Ido Weinstein | idoweinstein@mail.tau.ac.il
Amit Gabay | amitgabay@mail.tau.ac.il

VACUUM CLEANER - HLD

Tel Aviv University - Advanced Topics in Programming – Assignment 1

OVERVIEW

1. Scope

This document outlines the high-level functional design of our project. It highlights the main components, along with the rationale for them. The main event sequence is also outlined in this document. as well as our validation approach. The last part of this document details additional information about alternative options for design and implementation.

2. Objective and requirements

The project simulates the operation of an automatic vacuum cleaner. The simulation consists of the robot's internal state, the physical environment in which it operates, its sensors and the algorithm which governs its movement.

Detailed requirements can be found in the official guidelines document for assignment 1.

3. House Model

The physical environment (house) is a 4-connected grid. Each node contains an object: a wall, a docking station, or an empty space. Each empty space has a dirt level assigned to it, ranging from 0 (clean) to 9 (filthy). The house objects do not mutate during the operation of the simulation, which means they cannot change position, appear or disappear during the simulation process. Nevertheless, the dirt level of an empty space may decrease if a robot cleaned it.

Ido Weinstein | idoweinstein@mail.tau.ac.il
Amit Gabay | amitgabay@mail.tau.ac.il

SYSTEM DESIGN

4. System Components

4.1 Component Overview

The project contains the following classes, each representing a main component:

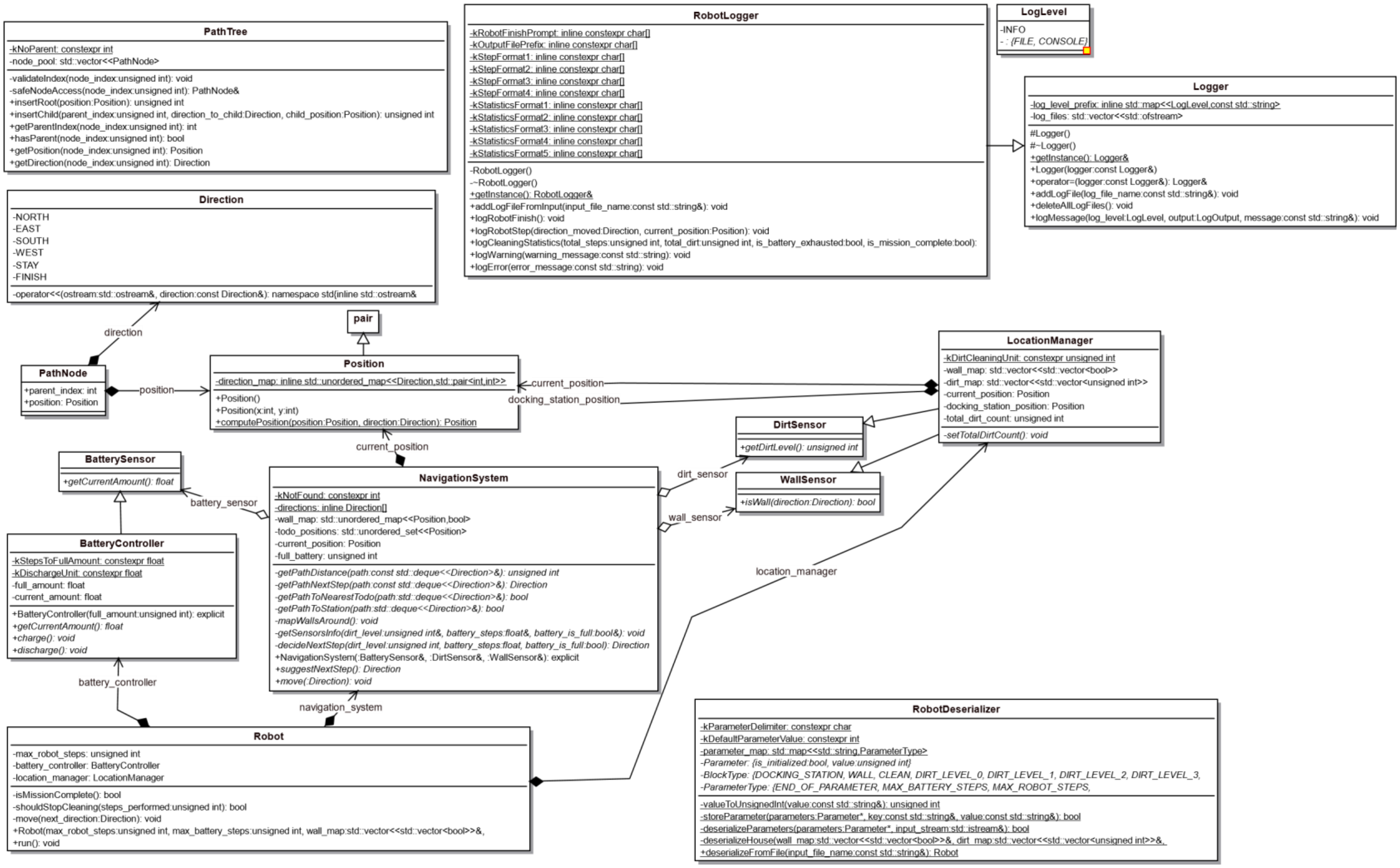
- ***Robot*** – representing the robot simulator. It serves as the main class and initiates the operation of all other components, directly or indirectly.
- ***LocationManager*** – represents the current state of the physical environment – its structure, dirt, and position of the robot. The location manager holds the true state of the environment in any given moment. The location manager implements the ***WallSensor*** and ***DirtSensor*** interfaces which exports the walls around the current location and the dirt level respectively.
- ***BatteryController*** – represents the current state of the battery of the robot. It implements the ***BatterySensor*** interface which exports the current state of the battery.
- ***NavigationSystem*** – represents the algorithm which governs the movement of the robot. The navigation system manages an independent internal state of the environment. The internal state is updated by the information from the sensors which ***LocationManager*** exports, and the decision of the direction of movement is also directed by the state of the battery which ***BatteryController*** exports.

