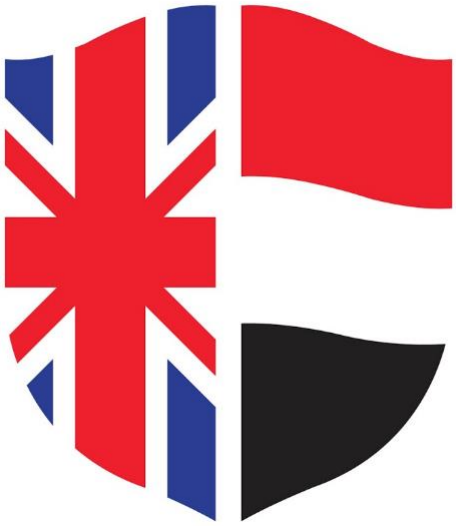


24CSAI04H

AI Planning for Robot Systems

Phase 1 Report

A Robotics Simulation of Krusty Krabs Restaurant

ID	Name	Email
224030	Amir Mohamed	amir224030@bue.edu.eg
226392	Merihan Ahmed	merihan226392@bue.edu.eg
228248	Omar Emara	omar228248@bue.edu.eg
218767	Sameh Ayman	sameh218767@bue.edu.eg

Table of Contents

Introduction 3

1. Objectives 3
2. Overview 3

Environment Design 4

Design Approach 5

3. Mapping 5
4. Path planning algorithms 5
5. Obstacle avoidance 6
6. Task Management 6
7. Protocol Compliance 6

Implementation Details 7

Challenges and Solutions 8

Testing and Results 9

References 10

Introduction

The customers dropped their money and went on a rampage to find who stole it. This is the theme of this project where we designed an intelligent robot agent inspired by Mr. Krabs from SpongeBob SquarePants that collects money dropped by customers at the Krusty Krab restaurant. Mr.Krabs and his robot clone need to gather the money quickly before the angry customers find him. To achieve this, our agents map their environments, implement key AI path-planning algorithms, and account for dynamic rerouting, obstacle avoidance, and multi-agent coordination.

Objectives

- Use inverse range sensor algorithm to allow our bots to sense and map our environment.
- Implement a dynamic environment using A* and Dijkstra's algorithms with multi-agents to navigate and evade dynamic and static obstacles in the most efficient manner possible.
- Handle dynamic obstacles in real-time through the logic of our algorithms.
- Create a communication protocol between the agents to divide, prioritize and gather all coins in the map.
- Implement prioritization of coins gathered and creating the path around it based on value.

Overview

This project involves designing an environment where a robot or an agent maps the environment and uses different search algorithms, namely A* and Dijkstra's, to navigate and traverse their environment. On their journey, they must evade and account for both dynamic and static obstacles in the form of angry customers, tables, seats and more. This is all done with the goal of gathering all the coins dropped in the map in a timely manner.

Environment Design

Both the environment and assets used were specifically designed to a *lore-accurate* replica of the Krusty Krabs restaurant from the original show. Everything except Mr.Krabs was created using Ibis Paint X.

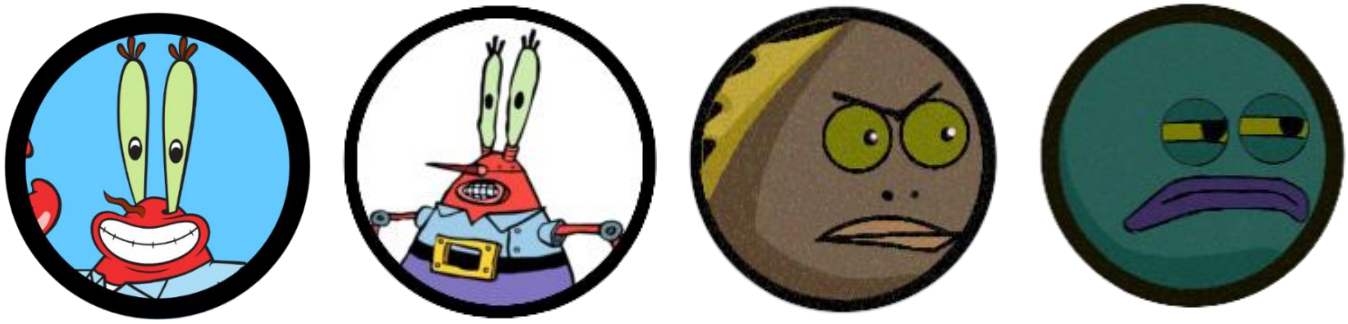


Figure 1: Mr.Krabs and angry customers (dynamic obstacles)

