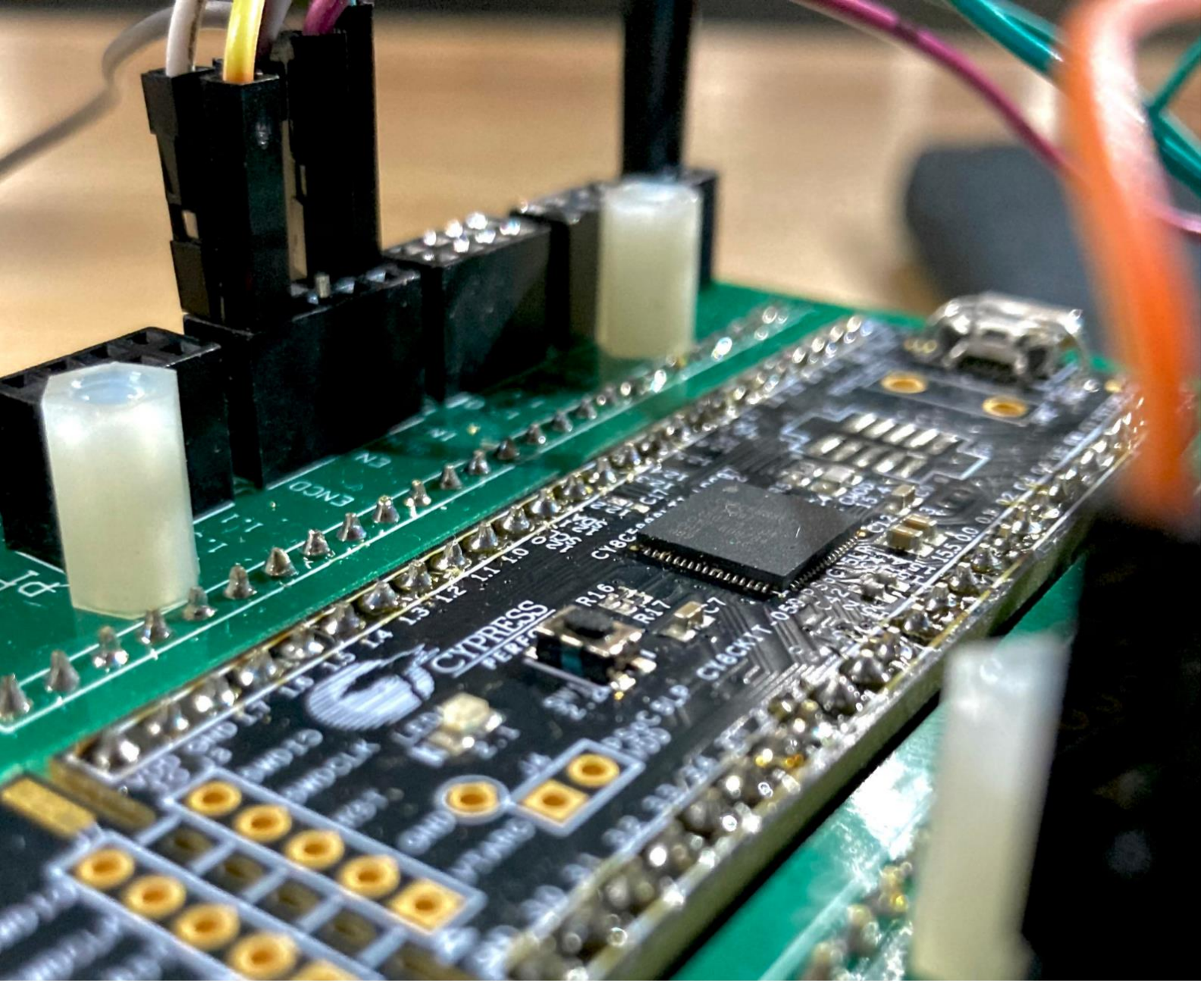

COMPSYS 301
Project Report - Software

Group 13

Bill Yang

Cecil Symes

Eric Zhao

Nikhil Kumar

Abstract

The goal of this project was to emulate the well-known arcade game “Pacman”, using a simulated robot and food pellets.

