INTI International College Penang **School of Engineering and Technology**

3+0 Bachelor of Science (Hons) in Computer Science, in collaboration with Coventry University, UK

3+0 Bachelor of Science (Hons) in Computing, in collaboration with Coventry University, UK

Coursework cover sheet

Section A - To be completed by the student

Full Name: **Sim Jin Yi**	
CU Student ID Number: **9658521**	
Semester: **August 2020 (Semester 8)**	
Lecturer: VASUKY MOHANAN	
Module Code and Title: 310CT INTELLIGENT AGENTS	
Assignment No. / Title: ASSIGNMENT 2	30% of Module Mark
Hand out date: 28/9/20	Due date: 23/11/20
Penalties: No late work will be accepted. If you are unable to submit coursework on time due to extenuating circumstances you may be eligible for an extension. Please consult the lecturer.	
Declaration: I/we the undersigned confirm that I/we have read and agree to abide by the University regulations on plagiarism and cheating and Faculty coursework policies and procedures. I/we confirm that this piece of work is my/our own. I/we consent to appropriate storage of our work for plagiarism checking. Signature(s): ----------	

Section B - To be completed by the module leader

Intended learning outcomes assessed by this work: 1. Critically analyse and apply a range of AI reasoning processes. 2. Design and implement an AI application using a collaborative agent based approach.		
Marking scheme	Max	Mark
1. Individual Assignment Report	100	
Total	100	
Lecturer's Feedback		
Internal Moderator's Feedback		

Table of Contents

1.0 Introduction

The application of technologies in households have been drastically evolving in the past decades. This was particularly notable with the prevalence of smart electrical appliances like vacuum robots, personal assistant speakers, IoT electrical switches, etc. Focusing on vacuum robots, various sensors such as bumper sensors, infrared sensors, etc. were installed to help the vacuum robots to navigate across and clean up a room effectively *(Baguley & McDonald, 2015)*. Also, the robots can also collaborate with each other in a multi-agent environment to improve the effectiveness of the robots by maximizing the coverage and shorten the time to clean up.

To study the application of the BDI architecture in building the communicative vacuum robots, JAM, which was based on the BDI architecture will be used to develop a program that contains two communicative vacuum robots in a grid. Then, the dusts scattered over the grid will be required to be cleaned by both the robots collaboratively. After cleaning all the dusts, both robots will be directed to an exit on the grid, after which all the operations were completed.

In a nutshell, the main objective was to build a program with the BDI architecture to showcase the operation of two vacuum robots collaborating to clean up a room. Apart from the objective, additional features will be added to the robots to improve the effectiveness in the collaboration between the robots which includes preventing to revisit a cell that has been previously visited. Also, the capability to handle the existence of obstacles in the middle of the grid which emulates the real-world scenarios were added to the robots for a more thorough view on the capabilities of collaborative robots using the BDI architecture in the real-world.

2.0 Program Design

2.1 Introduction

In this section, the design of the JAM program will be walked through for each and every plan involved. The flowchart will be included if necessary, followed by the JAM source code. Then, an explanation on the implementation will be included to justify the heuristics involved in the design of the algorithm especially on the navigation of the robots through the grid cells. On a side note, the source code included will contain in-detailed comments in the major sections to further aid in understanding the solution employed towards an issue.

In a nutshell, the program consists of two top-level goals (desires), which were to *clean* and *return*. The goal to *clean* must be attempted first, whereby the robots should attempt to clean so long the dusts are present. Otherwise, the goal to *return* will be initiated, denoting that all the dusts that can be cleaned were cleaned and the robots should exit from the grid.

As discussed, the original question specified a grid without obstacle. Nevertheless, the implementation to cater to the presence of obstacles in the grid was added by the collaborative robots. With such implementation, heuristics with the element of randomness were added to the navigation algorithm to ensure that the robots effectively search through all the grids.

As such, a time limit was also imposed to ensure that the program does not go into an infinite loop when the obstacles totally obstruct at least one dust, which also applies for the exit procedure as the obstacles can be placed around the exit, blocking and hindering the robots to exit. This means that the robots should eventually give up on searching and exiting after the time limit, which was also similar to the real-world scenario when the battery runs out.

