set due to the input, it changes the innerHTML of the bot's content to appropriate messages. These messages disappear automatically after the set time if the close button is not pressed through setTimeout function.

Since the control shifts to these functions after the set time, to avoid clashes between one message to another, the original text and current text are compared. In this way, a message has the power to only delete itself. Result messages are given message IDs because they stay on screen for longer, and shouldn't hide a more recent result message.

View

The View class is in the visual/js/view.js file. It handles most of the inputs and outputs in the Raphael grid.

Panel

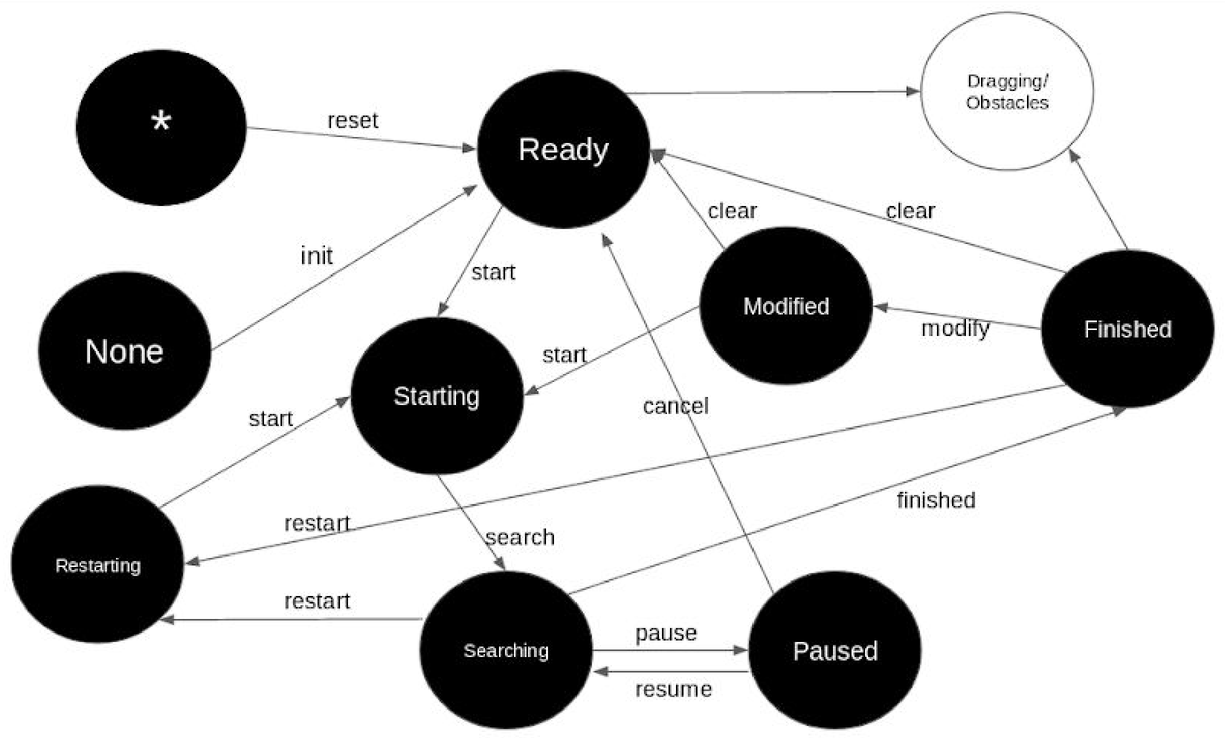
The Panel class is in the visual/js/panel.js file. This comprises of all the elements in the navbar. It handles the user inputs from the navigation also responsible for the initialization of the correct finder.