This involved creating many components, such as the creation and use of the shortest path algorithm, pathfinding through a maze, detecting intersections, and ensuring that this was all done without straying off the valid path. Various shortest path algorithms were tested and the most efficient algorithm, A*, was chosen. The design was aimed at being split into three conceptual parts: the A* algorithm, the robot controller, and the robot itself.

Problems encountered include the translation between coordinates and relative robot commands, detection of intersection type, and dealing with very tight turns. Intersection detection was initially able to detect different types of intersections but became problematic as more complex three-way T intersections and four-way cross intersections were encountered. The A* algorithm’s output was not useful without being translated into commands that could be executed relative to the robot’s orientation, so translation between coordinates and relative robot commands had to be handled. Finally, a fixed turn radius behaviour was used, which led to issues with the robot staying on the original path when presented with two turns very close to each other.

The implementation overall had strengths with its simplicity and modularity. Some drawbacks to be considered for future work include the fixed turn radius, the lack of granularity when controlling the car between nodes, and that by pathfinding entirely during the initialization of the program, a single error can render the entire list of instructions useless.

Table of Contents

Design

Planning & Ideation

Conceptually, the initial idea was to have three separate parts:

- Shortest path algorithm (A* algorithm)
- The simulated robot
- The robot controller

The approach taken treats the maze as a graph (Figure 1). In the ideation period it became clear that if the use of getting the current position was not an option, the robot could be treated as navigating a graph. The written code uses coordinates and arrays, but the operating principle of the system is that the robot is travelling across various nodes along edges. In the context of the maze, every corner, intersection, and dead-end can be considered a node vertex, and every valid path between nodes (turns/intersections) are edges (Figure 2). When on an edge, the robot will use a line following behaviour. Once on a node, the robot will transition to a robot control state, where an action must be purposely taken, such as turning, going straight through the intersection, or turning around 180 degrees.

The A* algorithm was to take in a map, a starting coordinate, and a destination coordinate, and append to a list of coordinates indicating the path to be taken.

The simulated robot was to blindly follow the line, alerting the controller upon reaching an intersection.

The controller was to interface with the robot and the A* algorithm. It would provide the A* algorithm with the coordinates and map, translate the output coordinates to robot relative instructions, and give the robot the next required command upon reaching an intersection.

Upon implementing the code, changes had to be made. The A* algorithm is detached from the controller and robot; however, the controller is heavily connected with the robot. The robot itself checks every tick to see whether it has reached an intersection, and upon reaching one, executes the next command from the list of actions to be taken.

The final implementation does not have the robot recall the A* algorithm at any point, with the entirety of the instruction list created in the virtualCarInit() function (Figure 6) (Figure 7).

MATLAB Analysis: Choosing an algorithm

To find which of the shortest path algorithms should be implemented, their time costs were compared in MATLAB.
As shown in the Appendices, both Breadth-first and Depth-first took 47 steps to reach the destination. The A* algorithm took only 29 steps, 22 less than BFS (Figure 3) and DFS (Figure 4), making it the most efficient algorithm, and the algorithm of choice.

It was concluded that A* will guarantee us to obtain the shortest possible path (Figure 5). However, BFS and DFS could be easily viable as they are more efficient and simpler due to

the maze construction. Therefore. if the maze is straightforward, best first or something similar should be used, when the maze is difficult, A* will demonstrate its advantage.

Design implementation in-depth:

Converting the given map

The given map represents walls with a 1, and valid paths with a 0. The A* algorithm expects a map that represents valid paths using a 1, so a function was written to create a new inverted map for the A* algorithm.

Additionally, an intersection map is created for the FollowInstructions() function. The ConvertToIntersectionMap() function indicates all intersections and turns with different numbers, so the controller can identify where certain actions are valid or not valid.

Running A* to all points of food

To navigate level 2, the A* algorithm is called at the beginning with the robot's starting position and the first pellet and repeated with successive pellets.
A caveat was that the pellets needed to be eaten in the order provided. The updateOrderMap() function creates a new map, replacing each food pellet with a wall. This map is fed to the A* algorithm, and when the commands have been successfully generated, the function is called again with the number of the food pellet that has been planned for, and the "eaten" food pellet is restored back to a path. This ensures that food pellets cannot be eaten out of order.