To relate the implementation of the robots to the BDI architecture, the beliefs, which were the knowledge of the agents toward the world are the facts in the implementation. Then, the desires are the top-level goals, which were either to *clean* or *return* with priority for the robots to clean over exit. Lastly, the intentions of the agent were reflected in the plans used to achieve the commitments, which were either to clean the dusts in the grid or return to the exit. Please find the complete source code under the GitHub repository at the following URL or in the appendix at the end of the report: https://github.com/simjinyi/310CT_Assignment_2.

2.2 Goals and Facts (Beliefs)

2.2.1 Source Code

```
/* BEGIN: Top-level Goals */
GOALS:
    ACHIEVE clean :UTILITY 10;
    ACHIEVE return :UTILITY 1;
/* END: Top-level Goals */


/* BEGIN: World Model */
FACTS:

    // Robot A (starting (0, 0))
    FACT robotA 0 0;
    FACT robotADirection "right";
    FACT robotASteps 0;

    // Robot B (starting (7, 0))
    FACT robotB 7 0;
    FACT robotBDirection "left";
    FACT robotBSteps 0;

    // List of dusts
    FACT dust 3 0;
    FACT dust 6 0;
    FACT dust 2 1;
    FACT dust 1 2;
    FACT dust 4 2;
    FACT dust 0 3;
    FACT dust 3 3;
    FACT dust 4 4;
    FACT dust 1 5;
    FACT dust 5 5;
    FACT dust 3 6;
```

```
    // List of obstacles
    // The dusts that are surrounded by obstacles cannot be cleaned
    // The robot will attempt to search through the grid for an entry
    // But if none, the robot will give up after the time limit (MAX_X * MAX_Y * 10)
    // FACT obstacle 4 1;


    // Exit location
    FACT exit 7 6;

    // Grid boundaries
    FACT MAX_X 7;
    FACT MAX_Y 6;

    // Whether if the all the grids were visited and the dusts were cleaned
    FACT dustCleaned "false";
/* END: World Model */
```

Excerpt 2.1: Goals and facts in the program

2.2.2 Explanation

The excerpt above shows the code in declaring the goals and facts in the program. The two top-level goals (desires) are to *clean* and *return*, which were assigned with the utility of ten and one respectively, denoting that the robots should prioritize the plan to clean over exiting with one plan provided for each goal. After that, the facts form the world model. Firstly, the information of both robots such as the location, direction, and steps while cleaning were stored.

Subsequently, the information regarding the location of the dusts on the grid were stored. Should the list of dusts be empty, the robots will attempt to exit from the grid immediately when the program starts. Besides that, the facts on the location of the obstacles in the grid follows the list of dusts. Here, the location of the obstacles were commented out from the code as the obstacles were not provided in the question and is optional in the code. However, the implementation to avoid the obstacles were done and will be discussed in the code.

Lastly, the facts on the point of interests such as the location of the exit, boundaries of the grid, and whether if all the dusts in the grid were cleaned were also included. The boundaries refer to the minimum and maximum points in the grid, whereby the minimum for the x-axis and y-axis are both zero, while the maximums of the grid is specified in the facts.