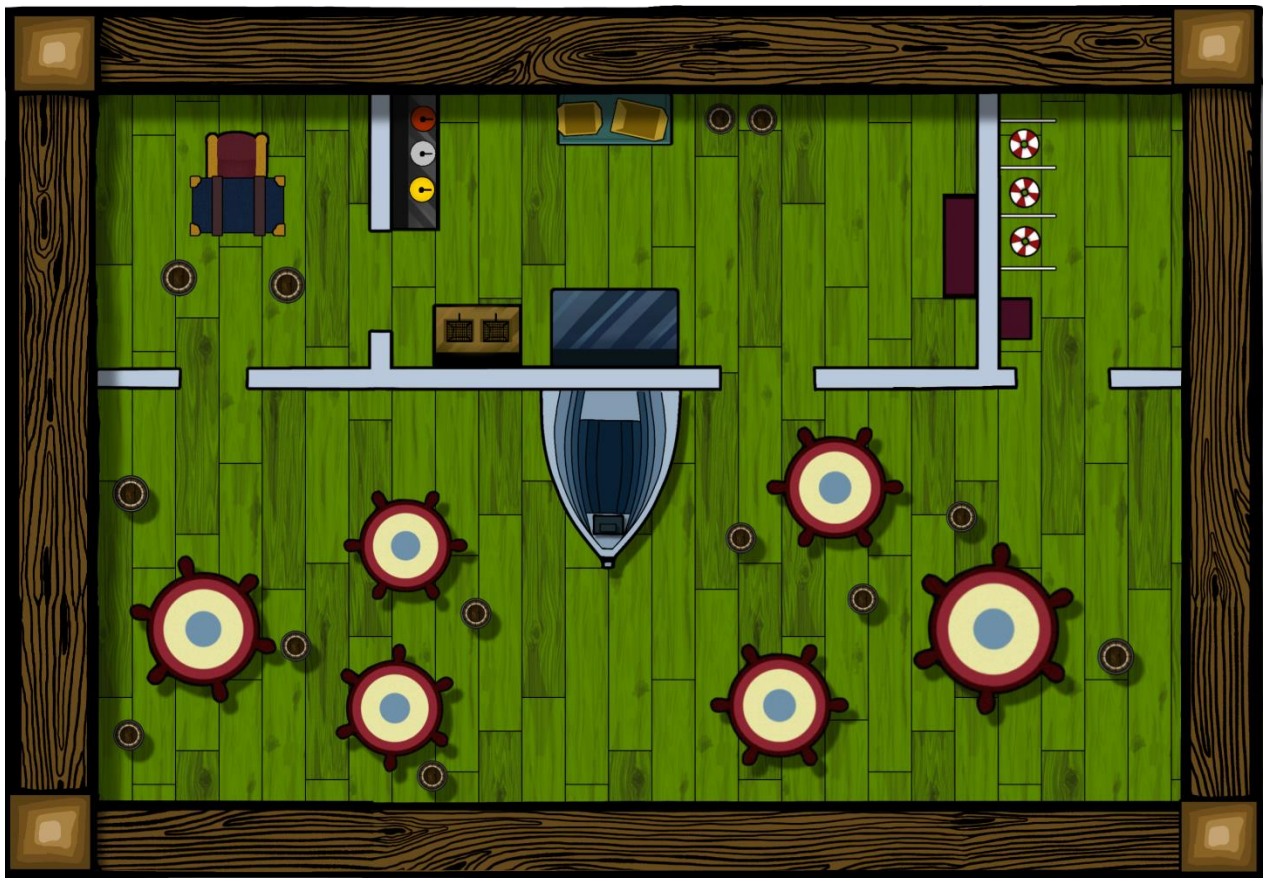


Figure 2: The Krusty Krabs main environment

Design Approach

1. Mapping

The environment is represented as a 2D occupancy grid of 2616 * 1816 / 2.2 cells. Using the mapping algorithm, the robot will be enabled to do movement within the said grid by sensing the obstacles, making a random move, and updating the occupancy grid. Each cell in this range is checked, and if an obstacle is found, further sensing along that direction is no longer encouraged. Inverse sensor model algorithm calculates the probability of each sensed cell whether it is an obstacle or not. The calculated probabilities get logged and then updated. This grid tracks probabilities of obstacles, shown visually as darker areas, while it displays the robot's position, path, and sensing range that is developing, an occupancy map used to navigate and avoid obstacles by the robot.

2. Path planning algorithms

- A*: We created the algorithm with the capability of handling dynamic obstacles in mind to allow it to get the best path from a starting point to the goal. It achieves its goal by avoiding both static and dynamic obstacles using heuristics, which give priority to cells closer to the target and initialize a queue to explore the possible paths. It then scans through the neighbors of each cell-skipping those outside bounds or occupied by moving obstacles or another robot-checking if they provide a shorter path through them. In case it does, it updates the records for that cell and inserts it into the queue. Once reached, the algorithm reconstructs the path. If unreachable, it just returns an empty path [2].
- Dijkstra: Much like A*, this algorithm finds the shortest path from a given starting location to a defined goal in the occupancy grid while avoiding static and dynamic obstacles. This routine first inserts the start location into a priority queue and then iteratively searches its surrounding cells. Around each considered cell, it looks into its neighboring cells, skipping any that are occupied by a static obstacle or a moving entity, or another robot unit. It keeps track of the lowest known cost to reach each cell and updates this cost if a cheaper path is found [1].

