Vacuum Cleaner Documentation

1. Introduction

The objective of the vacuum cleaner project is to simulate the behavior of a vacuum cleaner running and cleaning a house using the given algorithm. The task included reading home design information from a file, simulating the operation of a vacuum cleaner, and creating an output file with the results of the cleaning process

2. Design considerations

2.1 Resource Considerations

The project follows an object-oriented design model, and models the problem with division to 3 distinct classes:

- House: represents the layout of the house with walls and corridors, and maintains the cleaning state of the house (how much dirt per location).
- VaccumCleaner: Represents the vacuum cleaner and its functionality- clean, charge or move in a direction. In addition, maintaining the state of the vacuum cleaner along the program: current location, battery level etc.
- CleaningAlgorithm: Contains logic for determining the moves of the vacuum cleaner, conveying its decisions with getNextMove method. The class preserves moving and charging states between calls, so it can maintain logic based on the previous steps and current state.
- Index: Includes utility functions for reading input data and generating output.

2.2 Encapsulation and Modularity

Each class incorporates specific functionality, encouraging undependability and encapsulation. That way, it was possible to encapsulate the house map and state, vacuum cleaner position and more from the algorithm, as the demand was that the algorithm would be completely separated from the rest of the program. With our class design, we controlled directly what the algorithm sees along its run - only the 3 allowed sensors which were sent to him as arguments in each call to getNextMove.

Another merit of the class design is the modularity aspect. With the separation to 3 distinct objects representing different functionality, we will be able to easily modify our code to align with the next assignments. For example, by creating a whole class for the algorithm which can maintain state, it would be easier for us to implement a mapping of the house with BFS/DFS in the future.

2.3 Error Handling

The code includes extensive error handling to ensure robustness. For example, the updateDataFromFile function checks for various error conditions related to file reading and data validation, and every request for data in a certain index is checked before accessing the index. Although the class separation made more errors

possible because of the interactions between objects, we were aware of that and carefully checked for many kinds of errors and handled them accordingly.

Mostly we preferred to not abruptly stop the program, and tried to recover from errors when possible. For example, if house.getDirtLevel(location) is called with a location outside of the house coordinates, it returns 0 (no dirt) rather than stopping with an error.

3. Design Alternatives

3.1 Simpler class structure

The current system centralizes the control logic in the CleaningAlgorithm class . An alternative would have been to maintain the algorithm within the vacuum cleaner class and make it more simple with fewer objects. We eventually decided to create a separate class for the algorithm because it made it easier to encapsulate data from it, and felt more intuitive with regard to the assignment description which stated that the algorithm should be completely unaware of anything else in the program. In addition, as said above, it would be easier to implement a better algorithm in the next assignment with an algorithm class.

3.2 Stack-based route tracking

The path to the docking station is tracked using the stack of the CleaningAlgorithm class and it's not an optimal solution. Another option would have been to find a shortest path for example. A stack-based approach was chosen due to its simple iterative nature, avoiding potential problems with more complex solutions, as the demand was for a naive algorithm. In the next assignments we would probably map the house, and then we will adjust our solution to meet up with the exact requirements and be more efficient.

3.3 Mapping algorithm

As for now, the algorithm is very simple, and the decision making in getNextMove is of the following order (in general):

- If the max steps limitation is nearing - go back to the docking station and finish.
- Else, if battery steps are exactly as the length of the way back to docking - go to the docking station and fully charge.
- Else, if there is dirt on the current location (from dirt sensor) - stay and clean it completely.
- Else - make a move in the house. At first, a random direction with no wall ahead is chosen, and then in the next steps we will continue advancing in this direction until reaching a wall. Then, we will randomly decide on another safe direction and continue. Each step is documented in a stack so we would know the way back to docking when needed.

We could have made a better algorithm which maps the house structure and maintains a dirt state, however this algorithm suffices to answer the current requirements of the assignments (it is aware of battery level, won't go into walls and stay more than needed in any location etc.). Thus we preferred this simple algorithm, as we don't yet know the specific requirements for the algorithm in the next assignments.

4. Testing

Our approach to testing was to check that edge cases in the input file are managed without crashing, wrong file formats are also handled accordingly. We also ran many manual tests to check that the result is correct, by comparing the logging of the vacuum cleaner actions with the expected behavior of the algorithm, over various house structures and initializations.

5. Input file format

For your convenience, here is the input file format the program expects:

a b c d
- row 1 -
…
- row n -

Where <a,b> are positive indexes stating the docking station coordinates (starting from zero); c is a positive integer of max battery steps; d is a positive integer of max steps.
The rows represent the house map, where there could be any value from -1 to 9. -1 represents a wall, 0-9 indicates dirt level.

6. Conclusion

The design of the vacuum cleaner project emphasizes modularity, encapsulation, robustness, and ease of understanding. Following an object-oriented approach and separating the problem into different classes and states, made it easier to handle the assignment requirements, and have a decent and convenient groundwork for next assignments.

Classes Diagram

VaccumCleaner

- float batterySteps;
- const int maxBatterySteps;
- tuple<int,int> curLoc;

- **VaccumCleaner**(const int battery,const tuple<int,int> loc)
- const vector<string>& **cleanHouse**(House& h,CleaningAlgorithm& alg, const int maxSteps); //cleans the house according to the algorithm.
- int **stay**(House& h);//stays in current loc and cleans dirt/charging.
- int **move**(const int dir);//moving 1 step towards the direction dir.
- float **getBatterySteps**() const; //returns how much battery left currently
- tuple<int,int> **getCurrentLoc**() const; //return current location

House

- const tuple<int,int> houseSize;
- const tuple<int,int> dockingStationLoc;
- int totalDirtLeft;
- vector<vector<int>> houseMap;

- private bool **isLocInsideHouse**(const tuple<int,int>& loc) const;
- **House**(const tuple<int,int> dockingLoc,const vector<vector<int>> map)
- void **updateCleaningState**(const tuple<int,int> loc);//updates cleaning step on location loc, and decreases dirt level by 1.
- int **getDirtLevel**(const tuple<int,int> loc) const;//returns dirt level on loc.
- bool **isWallInDirection**(int direction, const tuple<int,int> curLoc) const; int **getTotalDirtLeft**()const;//returns the sum of dirt left in the house.
- tuple<int,int> **getDockingStationLoc**()const;
- string **toString**() const;

CleaningAlgorithm

- const size_t maxBattery;
- const size_t maxSteps;
- size_t curStep;
- int curDirection;
- bool isCharging;
- stack<int> pathToDocking; //stack to remember the way

- private void **chooseNextDirection**(const vector<int> wallSensor);//randomly choose direction to go next, with no wall ahead.
- **CleaningAlgorithm**(const int battery,const int steps);
- int **getNextMove**(const int dirtSensor,const int batterySensor, const vector<int>& wallSensor) //decide on next step according to the sensors.

Sequence Diagram

Here is a high level diagram of the program flow. We omitted some minor transactions to keep it simple and better convey the main flow.