In addition, several auxiliary components are implemented as well:

- ***RobotDeserializer*** – reads the input file and creates a compliant ***Robot*** object.
- ***RobotLogger*** – writes logs to the standard output and to the output file (distinguished by ***LogLevel***). It derives from ***Logger*** class which implements generic logging logic.
- ***Direction*** – an enum representing a direction in which the robot can move (or stay).
- ***Position*** – represents a position in the environment.
- ***PathTree*** – representing a tree of possible paths. ***NavigationSystem*** utilizes ***PathTree*** when the former decides the direction of movement. The path tree consists of (zero or more) *PathNode* instances.

4.2 Component Attributes and Relations

The attribute of each component and the relations between them are further elaborated in the following diagram:


PathTree
-kNoParent: constexpr int
-node_pool: std::vector<<PathNode>
-validateIndex(node_index:unsigned int): void
-safeNodeAccess(node_index:unsigned int): PathNode&
+insertRoot(position:Position): unsigned int
+insertChild(parent_index:unsigned int, direction_to_child:Direction, child_position:Position): unsigned int
+getParentIndex(node_index:unsigned int): int
+hasParent(node_index:unsigned int): bool
+getPosition(node_index:unsigned int): Position
+getDirection(node_index:unsigned int): Direction
RobotLogger
-kRobotFinishPrompt: inline constexpr char[]
-kOutputFilePrefix: inline constexpr char[]
-kStepFormat1: inline constexpr char[]
-kStepFormat2: inline constexpr char[]
-kStepFormat3: inline constexpr char[]
-kStepFormat4: inline constexpr char[]
-kStatisticsFormat1: inline constexpr char[]
-kStatisticsFormat2: inline constexpr char[]
-kStatisticsFormat3: inline constexpr char[]
-kStatisticsFormat4: inline constexpr char[]
-kStatisticsFormat5: inline constexpr char[]
-RobotLogger()
-~RobotLogger()
+getInstance(): RobotLogger&
+addLogFileFromInput(input_file_name:const std::string&): void
+logRobotFinish(): void
+logRobotStep(direction_moved:Direction, current_position:Position): void
+logCleaningStatistics(total_steps:unsigned int, total_dirt:unsigned int, is_battery_exhausted:bool, is_mission_complete:bool):
+logWarning(warning_message:const std::string): void
+logError(error_message:const std::string): void
LogLevel
-INFO
- : {FILE, CONSOLE}
Logger
-log_level_prefix: inline std::map<<LogLevel,const std::string>
-log_files: std::vector<<std::ofstream>
#Logger()
#~Logger()
+getInstance(): Logger&
+Logger(logger:const Logger&)
+operator=(logger:const Logger&): Logger&
+addLogFile(log_file_name:const std::string&): void
+deleteAllLogFiles(): void
+logMessage(log_level:LogLevel, output:LogOutput, message:const std::string&): void
Direction
-NORTH
-EAST
-SOUTH
-WEST
-STAY
-FINISH
-operator<<(ostream:std::ostream&, direction:const Direction&): namespace std{inline std::ostream&
direction
pair
PathNode
+parent_index: int
+position: Position