The output of the A* algorithm appends coordinates to a global vector of Pairs. This allows successive calls of the A* algorithm to stack, making sure not to override old coordinates.
This presents an issue, as the robot cannot call the global variables that store its position, and it is difficult to estimate the current coordinates of the robot based on speed and distance. This is where the node-based approach of the design comes in.

FollowInstructions() is the function which takes the output of the A* algorithm (a vector named algoOut) and converts it turning instructions for when the robot needs to decide at an intersection. This function is the final component required during initialization for the robot to successfully travel from one point to another using the shortest path.

Running robot automatically

After all initialisation, the robot switches to the deployed mode and runs the virtualCarUpdate() (Figure 7) function. The robot proceeds to default to the line following code provided initially with the project. The line following code does not have any logic for detecting and dealing with intersections, so the DetectIntersection function runs every tick. This checks for an intersection by checking if the centre sensor is active, and more than one of the right or left sensors is active. It also detects if a dead end is detected by keeping track of the previous sensor data. DetectIntersection allows the robot to transition between states of following a line to executing a specific action at a node.

Once an intersection is detected, the robot must execute an action. The robot looks at the next instruction in the CommandList, and then conducts the appropriate action to take at the node; either to turn, go straight, or do a 180.

Line following and fixed turn making

Line following logic was provided with the project and was satisfactory for the robot to use when travelling between nodes.

Due to the nature of the code, turning is implemented on a “per tick” basis. Flags are raised at the start of turns, and slight linear and angular adjustments are executed each tick until the desired angle is reached. The turning system doesn’t use the sensors at all, meaning that the turning radius is fixed, and the robot will turn irrespective of any line detection until the desired angle is reached. This proved to be effective in the test map, however, the final given map contained turns that were one unit apart, which presented issues covered in the “Drawbacks” section.

Strengths and reasoning behind the design

The design proves to be an effective, efficient and simple approach to this design challenge. The relatively small amount of complex logic required to successfully have the robot navigate the maze is what led us to this design. Some strengths of this design include:

- Runtime performance
- Coordinate Agnostic
- Simplicity
- Simple API integration

The pathfinding function is always calculated at initialization time, and not runtime. The advantage of this approach is that the pathfinding code does not need to be called in the virtualCarUpdate() function. This keeps the execution time per tick of the virtualCarUpdate() function consistent and means that calculating distance, or taking a turn, can be done consistently. A large amount of initialization computation means that rather than have computation done during the running of the robot, which could mess up important timing/detection, it is all done beforehand. Not calling the pathfinding logic again also means that the robot never needs to pause and calculate a new path and its associated commands, making the robot travel quicker and smoother.

A Quality of this design, and an aspect focused on during implementation was the separation of concerns and modularity within the design. There is no real object-oriented design in the C++ code, as an imperative coding style is used in C programming, yet the design aims for separation of concerns. This was done by having defined global variables of which functions would write to, and other functions would read from. These variables would act as the data intermediaries from one function to another, hence acting as interfaces between elements. Anyone module could be replaced if they operate correctly on the interface variables, allowing for modularity in the design.

One constraint was that the robot could not access its current coordinates using the provided global variables for pathfinding. As a result, the code was made to coordinate agnostic as possible. The use of the node-based operation method means the robot is not relying on any coordinate inputs or checking coordinates for any operation during runtime.

The design also fitted well into the tick-based nature of the robot, and as such, made lower-level API integration smooth; partly due to the modularity of the code, but also the software design allowed us to integrate tick-based turning into the system with ease.

Drawbacks and limitations of Design

- Fault Intolerance

By choosing to generate all instructions at the beginning of the program, there is zero room for error at runtime. If the robot was to incorrectly detect an intersection or simply not detect an intersection, then the entire robot command list is out of order and the robot will execute wrong instructions, ruining the entire run. During testing, there was confidence that the intersection detection routine was robust enough, however, there was no method of detecting false positives and therefore the robot is prone to out-of-order issues. On top of this, there is no logic for detecting any faults that might have occurred during runtime or any means to remedy out-of-order or other related issues.