Guide

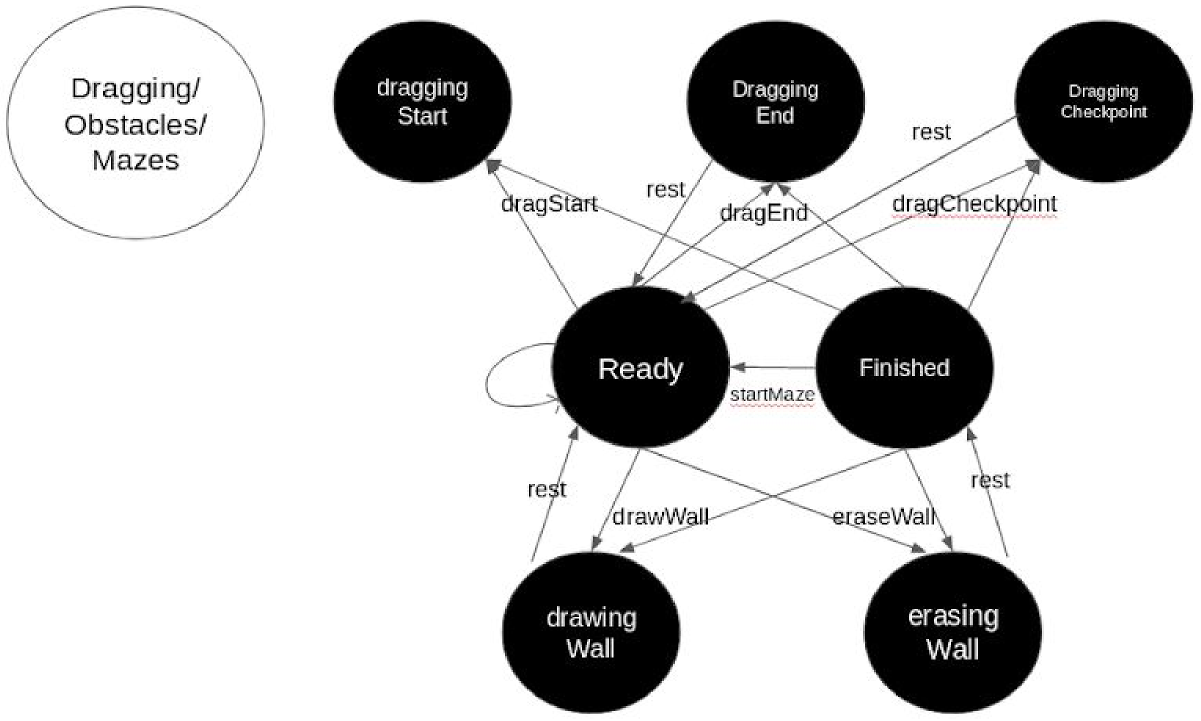
The Guide class is in the visual/js/guide.js file. Sweetalert's help is taken to display the tutorial aesthetically. It contains an async function backAndForth that decides which message and gif to display according to which step the user is in the guide.

Flow: State Diagrams

The application's architecture can be understood in terms of the state diagrams given below. At each state, the viable controls, objects on the screen, message displays etc. will be different. Therefore, the code structure will be easy to understand, the transition from one state to another, and any additional features can be smoothly implemented.



The state diagram for Dragging, Obstacles and Mazes is given below.



Features Implemented

Dynamic Control Buttons

Depending on which state the user is, the buttons dynamically change in order to show the viable options. This is a more interactive approach than simply disabling static buttons.

Searching

The searching can stay in 5 different states:

- Start search - This is used to initialize the search
- Pause search - When the search is ongoing, this is used to pause the search.
- Resume search - When the search is paused, this is used to resume the search again.
- Cancel search - When the search is paused, this is used to cancel the search.
- Restart search - When the search is complete, this is used to restart the search.

Clearing

The clearing is of 3 types:

- Clear obstacles - This is used to clear all the obstacles on the grid.
- Clear checkpoints - This is used to clear all the checkpoints.
- Clear path - After the path is found, this is used to clear the path.

Checkpoints

One can put the checkpoints using Ctrl+Click. You can drag the checkpoints also. You can put a max of 4 checkpoints. After you start the search, the TSP function finds the sequence of checkpoints that will lead to the smallest path in between the source to destination. Dragging the checkpoints also supports real-time rendering.

Real-Time Rendering

After the search is complete, one can drag the checkpoints, start points or endpoints. While dragging the mouse, the shortest path will be rendered in real-time. After the left click is stopped, it will also render the visited, unvisited and the tested nodes.

Travelling Salesman Problem

We have implemented a modified version of the Travelling Salesman problem. We have iterated through all the permutations of checkpoints to find the shortest path in between them.

Walls

One can draw walls while clicking and dragging in the grid. While finding the path, the finder avoids the walls. You can delete the walls by dragging on the walls.