2.3 Plan: Achieve Clean

2.3.1 Flowchart

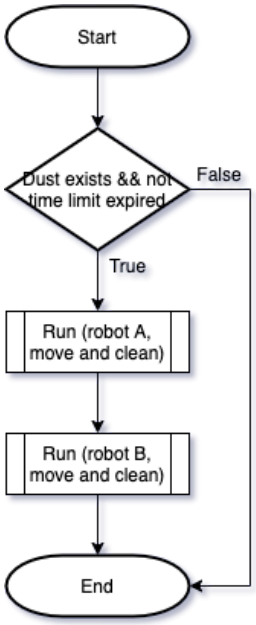

Figure 2.1: Flowchart for the plan to clean

2.3.2 Source Code

```
/* BEGIN: PLAN clean */
PLAN:
{
NAME:
    "Walkthrough the grids and clean the dusts"

DOCUMENTATION:
    "The plan achieves the top-level goal to clean the dusts,
        - The robots will attempt to clean until either,
            - All the dusts were cleaned.
            - The time limit had expired.
        - Both robot will make a move in each iteration.
        - The checking on the status of the grid as follow will be performed every iteration:
            - Whether if all the dusts were cleaned.
            - Whether if the time limit had expired."

GOAL:
    ACHIEVE clean;

BODY:

    EXECUTE println "System: Started";

    // Perform the initial check on the status
    PERFORM check;

    // Obtain the grid boundaries and the time limit to clean
    RETRIEVE dustCleaned $dustCleaned;
    RETRIEVE MAX_X $MAX_X;
    RETRIEVE MAX_Y $MAX_Y;

    ASSIGN $MAX_TRIES (* $MAX_X $MAX_Y 10);
    ASSIGN $iteration 0;

    // Loop to keep the robot roaming around the grid until,
    //  - All the dusts were cleaned.
    //  - The time limit had expired.
    WHILE: TEST(&& (== $dustCleaned "false") (< $iteration $MAX_TRIES)) {

        // Advances one step for both robot
        PERFORM run "robotA";
        PERFORM run "robotB";

        // Check and retrieve the status of the grid
        PERFORM check;
        RETRIEVE dustCleaned $dustCleaned;
        ASSIGN $iteration (+ $iteration 1);
    };
```

```
    EXECUTE print "System: Exiting ";
    WHEN: TEST(== $dustCleaned "true") {
        EXECUTE print "(All Dusts Collected)";
    };

    EXECUTE println "";

FAILURE:
    EXECUTE println "System: Failed to Clean";
}
/* END: PLAN clean */
```

Excerpt 2.2: Plan to achieve the top-level goal to clean

2.3.3 Explanation

The flowchart provide a high-level view over the operations in the plan to clean the dusts in the grid. Firstly, the loop checks if all the dusts were cleaned and the execution time was within the limit before proceeding. The time limit was added to the program to ensure that the code does not go into an infinite loop under certain circumstances. The issue is particularly prominent when the placement of the obstacles completely blocks at least one of the dusts, hindering the dust collection process which ultimately results in an infinite search loop by the robots. Then, the plan to check for the existence of the dusts in the grid will be explained later. The time limit allocated will be ten times the number of cells in the grid as heuristically, the robots should have completed scanning over all the cells in the grid if they were not blocked.