- Granularity of Control is diminished

Every single corner and intersection are treated as a nodes where the robot can conduct an action. There is no control over the robot when it is travelling along an edge, as it follows lines between nodes by default. This means there is a diminished level of control over the robot, and it cannot conduct actions which would otherwise be beneficial. One example of this is when a food pellet is in the middle of a path between intersections and the robot must then turn around to reach the next food pellet. For the robot to conduct this set of actions, it must travel the entire length of the path, to then reach the intersection node to do a 180 action, contrasting the more efficient and intuitive case of turning 180 once it has reached the food pellet. Whilst the robot can complete the required actions, having actions only occur at intersections makes the robot less robust in unfavourable map layouts.

- No control system for turning

The current scheme for turning the robot is to simply transition the robot to the action state, ignore all sensor inputs, and change the angular and linear positions every tick for a fixed amount until the desired cardinal direction is reached. This strategy of turning is based on assumptions that:

  - The map only has one fixed radius for every corner and intersection.
  - The map has no turns in quick succession.

This is the “Achilles heel” of the design. Whilst intersection detection is ignored during a turn, the intersection detection subroutine could detect false positives or negatives due to subpar turns. This can be seen when there are tight turns (1 unit/hairpins), which were not present in the test map but were present in the final map. The fixed turning radius is not able to take these turns as cleanly and there is a chance that the robot will miss an intersection, or run off the track and detect a “dead end” simply due to the turn radius being too small for the fixed turn radius. This leads to the out-of-order issue mentioned earlier and with no-fault recovery means the entire run has been ruined.

Hurdles & Roadblocks

Converting from Coordinate Pairs to Robot Commands

An issue discovered during implementation was that the algorithm output made sense from a bird's eye/top-down view, yet the robot cannot run on these coordinate values alone. It also cannot have an idea of the compass direction it faces, as it could not call the current car angle variables to check.

As such, a function was written that looks at the list of coordinates outputted from the A* algorithm and creates a list of orientation relative commands for the robot to follow.

Detecting the Type of Intersection

The initial implementation of intersection detection was able to detect the type of intersection. This enabled the robot to see a turn, and continue line following through the corner.
However, this presented issues when it came to detecting T intersections. No matter which entrance of the T intersection was taken, the robot would always detect a turn immediately, and it would take a few ticks to correct the assumption from turn to T intersection. This was because the robot will not always approach a T intersection perfectly, thus one side of the sensors array will be activated first, indicating a turn. By the time the robot corrected the assumption, it would be facing around 45 degrees in a direction.

Brainstormed solutions included implementing a 45-degree turn, but the chosen solution was simply to treat turns as intersections. This created its own issues with respect to the turning code implemented and meant that the generated command list would be much longer. Some benefits included that the intersection detection logic became much simpler, as every single turn and intersection was treated equally and no differentiating between the intersection types was necessary. Because the Command List would contain actions for any intersection or corner, it does not matter the type of intersection that the robot is at, simply if it at an intersection or not. This design decision removed this intersection type detection issue entirely.

Future Work

Turning with feedback to deal with Hairpin Turns

The test map given contained no four-way (+) intersections, nor did it contain two turns within one unit of each other. However, the final map did contain both. The + intersection was not an issue due to the chosen implementation as previously mentioned, where all intersections are detected immediately and subsequently dealt with. However, the turns being within one unit of each other posed an issue. These "hairpin" turns did not allow the robot sufficient time to turn and re-centre before reaching the next turn, meaning that the robot would run off the track or miss the second turn fairly often.

A simple change would be to simply move the robot forward until the turning pivot is on top of the turn and then to pivot on the spot. Testing could be done to get a value for the distance needed to position the robot's pivot above the turn.

Another more robust solution would be to implement feedback during the turn, dynamically changing the turn radius depending on which sensor is being activated and the strength of the sensor. This will allow for better turning and reduce the out of order errors that are likely to occur due to false positives intersection detections from imperfect turns.

Implementing a more granular pathfinding system, error detection and correction

Currently, the robot doesn't attempt to correct itself if a mistake is made, and a single mistake will invalidate the rest of the command list. Other issues as discussed is the loss of granular control over the robot.

The calculation of distance would be the next step for improving the robot's robustness and would help with granularity of control as well as error detection. A distance per path parameter and timing the robot as it travels along a path between nodes can be used to remedy some of the issues discussed.