position
Position
-direction_map: inline std::unordered_map<<Direction,std::pair<int,int>>
+Position()
+Position(x:int, y:int)
+computePosition(position:Position, direction:Direction): Position
current_position
docking_station_position
LocationManager
-kDirtCleaningUnit: constexpr unsigned int
-wall_map: std::vector<<std::vector<bool>>
-dirt_map: std::vector<<std::vector<unsigned int>>
-current_position: Position
-docking_station_position: Position
-total_dirt_count: unsigned int
-setTotalDirtCount(): void
DirtSensor
+getDirtLevel(): unsigned int
current_position
BatterySensor
+getCurrentAmount(): float
battery_sensor
NavigationSystem
-kNotFound: constexpr int
-directions: inline Direction[]
-wall_map: std::unordered_map<<Position,bool>
-todo_positions: std::unordered_set<<Position>
-current_position: Position
-full_battery: unsigned int
-getPathDistance(path:const std::deque<<Direction>&): unsigned int
-getPathNextStep(path:const std::deque<<Direction>&): Direction
-getPathToNearestTodo(path:std::deque<<Direction>&): bool
-getPathToStation(path:std::deque<<Direction>&): bool
-mapWallsAround(): void
-getSensorsInfo(dirt_level:unsigned int&, battery_steps:float&, battery_is_full:bool&): void
-decideNextStep(dirt_level:unsigned int, battery_steps:float, battery_is_full:bool): Direction
+NavigationSystem(:BatterySensor&, :DirtSensor&, :WallSensor&): explicit
+suggestNextStep(): Direction
+move(:Direction): void
dirt_sensor
WallSensor
+isWall(direction:Direction): bool
wall_sensor
BatteryController
-kStepsToFullAmount: constexpr float
-kDischargeUnit: constexpr float
-full_amount: float
-current_amount: float
+BatteryController(full_amount:unsigned int): explicit
+getCurrentAmount(): float
+charge(): void
+discharge(): void
location_manager
battery_controller
navigation_system
Robot
-max_robot_steps: unsigned int
-battery_controller: BatteryController
-location_manager: LocationManager
-isMissionComplete(): bool
-shouldStopCleaning(steps_performed:unsigned int): bool
-move(next_direction:Direction): void
+Robot(max_robot_steps:unsigned int, max_battery_steps:unsigned int, wall_map:std::vector<<std::vector<bool>>&,
+run(): void
RobotDeserializer
-kParameterDelimiter: constexpr char
-kDefaultParameterValue: constexpr int
-parameter_map: std::map<<std::string,ParameterType>
-Parameter: {is_initialized:bool, value:unsigned int}
-BlockType: {DOCKING_STATION, WALL, CLEAN, DIRT_LEVEL_0, DIRT_LEVEL_1, DIRT_LEVEL_2, DIRT_LEVEL_3,
-ParameterType: {END_OF_PARAMETER, MAX_BATTERY_STEPS, MAX_ROBOT_STEPS,
-valueToUnsignedInt(value:const std::string&): unsigned int
-storeParameter(parameters:Parameter*, key:const std::string&, value:const std::string&): bool
-deserializeParameters(parameters:Parameter*, input_stream:std::istream&): bool
-deserializeHouse(wall_map:std::vector<<std::vector<bool>>&, dirt_map:std::vector<<std::vector<unsigned int>>&,
+deserializeFromFile(input_file_name:const std::string&): Robot

