Group 44

SWEN30006 Final Report

Initial Problems Found

The strategy of moving along the left hand side of walls in the given `AIController` class works perfectly when dealing with an unknown maze map with destination on the outer wall, which is the fastest approach and guarantee not to get lost. It acts based on the current state only which is mindless. Since the map of walls, entrance and exit cells can be accessed by calling `getMap()` function, it could be optimised further from there.

Besides its own strategy, we analysed the following problems additionally:

1. It has no memory of the overall outlook of the entire map and explored areas on the map, and no prediction or plan for what to do next.
2. It lacks the ability of retrieving the keys and repairing itself, which makes it harder to complete the task for all the maps.
3. It can not escape if the exit tile is away from the walls.

Final Solutions

In order to accomplish the task and achieve a better outcome, the solutions we came out with are:

1. To implement a `MapRecorder` class to record and update the map constantly, using the initial map including the locations of walls by calling `getMap()` function of the car's radar which detects a 9x9 tiles area around it.
2. To use a `Pipeline` structure to calculate and optimize paths based on a given set of destinations. The pipeline has `AStar`, `AddDestinationPair`, `AvoidWall`, `RemoveRedundantPath` and `SimplifyPath` classes as its steps in order.
3. To utilize `StrategyManager` to manage the strategy pattern, and to switch between different strategies, including `EscapeStrategy`, `RetrieveKeyStrategy` and `RepairStrategy`, throughout the trip based on the current situation.

Map Recorder

`MapRecorder` class follows the *Information Expert* principle. It stores all information about the known map at the current point of time, including the full map obtained when the game starts. Besides the type of `MapTile`, enumeration class `TileStatus` is created to determine whether a tile is searched, unsearched and unreachable (wall tiles or area enclosed by walls). `MapRecorder` identifies unreachable areas by using Depth First Search algorithm (DFS) during the initialization process and updates the map whenever a new area is discovered by the car. For the need of AI strategies (collecting keys, repairing and escaping), coordinates of keys, health traps and exits will be recorded. Additionally, a method `coordinatesToExplore()` is provided to calculate a set of evenly distributed coordinates that covers all the unsearched places of the current map.

Path Finding Algorithm - A*

Instead of a strategy that moves along the walls in the given AI controller class, `AStar` class is considered as a more powerful approach for path finding. It uses Manhattan Distance as heuristic values to guide its search, which gives the shortest path and add weights for different situations. For example, a large weight is added when moving through *Lava* Traps and making unnecessary turns in a path.

Moveover, A* algorithm provides a total cost for a particular path. This helps to determine the evaluation function that decides when `repairStrategy` (navigating the car to nearest *Health Trap*) shall take over as the activate strategy which is explained below.

Pipeline

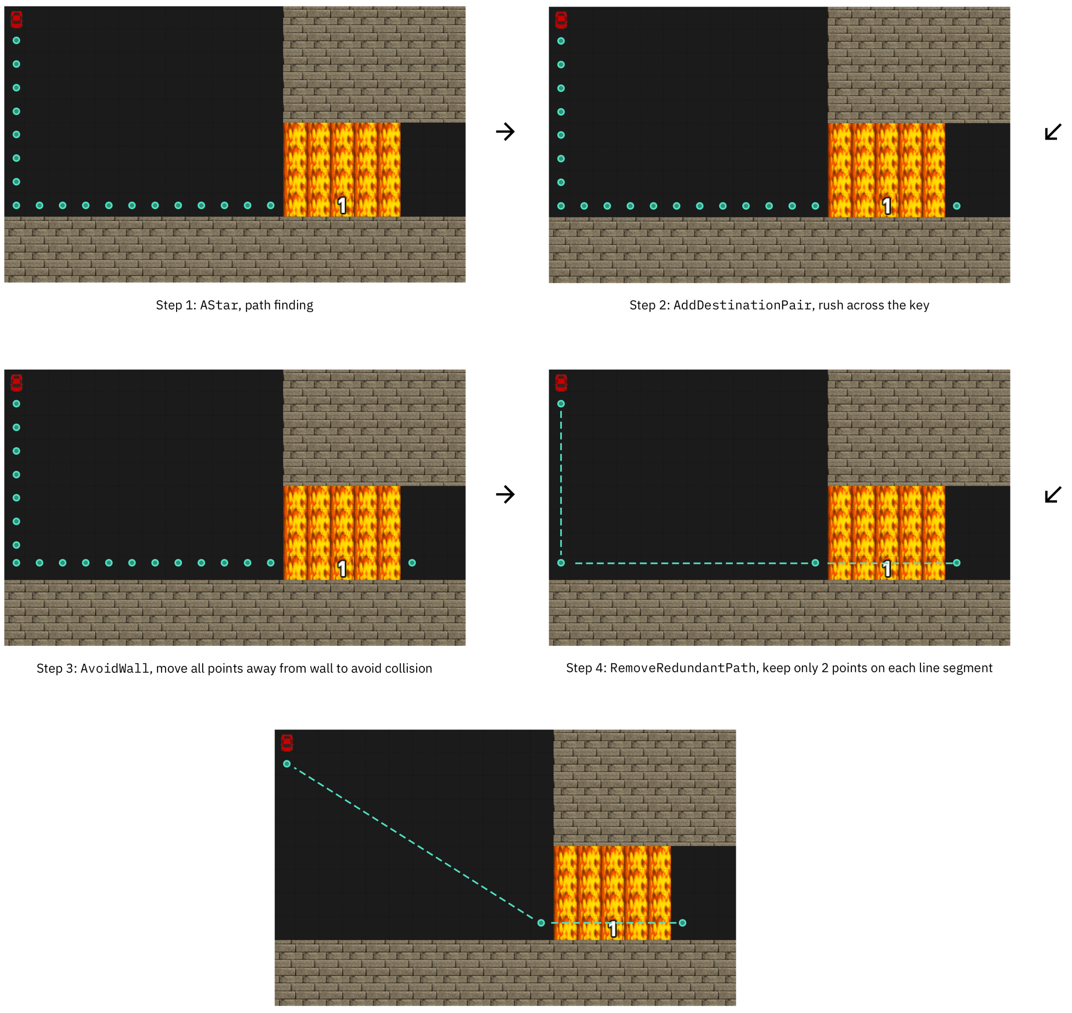
After the fundamental implementations completed, we found that when getting the paths, a series of functions are referenced:

1. Find the closest target and generate the path using A* algorithm
2. If the selected target is only partial, add a second target. The aim is to collect the next key. We try to generate a pair of targets that make the car to rush through the lava trap.
3. Prevent the car from colliding the wall by adjusting the path slightly away from the it
4. Remove the collinear coordinates in the path
5. Simplify the path by removing unnecessary turnings

To create a smooth streamlined processing of coordinates, pipeline structure is used to create a trajectory which reduces the redundancy of referencing a series of functions. All the functions are separated into several classes to assign responsibilities clearly. Steps inclueds `AStar`, `AddDestinationPair`, `AvoidWall`, `RemoveRedundantPath`, `SimplifyPath` are used in the pipeline.

Firstly, `AStar` is used to compute the most optimal path among all destination candidates. Then, `AddDestinationPair` appends the corresponding coordinate if the path decided by A* ends on one side of the key. `AvoidWall` is used to adjust the given path by adding an extra safety distance to prevent the wall collision. After that, `RemoveRedundantPath` takes out the redundant collinear points along the path, as more points bring more pauses. Finally, `SimplifyPath` finds the clustering coordinates, and simplifies into straighten path to minimize turns.

Additionally, `Step` interface is defined for `Pipeline`. This is to generalize the interface between steps, and to allow a more flexible combination of steps in a pipeline. For instance, the class `AddDestinationPair` is added in a later stage of development, with the uniform `Step` interface, the insertion of a new step is made easy, and it does not require any change to the code code of other existing steps.



Step 1: `AStar`, path finding
Step 2: `AddDestinationPair`, rush across the key
Step 3: `AvoidWall`, move all points away from wall to avoid collision
Step 4: `RemoveRedundantPath`, keep only 2 points on each line segment
Step 5: `SimplifyPath`, Remove all unnecessary turnings if no obstacle is found

Figure 1. An illustration of how the pipeline transforms a path.

Strategy Pattern and Strategy Manager

To apply different strategies for various situations, strategy pattern is used for a convenient swap between strategies. Three strategy classes implementing the `Strategy` interface are created, including `EscapeStrategy` (navigate the car to the exit), `RetrieveKeyStrategy` (exploring the map and pick up the next key if found) and `RepairStrategy` (navigate to a health trap when the evaluation function is satisfied). As for `MyAIController`, it has no knowledge of which strategy is active. Whenever it needs to recompute targets or need a new set of targets, it can simply call the `getTargets()` method of `StrategyManager`.

`StrategyManager` applies the Pure Fabrication pattern, which is assigned with a highly cohesive set of responsibilities, including the maintenance of the current strategy and switch to another strategy based on the current situation. It uses a stack to store pending strategies, the top one is considered as active and will be pop out from the stack once it finishes its own task. If a strategy needs to take over control, it will be pushed onto the top of the stack. The reason of choosing stack is its “last-in-first-out” (LIFO) property which is very helpful in the strategy switching process here. For example, once the `RepairStrategy` completes its task, the next one to activate is the previous active strategy before the `RepairStrategy` took over control. By using the stack structure, we can simply pop out the `RepairStrategy` once it’s completed to resume the previous strategy. Thus, in general, `EscapeStrategy` and `RetrieveKeyStrategy` will be pushed into the stack during the initialization and `RepairStrategy` will be waiting for the right condition to take over the control.