Mazes

There are various maze algorithms added for maze tracks.

Finders

We have finders which try to find the shortest path between source and destination covering all checkpoints and avoiding walls and obstacles.
Currently, there are 12 path-finders bundled in this library, namely:

* AStarFinder` *
* BestFirstFinder`
* BreadthFirstFinder` *
* DijkstraFinder` *
* IDAStarFinder.js` *

* JumpPointFinder` *
* OrthogonalJumpPointFinder` *
* BiAStarFinder`
* BiBestFirstFinder`
* BiBreadthFirstFinder` *
* BiDijkstraFinder` *
* CollaborativeLearningAgentsFinder

The ones marked * find the approximate shortest path by various heuristics.

Heuristics

There are various Heuristics used by different finders

Heuristic Name	**Distance Returned**
Manhattan distance	dx + dy
Powered Manhattan distance	$(dx + dy)^2$
Extra Powered Manhattan	$(dx + dy)^{10}$
Euclidean distance	$dx^2 + dy^2$
Octile distance	(√2 - 1)*dx, when dx<dy
	(√2 - 1)F*dy, otherwise
Chebyshev	max(dx, dy)

References: Stanford GameProgramming

Guide

The guide at the start of the project gives an overview of all the important features implemented in an aesthetic manner using sweetalert. The user can go to the next step, one step back, or click outside the tutorial box to exit.

Speed

The speed of the bot, that is, rendering speed is controlled by setting the operationsPerSecond variable through the slider value. The interval is then calculated as 1000/(operationsPerSecond). If the Agent is not searching, there's no requirement to render anything. Else, the path is rendered after each interval as calculated. This is a recursive loop that ends when the destination is reached.

Message Bot TARS

The message bot TARS gives the user updates and tips throughout the project. It displays appropriate messages on the screen after various actions with varying display time. The bot can also be closed with a red X button.

Algorithms for path-finding (and their pseudocodes)

Searching

- **A* algorithm:**

  Based on: https://www.geeksforgeeks.org/a-search-algorithm/.

```
1.  Initialize the open list
2.  Initialize the closed list
    put the starting node on the open
    list (you can leave its f at zero)

3.  while the open list is not empty
    a) find the node with the least f on
       the open list, call it "q"

    b) pop q off the open list

    c) generate q's 8 successors and set their
       parents to q

    d) for each successor
        i) if successor is the goal, stop search
          successor.g = q.g + distance between
                              successor and q
          successor.h = distance from goal to
          successor (This can be done using many
          ways, we will discuss three heuristics-
          Manhattan, Diagonal and Euclidean
          Heuristics)

          successor.f = successor.g + successor.h

        ii) if a node with the same position as
            successor is in the OPEN list which has a
           lower f than successor, skip this successor

        iii) if a node with the same position as
            successor  is in the CLOSED list which
        has a lower f than successor, skip this successor
        otherwise, add  the node to the open list
        end (for loop)

    e) push q on the closed list
    end (while loop)
```

- **Bidirectional A***

  Extension on A* based on https://en.wikipedia.org/wiki/Bidirectional_search

- **Breadth-First Search**

  Based on:
  https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

```
BFS (G, s)                   //G graph, S source
let Q be queue.
Q.enqueue( s ) //Inserting s in queue until all its
neighbour vertices are marked.

mark S as visited.
 while ( Q is not empty)
//Removing that vertex from queue,whose
neighbour will be visited now
       v  =  Q.dequeue( )
      //processing all the neighbours of v
      for all neighbours w of v in Graph G
           if w is not visited
                    Q.enqueue( w )
     //Stores w in Q to    further visit its
neighbour  mark w as visited.
```

- **Bidirectional Breadth-First Search**

  Extension on Breadth-First Search based on https://en.wikipedia.org/wiki/Bidirectional_search

- **Dijkstra**

  Based on: https://www.programiz.com/dsa/dijkstra-algorithm

```
function Dijkstra(Graph, source)
for each vertex v in Graph: // Initialization
dist[v] := infinity // initial distance from source to
vertex v is set to infinite
previous[v] := undefined // Previous node in
optimal path from source
dist[source] := 0 // Distance from source to
source
Q := the set of all nodes in Graph // all nodes in
the graph are unoptimized - thus are in Q
while Q is not empty: // main loop
u := node in Q with smallest dist[ ]
remove u from Q
for each neighbor v of u: // where v has not yet
been removed from Q.
alt := dist[u] + dist_between(u, v)
if alt < dist[v] // Relax (u,v)
dist[v] := alt
previous[v] := u
return previous[ ]
```