5. Program Flow

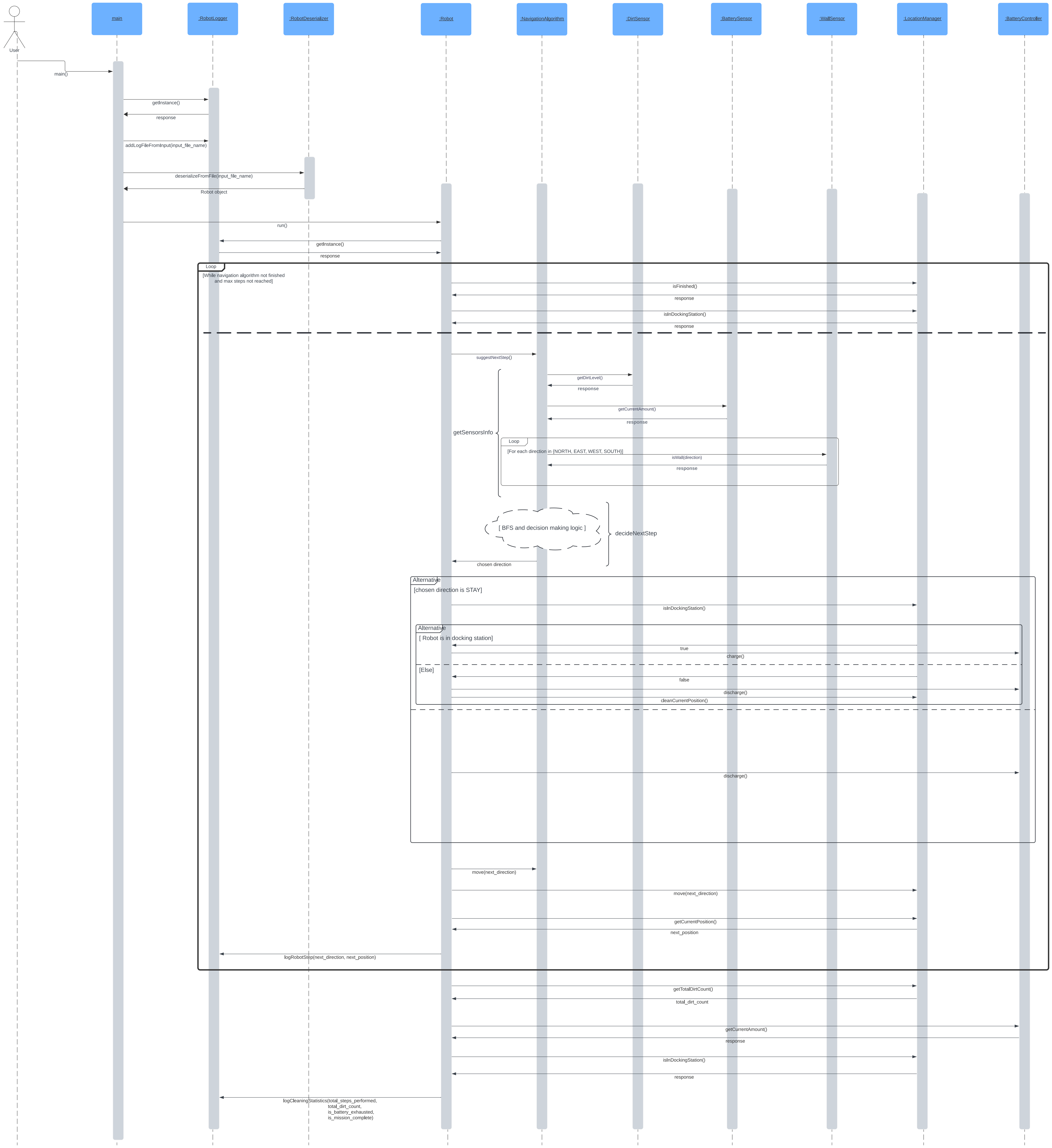
The project contains many possible flows. The sequence diagram on the following page details the main event sequence of the program.