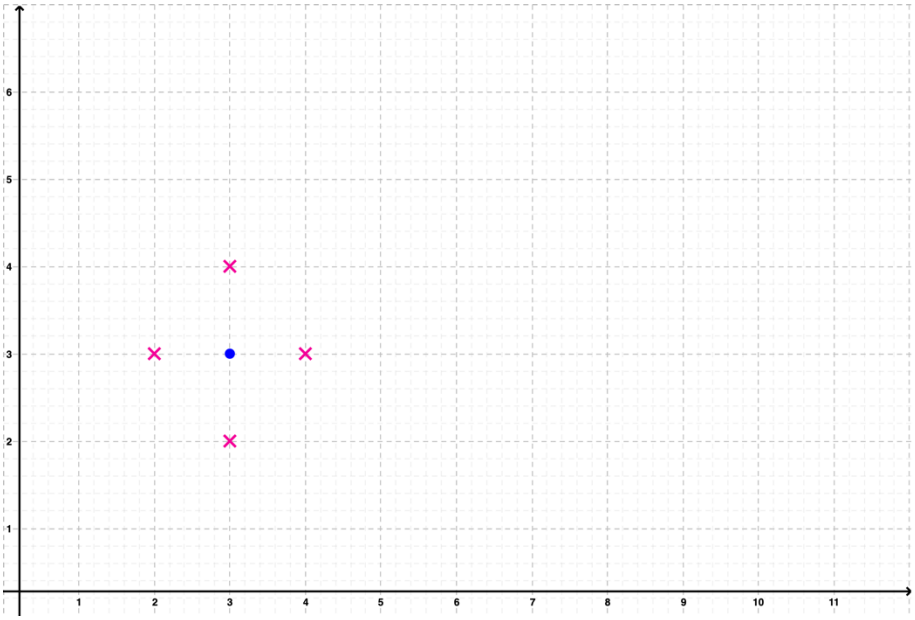

Figure 2.2: Sample scenario when a dust was blocked entirely

As shown in Figure 2.2, a dust (marked in blue) was surrounded by four obstacles (marked in red), which hinders the robots to clean up the dust at position (3, 3). Therefore, the robots will continue to search for the leftover dust until the expiry of the time limit, which was achieved in the code with the *$iteration* variable to keep track of the steps taken by the robots. This also denotes that the battery level of the robots in real-world, whereby the robots give up on searching for the dusts after the battery level is low, and attempt to achieve the second goal which was to exit from the grid which will be explained in the plan in achieving the *return* goal.

2.4 Plan: Achieve Run

2.4.1 Flowchart

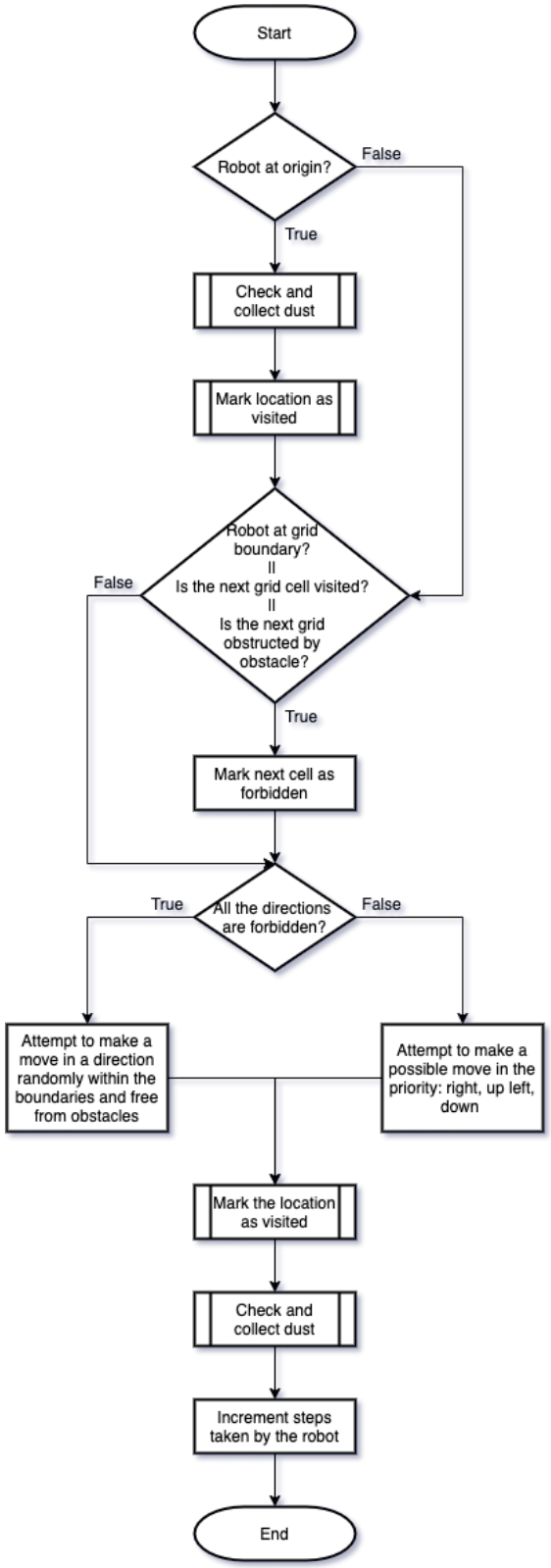

Figure 2.3: Flowchart to decide a direction to move and clean the dusts for the plan