- **Bidirectional Dijkstra**

  Extension on Dijkstra based on https://en.wikipedia.org/wiki/Bidirectional_search

- **Best First Search**

  Based on: https://courses.cs.washington.edu/courses/cse326/03su/homework/hw3/bestfirstsearch.html

```
Best-First-Search(Grah g, Node start)
    1) Create an empty PriorityQueue
      PriorityQueue pq;
    2) Insert "start" in pq.
      pq.insert(start)
    3) Until PriorityQueue is empty
         u = PriorityQueue.DeleteMin
         If u is the goal
            Exit
         Else
            Foreach neighbor v of u
               If v "Unvisited"
                   Mark v "Visited"
                   pq.insert(v)
            Mark u "Examined"
End procedure
```

- **Bidirectional Best First Search**

  Extension on Best First Search based on https://en.wikipedia.org/wiki/Bidirectional_search

- **IDA***

  Based on:
  https://en.wikipedia.org/wiki/Iterative_deepening_A*

- **Jump Point Search**

  Based on:
  https://github.com/kevinsheehan/jps

- **Orthogonal Jump Point Search**

  Based on:
  https://users.cecs.anu.edu.au/~dharabor/data/papers/harabor-grastien-icaps14.pdf

- **Collaborative Learning Agents:**

  Based on:
  https://courses.cs.washington.edu/courses/cse522/05au/awerbuch_kleinberg.pdf

Maze Generation

- **Random Maze:**

  It goes through all the grid and assigns it as a wall with prob 0.2.

- **Recursive Division Maze:**

  Based on https://gist.github.com/jamis/761525. It recursively divides the grid horizontally and vertically (alternating between them) and generates a Maze.

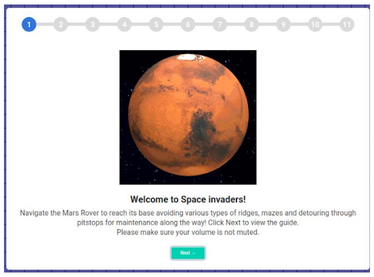
- **Stair Maze:**

  It draws a stair pattern with repeated upward and downward step shapes.

Usability Engineering Principles in Design

To make the user experience great, we designed the UI using usability engineering principles in mind.

Appropriateness Recognizability : The path finding app is designed such that all the features added are very relevant. It has a high degree of appropriateness.



Welcome to Space invaders!

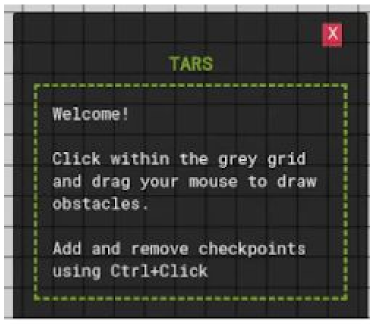
Navigate the Mars Rover to reach its base avoiding various types of ridges, mazes and detouring through pitstops for maintenance along the way! Click Next to view the guide.
Please make sure your volume is not muted.

Learnability: The application can be used by users easily, the controls are intuitive and the learning curve for this is very high.

Operability: The application's features are very simple and easy to operate. Dynamic controls also give viable options to the users.

User Interface Aesthetics: The user interface was created to increase the aesthetic value of the application. The colour theme was chosen to suit a space exploration mood.

Accessibility: Very accessible since anyone with the internet can use it.



TARS

Welcome!

Click within the grey grid and drag your mouse to draw obstacles.

Add and remove checkpoints using Ctrl+Click

Psychological/ cognitive factors

Since the message bot TARS gives continuous feedback to the user, they get responses, prompts and suggestions for how to go about. They will be confident about realizing every feature correctly.

Conclusion

In this application, we have implemented a variety of relevant features keeping good coding and design principles in mind. We have gained experience not only in the programming realm, but also in working as a team. Overall, we really enjoyed the mentorship program and the process. Thank you for this opportunity!