- Granularity of control
    - When the robot must conduct an action which requires action along a path, making use of a timer and computing the distance travelled along a path would allow the robot to conduct a certain action once it reaches a certain distance along the path, rather than at the end of a path. This change offers control at all points in the maze, rather than at intersections.
- Error Detection
    - If the robot has yet reached an expected intersection after a certain distance travelled, or if an expected intersection has been hit too soon, or any other discrepancy between expected and actual distances, or sensor data, then there is a certainty that an error has occurred. This detection of error will allow the triggering of a recovery routine. This routine would recover the position of the robot and discard the out of order instructions, recalculate new instructions which would return the robot back onto the right route.

Conclusion

Many different design choices were considered during the implementation of this project. Treating all intersections and turns equally as nodes provided an efficient way to avoid differentiating between intersection types. All pathfinding was done prior to moving the robot, ensuring that the robot could execute smoothly during runtime, but also meant that a single mistake could set the robot onto the wrong path. There remains room for improvement, but overall, the team believes a good compromise between simplicity and effectiveness was found, and the solution offered worked well.

Appendices

Work Distribution

Hours worked →	**Bill**	**Nikhil**	**Eric**	**Cecil**
Planning & Ideation	3 Hrs	4 Hrs	3 Hrs	5 Hrs
BFS in MATLAB	N/A	1 Hrs	2 Hrs	N/A
DFS in MATLAB	N/A	1 Hrs	2 Hrs	N/A
A* in MATLAB	N/A	6 hrs	6 Hrs	N/A
C++ Port	N/A	5 Hrs	N/A	N/A
Algorithms Research	N/A	4 hrs	N/A	N/A
Detect Intersection	N/A	.5 hrs	1 Hr	3 Hours
Line Following	N/A	1 Hrs	N/A	.5 Hours
Robot Control Instructions (Turns etc.)	N/A	5 hrs	N/A	.5 Hours
Intersection Map	6 Hrs	.2 Hrs	3 Hrs	N/A
Follow Instructions	10 Hrs	.5 Hrs	N/A	N/A
General Debugging	3 Hrs	5 hrs	2 Hrs	8 Hours
Group Meetings	5 Hrs	5 Hrs	5 Hrs	5 Hrs