The diagram does **not** describe the following flows:

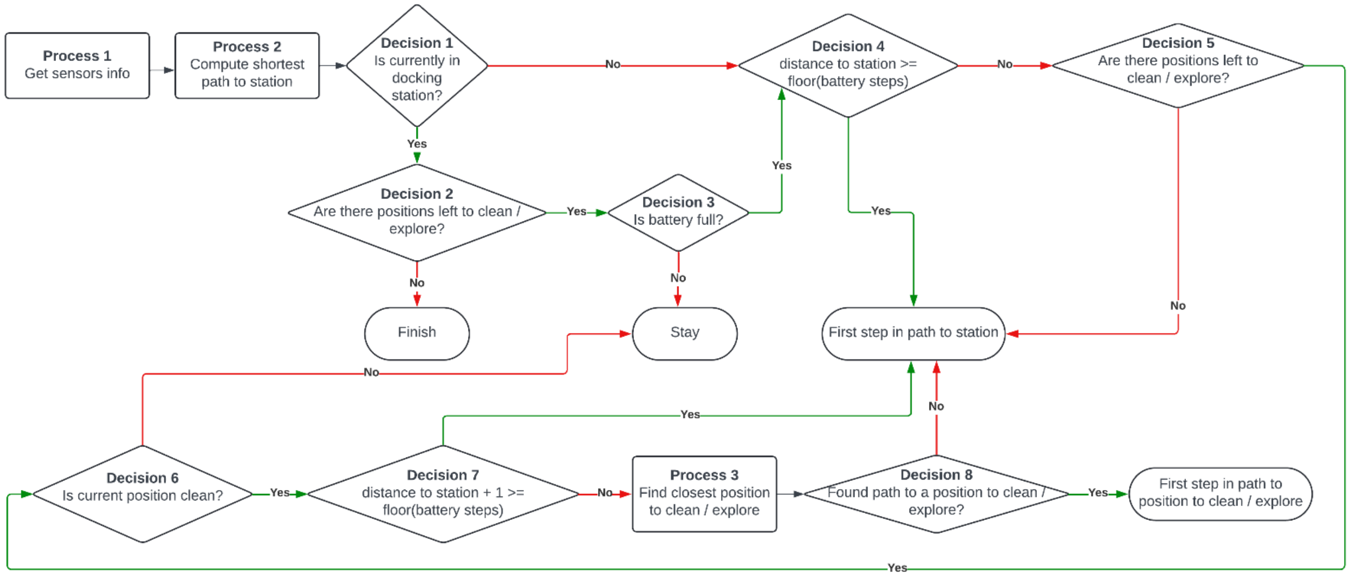
1. **Self-flows of the components** (a method calling another method of the same class). The decision-making process of the algorithm is detailed in the next section.
2. **Utilization of auxiliary components**, as they are of less significance. It also doesn't describe the path-tree building logic, which is loaded with technical details and can briefly explained as performing a BFS.
3. **Errors and error handling**, as it is not the main flow of the program.
4. **Tests and visual simulator**, as they are not part of the program.