2.4.2 Source Code

```
/* BEGIN: PLAN run $robot */
PLAN:
{
NAME:
    "Decide and advances the selected robot for one step"

DOCUMENTATION:
    "The plan achieves the sub-goal to advance the selected robot for one step,
        - The plan selects the best cell for the selected robot to advance next based on
heuristics.
            - The best cell is selected based on,
                - Whether if the cell was visited.
                - Whether if the next cell is still within the boundaries of the grid.
                - Whether if the next cell is an obstacle.
        - Once there is no better cell, the plan advances the robot randomly within the grid.
        - This hopes that the robot will eventually find an uncollected dust.
        - In the process, the robot will:
            - Update the fact on the visited cells.
            - Attempt to check and collect the dusts."

GOAL:
    ACHIEVE run $robot;

BODY:
    RETRIEVE MAX_X $MAX_X;
    RETRIEVE MAX_Y $MAX_Y;

    // Robot A selected
    WHEN: TEST(== $robot "robotA") {
        RETRIEVE robotA $x $y;
        RETRIEVE robotASteps $iteration;

        WHEN: TEST(<= $iteration 0) {
            EXECUTE println "Robot A: Started at (" $x ", " $y ")";
            PERFORM collect "robotA";
            PERFORM store "robotA";
        };


        /* BEGIN: Robot Movement Logic */
        // Variables to store the valid moves
        ASSIGN $up "true";
        ASSIGN $down "true";
        ASSIGN $left "true";
        ASSIGN $right "true";
        ASSIGN $moved "false";

        // Check which direction is valid and can be moved to base on,
        // - Whether if the robot is at the boundary
```

```
        // - Whether if the cell to be advanced to was visited.
        // - Whether if the cell to be advanced to was blocked by an obstacle.
        WHEN: TEST(|| (<= $x 0) (FACT visited (- $x 1) $y) (FACT obstacle (- $x 1) $y)) {
            ASSIGN $left "false";
        };

        WHEN: TEST(|| (>= $x $MAX_X) (FACT visited (+ $x 1) $y) (FACT obstacle (+ $x 1) $y))
{
            ASSIGN $right "false";
        };

        WHEN: TEST(|| (<= $y 0) (FACT visited $x (- $y 1)) (FACT obstacle $x (- $y 1))) {
            ASSIGN $down "false";
        };

        WHEN: TEST(|| (>= $y $MAX_Y) (FACT visited $x (+ $y 1)) (FACT obstacle $x (+ $y 1)))
{
            ASSIGN $up "false";
        };

        // Check if there is no better move
        WHEN: TEST(&& (== $up "false") (== $down "false") (== $left "false") (== $right
"false")) {

            EXECUTE println "Robot A: No Better Move, Attempting a Random Move";

            // If the robot is at the boundary, move in the opposite direction
            WHEN: TEST(<= $x 0) {
                UPDATE (robotADirection) (robotADirection "right");
                ASSIGN $moved "true";
            };

            WHEN: TEST(<= $y 0) {
                UPDATE (robotADirection) (robotADirection "up");
                ASSIGN $moved "true";
            };

            WHEN: TEST(>= $x $MAX_X) {
                UPDATE (robotADirection) (robotADirection "left");
                ASSIGN $moved "true";
            };

            WHEN: TEST(>= $y $MAX_Y) {
                UPDATE (robotADirection) (robotADirection "down");
                ASSIGN $moved "true";
            };

            // If the robot is not at the boundary, attempt to make a move randomly
            WHEN: TEST(== $moved "false") {
                DO_ANY {
                    UPDATE (robotADirection) (robotADirection "right");
```

```
                ASSIGN $moved "true";
            } {
                UPDATE (robotADirection) (robotADirection "up");
                ASSIGN $moved "true";
            } {
                UPDATE (robotADirection) (robotADirection "left");
                ASSIGN $moved "true";
            } {
                UPDATE (robotADirection) (robotADirection "down");