Figures

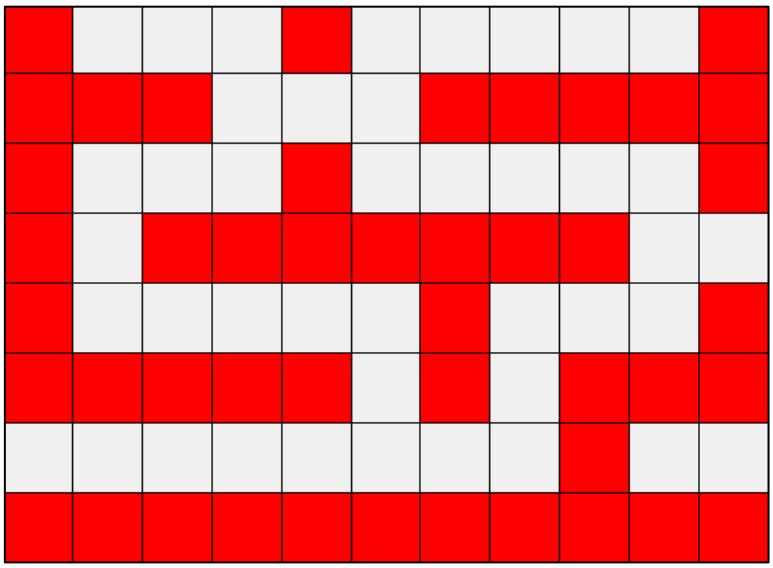

Figure 1: Maze View

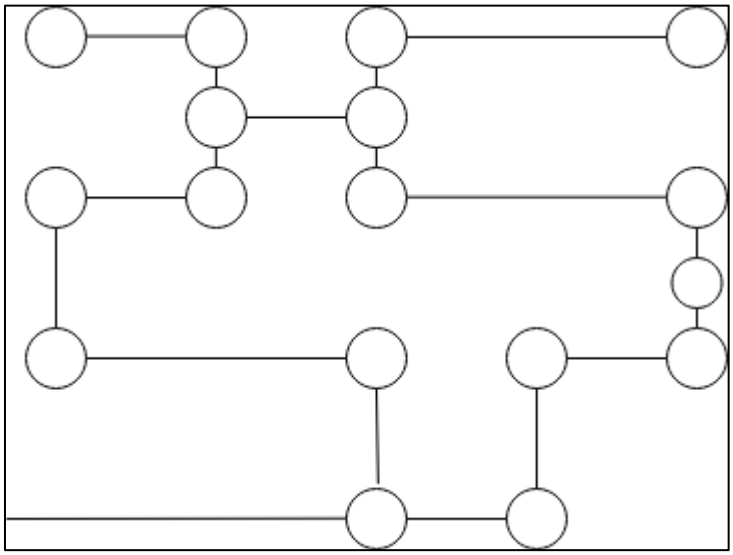

Figure 2: Node View

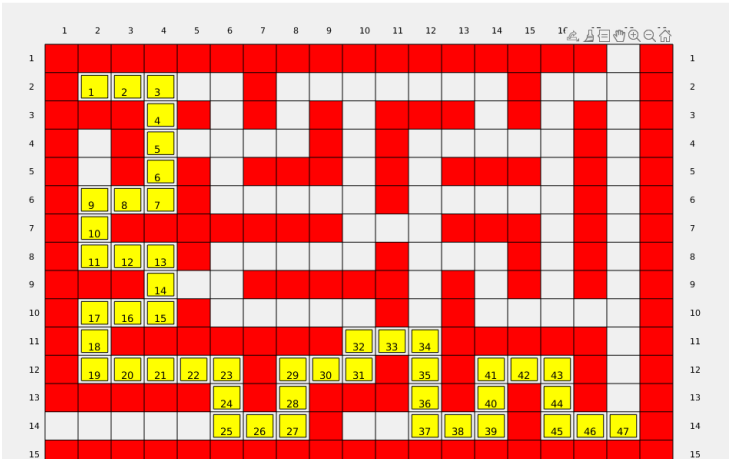