It is also worth noting again that the sensors are pure abstract classes and do not implement any logic. ***LocationManager*** implements both ***DirtSensor*** or ***WallSensor***, while ***BatterySensor*** is implemented by ***BatteryController***.

Ido Weinstein | idoweinstein@mail.tau.ac.il
Amit Gabay | amitgabay@mail.tau.ac.il



5.1 Algorithm’s Decision-making Flow


Process 1
Get sensors info
Process 2
Compute shortest path to station
Decision 1
Is currently in docking station?
No
Decision 4
distance to station >= floor(battery steps)
No
Decision 5
Are there positions left to clean / explore?
Yes
Decision 2
Are there positions left to clean / explore?
Yes
Decision 3
Is battery full?
Yes
Yes
No
No
No
Finish
Stay
First step in path to station
No
Yes
No
Decision 6
Is current position clean?
Yes
Decision 7
distance to station + 1 >= floor(battery steps)
No
Process 3
Find closest position to clean / explore
Decision 8
Found path to a position to clean / explore?
Yes
First step in path to position to clean / explore
Yes

The diagram above implies:

1. If not finished, the robot charges until the battery is full whenever it returns to the station.
2. If possible, the navigation system prefers cleaning over exploring. It suggests cleaning the current position before reaching other positions to clean or explore.
3. If making a certain step makes the docking station unreachable due to lack of battery, the algorithm wouldn’t suggest it.
4. If there are no longer positions to clear or explore, the robot returns to the station and then finishes.
5. Erroneous states are also handled:
    a. Distance to station should never exceed the current battery steps. However, **decision 4** still handles this case.
    b. If the navigation system registers there are positions to clean / explore (**decision 5**), then the algorithm should always find a path to a position to clean / explore. However, **decision 8** handles the case in which path is not found.

6. Inputs and Outputs

6.1 Input File

The input file contains information about the maximum amount of allowed steps, the total battery capacity, and the map of the house (the physical environment).

In general, the format of the input file is as follows:

```
max_robot_steps <value>
max_battery_steps <value>
house
<map>
```

6.1.2 Max Robot Steps

Denoted as *max_robot_steps* in the input file. It represents the total amount of steps the robot is allowed to take during the entire simulation process. The robot may take fewer steps than allowed if the algorithm detected that the reachable part of the room is completely empty, and the robot is currently on the docking station.

The default value of this attribute is 0, and this value must be a non-negative integer. If the value is missing or invalid, the simulation uses the default value instead.

6.1.3 Max Battery Steps

Denoted as *max_battery_steps* in the input file. It represents the total capacity of the battery, where the unit of measurement is steps. The robot may take more steps than *max_battery_steps* in the entire simulation, but the battery must be charged because the remaining capacity of the battery can never reach below 0 capacity.

The default value of this attribute is 0, and this value must be a non-negative integer. If the value is missing or invalid, the simulation uses the default value instead.

6.1.4 House

Denoted as *house* in the input file. It represents the objects that make up the physical environment in which the robot operates. In the file, it is formatted as a 2d jagged array where each cell contains a different character representing the object in that position. For example:

```
1102xxxxxxxxxxxxx
xx@x7xx8000503021xxx
3x0x20x0xx7xxx770114
449016150002xxxxxxxx
```

The map uses the following legend:

@	docking station
X	wall (non-navigable position)
[space]	empty (navigable) position with dirt level of 0
0	empty position with dirt level of 0, same as [space]
⋮	
9	empty position with dirt level of 9

6.2 Output File

The output file details each step the robot made during the simulation. It also indicates if the robot reached all accessible positions and shows other statistics about the simulation.