3. Obstacle avoidance

- Dynamic obstacles: the dynamic obstacles were designed to move in one direction until it hits an obstacle and then take a 90-degree rotation with occasional random chance to switch into another random direction to introduce further unpredictability in the environment.
- Static obstacles: are impassable areas that the navigation algorithm avoids, and these include the tables, sets, room items and walls.
- Agent's sensing: a mechanism for the agent to discover its space. It has a configured maximum range and works by calculating the Euclidean distance from the robot's current position. It also ensures that robots do not scan any out- of-bound cells avoiding unnecessary computations.

4. Task Management

- Target prioritization: each target is spawned with its own priority based on its cost, depending on said cost, the agent may take prioritize a different path to guarantee the highest total points with each algorithm.

5. Protocol Compliance

- Multi-agent coordination: the two agents coordinate between themselves by dividing the rewards amongst themselves. The rewards are first sorted based on priority, then the rewards get divided by the number of the agents. Each agent is the tasked to navigate a path to the coins from the highest to the lowest priority on their list as they operate simultaneously to reach their targets while avoiding any potential obstacles and treat each other as dynamic obstacles so they do not interfere with each other's paths.

Implementation Details

To implement the project, we have used a wide range of development and design tools. For the design part we used Ibis paint X to design the assets that later were used for the app. For development, we ended up using VS Code and Jupyter notebooks.

The app's code was written in Python and PyGame which is a popular python library for developing games. Many internal python modules such as math, random and time were incorporated as well. The code was written in a modular fashion making extensive use of best coding practices and was beautifully broken down into manageable functions and classes.
On a software level, we also employed NumPy for managing the occupancy grid and general matrix manipulations, Priority Queues from Python's native queue library in the implementation of the A* and Dijkstra algorithms.

Challenges and Solutions

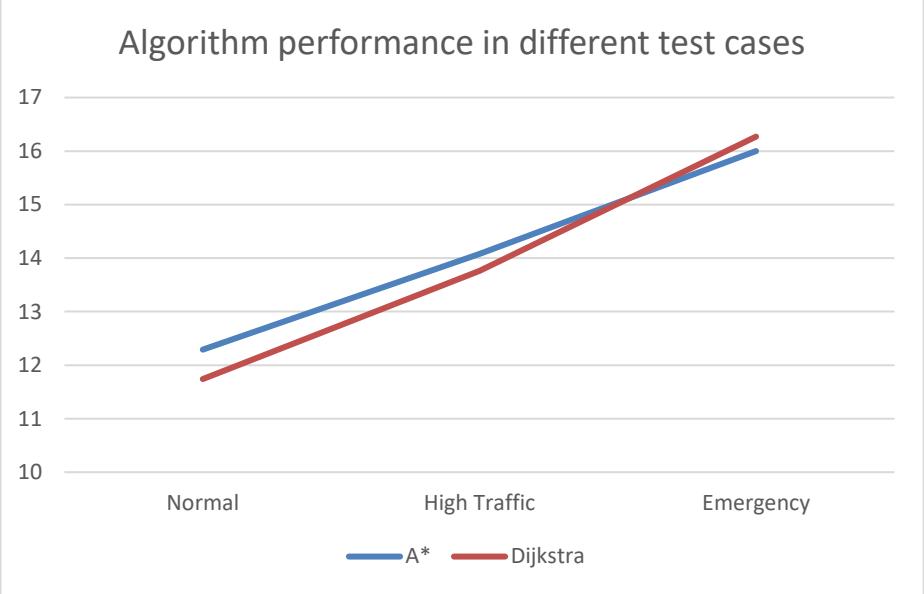
Implementing this simulation was indeed a tough task, we have gone through many challenges that we were not anticipating while planning the development process. The following are some of the interesting hurdles we faced:

Challenge	Solution
Slow mapping	We tweaked the agent's speed and max range constants in the move and sense methods of the Robot class respectively to have extra number of cells per move.
Agent getting stuck while mapping	The agent uses a retry mechanism with a maximum number of attempts to find a valid move. If it fails to move, it stops, preventing infinite loops and reducing unnecessary movements.
The two agents colliding	Each agent considers the other as an obstacle hence avoiding collisions and reusing the obstacles avoidance logic for better code architecture

Testing and Results

We presented a couple of test cases to the agents – normal, high traffic and emergency. Each test case with different number of dynamic obstacles (4, 8, and 12 respectively). We then measured the time (in seconds) it took for each of the planning algorithms to reach the targets, one target each for each test case. The results are shown below:

	A*	Dijkstra
Normal	12.29	11.74
High Traffic	14.08	13.76
Emergency	16.00	16.27



The results aren't totally conclusive as there is the random factor of dynamic obstacles movement as well as an inherit bias in the direction they choose to go to, which is based on their starting points. However, from the data we gathered we can say that A* performs slightly better than Dijkstra for most cases, while Dijkstra prevails ever so slightly in the emergency case.

References

[1] GeeksforGeeks, “Dijsktra’s algorithm,” GeeksforGeeks, Nov. 19, 2018. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

[2] R. Belwariar, “A* Search Algorithm - GeeksforGeeks,” *GeeksforGeeks*, Sep. 07, 2018. https://www.geeksforgeeks.org/a-search-algorithm