Figure 3 : BFS

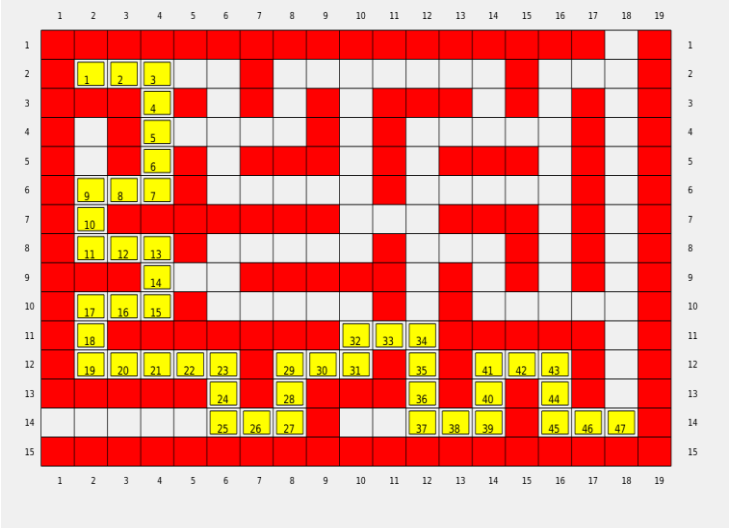

Figure 4: DFS

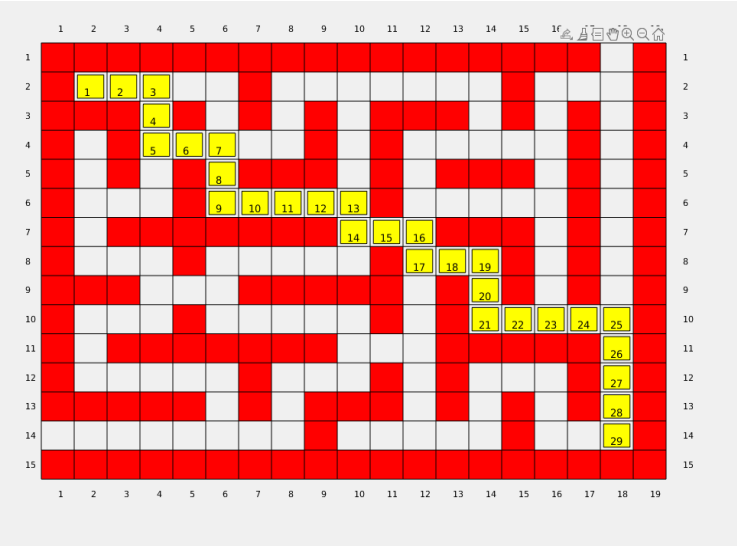

Figure 5: A Search*

Operation Flowcharts