In general, the format is as follows:

```
[Step] Robot took step to <direction> - New Position <y, x>
 ⋮
[Step] Robot took step to <direction> - New Position <y, x>
### Program Terminated ###
Total Steps Taken: <value>
Total Dirt Left: <value>
Is Battery Exhausted: <value>
Mission Succeeded: <value>
```

If all accessible positions were reached and cleaned completely, the following line is printed before the final statistics:

```
[FINISH] Robot finished cleaning all accessible places!
```

VALIDATION

To validate the correctness of the program, a set of automatic tests were created and performed. In addition, we ran a visual simulator of the program and checked visually if each step complies with the house model and the instructions of the assignment.

7. Automatic Tests

The automatic tests are a combination of system and unit tests which aim to validate the correctness of different scenarios the simulation program might encounter. The tests are separated into different files, where each file contains tests for a specific component (module).

7.1 Robot Tests

These tests are in fact complete system tests, which take different inputs and validate the output file the program yields.

The tests scenarios are:

- Sanity – smoke-test scenario of a trivial house
- Trapped dirt – a house containing an inaccessible dirty position.
- Maze – a house which is relatively hard to navigate because of its walls.
- Minimal battery to complete – a battery which has the exact capacity required to clean the house.
- No dirt – a house with no dirt.
- Too distant dirt – a battery which does not have enough battery to reach and clean a dirty position.
- All characters – a house with all available type of objects and dirt levels (represented by different characters in the input file).
- No house – an input file missing a house attribute.
- No docking station - an input file missing docking station.

The scenarios above are validated by the output file the program creates. To test the output file, an infrastructure which reads and deserialized the output file was created.

7.2 RobotDeserializer Tests

These tests check the warnings produced by processing different input files.

The test scenarios are:

- Invalid input file – an input file name which does not exist.
- Input sanity – a trivial input file.
- Invalid parameters – an input file containing an unsupported attribute.
- Missing parameters – an input file missing a required attribute.
- Missing parameter value – an input file with an attribute missing a value.
- Missing house – an input file missing a house attribute.
- Missing docking station – an input file its house missing a docking station.
- Duplicate docking station – an input file with more than one docking station.
- Invalid house character – an input file with a house attribute its value contains an unsupported character.

7.3 NavigationSystem Tests

These tests check the correctness of the navigation algorithm in different scenarios.

The test scenarios are:

- Blocked by walls – the robot is surrounded by walls (all 4 directions).
- Dirty docking station – the robot is standing on a dirty docking station.
  NOTE: This test does not simulate a valid scenario.
- Return to docking station – the robot returns to station upon completion of the mission.
- Too low battery to get further – the remaining battery capacity forces the robot to stay in its current position.
- Too low battery to stay – the remaining battery capacity forces the robot to go to station.

The navigation system interacts with its sensors to decide movement. We mocked the sensors to eliminate any bug that may be not related to the code of the navigation system itself.

7.4 LocationManager Tests

The test scenarios are:

- Total dirt count sanity – check if the location manager sums the total dirt count correctly.
- Starting at docking station – check if the location manager places the robot at the docking station position when it starts.
- Overclean position – clean a position with a dirt level of 0.
- Random navigation – move the robot randomly to navigable positions.
- Move out of bounds – move to position out of the bounds of the house.

When the location manager receives a request to move to a non-navigable position, it throws an error. The relevant tests check this behavior happens in practice.

7.5 BatteryController Tests

The test scenarios are:

- Discharge amount sanity – discharge a battery and compare the actual to the expected capacity.
- Charging time sanity – discharge a battery completely, then charge it and make sure it takes exactly 20 charges to reach total capacity.
- Non integral current amount – discharge and charge the battery, causing its remaining capacity to be non-integral.