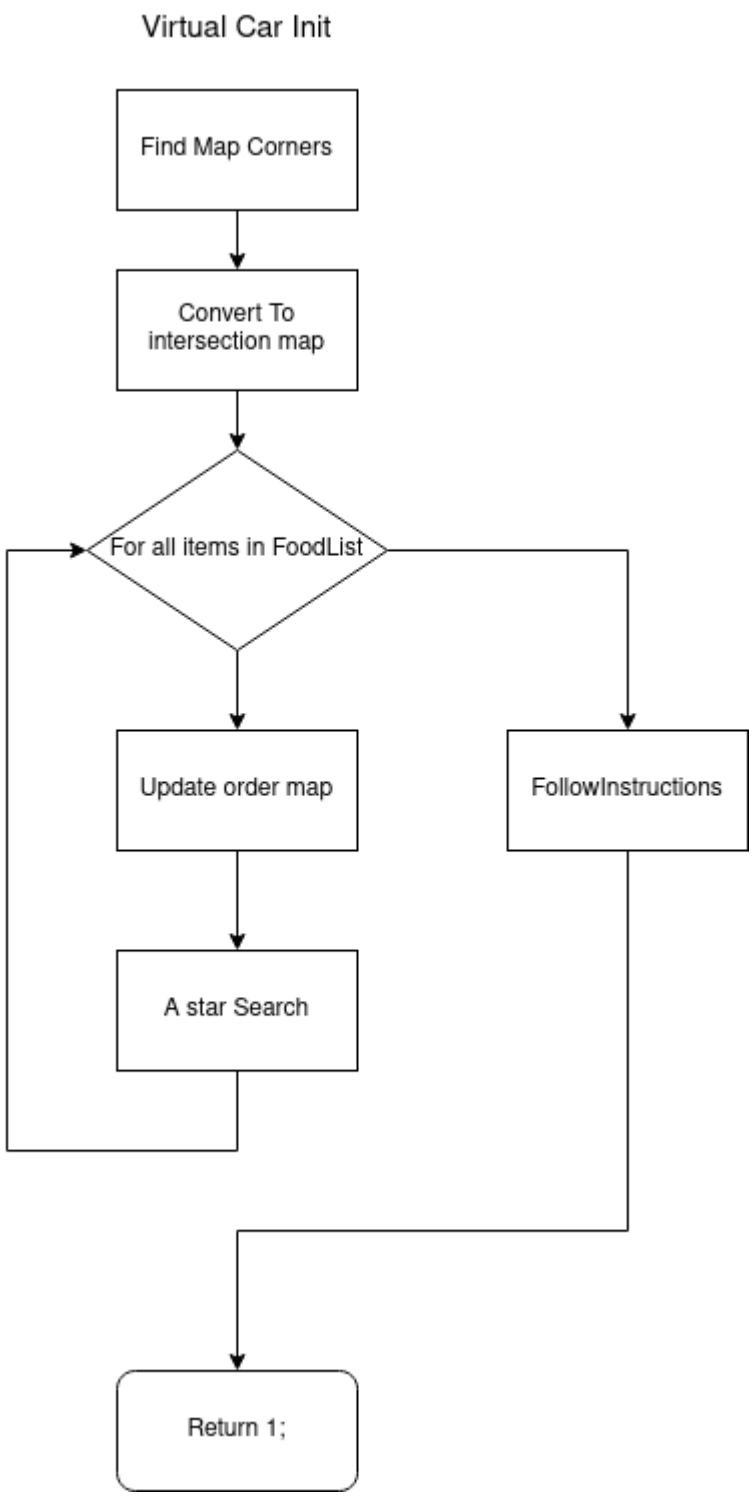

Figure 6 Virtual Car Init Flowchart

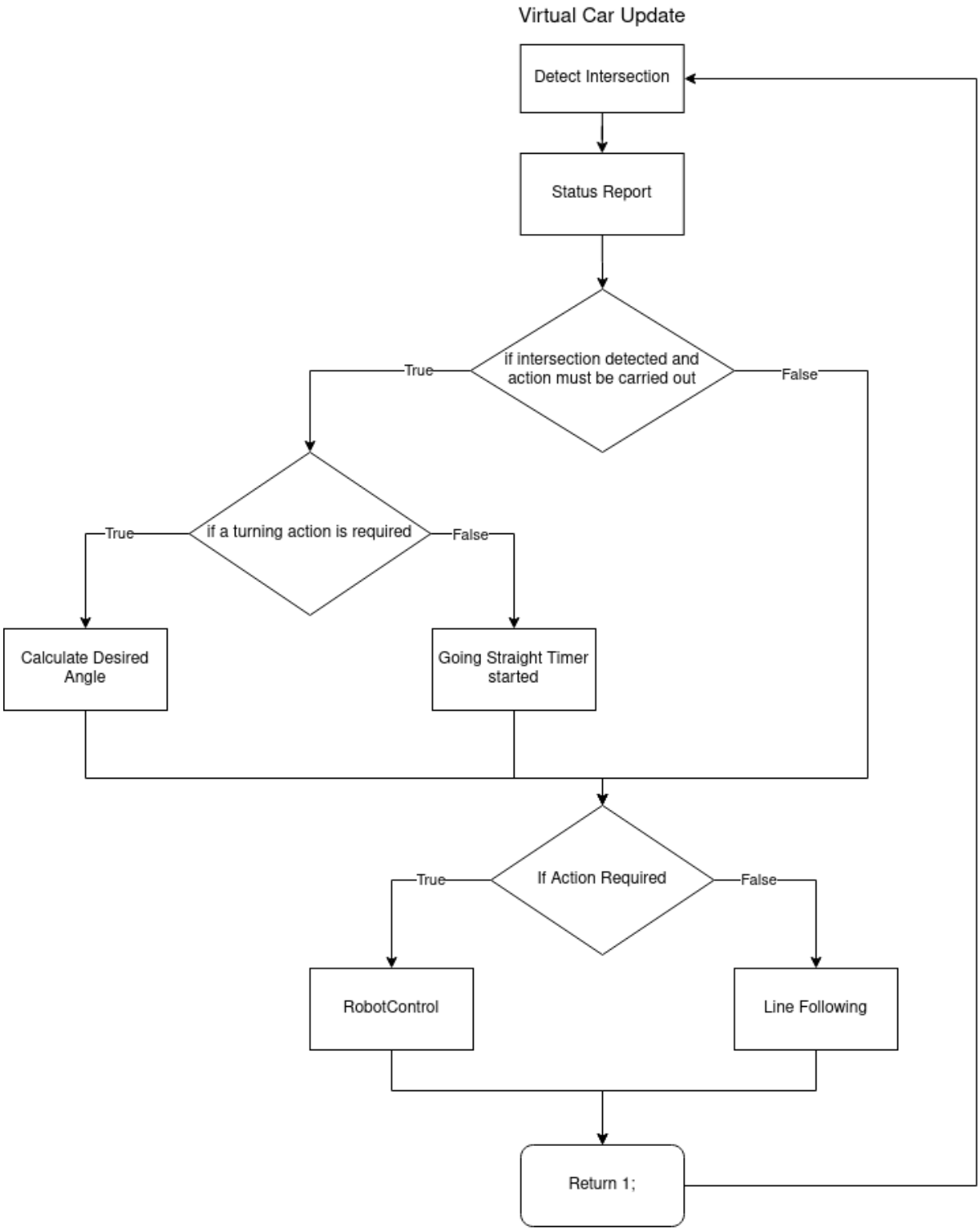

Figure 7 Virtual Car Update Flowchart