7.6 PathTree Tests

A path tree is a rooted tree where each node represents a position, and a parent node is a neighbor of all its children. When a node is inserted into the tree, it receives a unique index.

The test scenarios are:

- Insert root sanity – insert a root to a path tree.

- Bad parent index – insert a child with an invalid parent index.
- Bad node index – try to get properties of nodes with invalid index.

ALTERNATIVES AND DISCUSSION

8. House Model – Zero-width Wall Support

A real house might have a wall with negligible with, or other furniture of insignificant depth which blocks its space. For example, doors, screens (space dividers), and baby gates all match this category. Our current house model forces all objects to have non-zero dimensions, failing to represent the concept of “zero-width”.

We could adjust the model to support zero-width by changing the representation of a non-navigable position. A non-navigable position (wall) is now represented by a different type of node in the grid-graph. Instead, if a robot can’t reach from position A to position B, it could be represented by a lack of an edge between these two places.

However, we chose to not support it due to the following reasons:

1. It is not a real thing – “zero-width” is an abstraction of a concept, and there are no actual house objects with no width at all. All objects with a so called “zero-width”, do have a positive width, albeit negligible. Therefore, by increasing the resolution of the house map and adjusting the battery capacity and steps limits accordingly, one can represent a house with negligible-width objects using the chosen house model.
2. User friendly input – it is very easy to create and modify the input file in its current form. Adding zero width support means adding information about the edges, which can make the format of the house map confusing.
3. Ease of implementation – we were directed to adopt a house model which makes the implementation easier. Supporting zero-width surely makes the implementation more complex.

9. RobotLogger Redesign for Testing

Both ***Logger*** and ***RobotLogger*** (which derives from the former) follow the singleton pattern as they both create a single logger. The logger is then reachable from different points of the program by calling ***RobotLogger::getInstance*** function. This design pattern pose challenges when it comes to testing the entire program, because we cannot mock ***RobotLogger::getInstance***.

Possible solutions to the problem are:

1. Don’t mock the function, let the logger write to the output file and then parse the output file.
2. Call ***RobotLogger::getInstance*** in ***main***. Then pass the logger to any component which uses it. The tests then can create a class which derives from ***RobotLogger*** and exports the values the test would like to check.
3. Make ***RobotLogger*** itself export the values the test would like to check.

The chosen solution is the first option, for the following reasons:

1. It does not require changing the code for the tests, making it prone to less bugs since it is relatively simple in its current form.
2. Writing to the output file is part of the program, so testing it when running system tests enhances the integrity of the program.

10. NavigationSystem Suggestion API

Originally, we designed the algorithm to suggest the direction of movement via ***NavigationSystem::suggestNextStep*** which returned one of the following: North, East, South, West, Stay.

The mentioned API poses two core problems:

1. The algorithm must suggest a direction of movement, even if it has no suggestion. The scenario of no suggestion might occur in case the algorithm mapped the entire accessible map and cleaned it.

2. The algorithm must suggest one step, even if it already knows an entire path to a position. This scenario happens very frequently, as our algorithm is based on finding the shortest paths to specific positions and suggesting the first step to the chosen target. The paths often consist of two or more steps. For example, returning to the station for charging requires usually more than one step. Returning to the closest dirty spot after charging also tends to require multiple steps.

Solving the second problem felt a bit out of scope of this assignment, as it forces the algorithm to acknowledge it could encounter new information during the movement along the path, which could change its suggestion. For example:

1. It could be notified by the wall sensor of a new non-wall position which could shorten the shortest path to the desired destination.
2. It could be notified by the dirt sensor of a dirt along the path it has just enough battery capacity to clean.

Solving all these challenges is surely possible, but requires further engineering and redesigning which, as mentioned above, is out of the scope of this assignment.

The first problem was eventually solved by adding a new suggestion type – Finish. When the algorithm suggests it, the robot knows the algorithm has no suggestions, stops the operation of the robot and initiates the termination logic of the simulation.