                ASSIGN $moved "true";
            };
        };
    };

    // Attempt to move rightward whenever possible
    WHEN: TEST(&& (== $right "true") (== $moved "false")) {
        UPDATE (robotADirection) (robotADirection "right");
        ASSIGN $moved "true";
    };

    // Otherwise, attempt to move upward
    WHEN: TEST(&& (== $up "true") (== $moved "false")) {
        UPDATE (robotADirection) (robotADirection "up");
        ASSIGN $moved "true";
    };

    // Otherwise, attempt to move leftward
    WHEN: TEST(&& (== $left "true") (== $moved "false")) {
        UPDATE (robotADirection) (robotADirection "left");
        ASSIGN $moved "true";
    };

    // If all else fails, attempt to move downward
    WHEN: TEST(&& (== $down "true") (== $moved "false")) {
        UPDATE (robotADirection) (robotADirection "down");
        ASSIGN $moved "true";
    };
    /* END: Robot Movement Logic */


    // Move the robot in the decided direction for one step
    // Store the location as visited in the world model (fact)
    // Check and collect the dust if available at the location
    RETRIEVE robotADirection $direction;
    PERFORM move "robotA" $direction;
    PERFORM store "robotA";
    PERFORM collect "robotA";

    UPDATE (robotASteps) (robotASteps (+ $iteration 1));
};
```

```
    // Please refer to the comments in the section above
    // The code is identical but applicable for Robot B
    WHEN: TEST(== $robot "robotB") {
        RETRIEVE robotB $x $y;
        RETRIEVE robotBSteps $iteration;

        // First iteration
        WHEN: TEST(<= $iteration 0) {
            EXECUTE println "Robot B: Started at (" $x ", " $y ")";
            PERFORM collect "robotB";
            PERFORM store "robotB";
        };

        ASSIGN $up "true";
        ASSIGN $down "true";
        ASSIGN $left "true";
        ASSIGN $right "true";
        ASSIGN $moved "false";

        WHEN: TEST(|| (<= $x 0) (FACT visited (- $x 1) $y) (FACT obstacle (- $x 1) $y)) {
            ASSIGN $left "false";
        };

        WHEN: TEST(|| (>= $x $MAX_X) (FACT visited (+ $x 1) $y) (FACT obstacle (+ $x 1) $y))
{
            ASSIGN $right "false";
        };

        WHEN: TEST(|| (<= $y 0) (FACT visited $x (- $y 1)) (FACT obstacle $x (- $y 1))) {
            ASSIGN $down "false";
        };

        WHEN: TEST(|| (>= $y $MAX_Y) (FACT visited $x (+ $y 1)) (FACT obstacle $x (+ $y 1)))
{
            ASSIGN $up "false";
        };

        WHEN: TEST(&& (== $up "false") (== $down "false") (== $left "false") (== $right
"false")) {

            EXECUTE println "Robot B: No Better Move, Attempting a Random Move";

            WHEN: TEST(<= $x 0) {
                UPDATE (robotBDirection) (robotBDirection "right");
                ASSIGN $moved "true";
            };

            WHEN: TEST(<= $y 0) {
                UPDATE (robotBDirection) (robotBDirection "up");
                ASSIGN $moved "true";
            };
```

```
    WHEN: TEST(>= $x $MAX_X) {
        UPDATE (robotBDirection) (robotBDirection "left");
        ASSIGN $moved "true";
    };

    WHEN: TEST(>= $y $MAX_Y) {
        UPDATE (robotBDirection) (robotBDirection "down");
        ASSIGN $moved "true";
    };

    WHEN: TEST(== $moved "false") {
        DO_ANY {
            UPDATE (robotBDirection) (robotBDirection "right");
            ASSIGN $moved "true";
        } {
            UPDATE (robotBDirection) (robotBDirection "up");
            ASSIGN $moved "true";
        } {
            UPDATE (robotBDirection) (robotBDirection "left");
            ASSIGN $moved "true";
        } {
            UPDATE (robotBDirection) (robotBDirection "down");
            ASSIGN $moved "true";
        };
    };
};

WHEN: TEST(&& (== $right "true") (== $moved "false")) {
    UPDATE (robotBDirection) (robotBDirection "right");
    ASSIGN $moved "true";
};

WHEN: TEST(&& (== $up "true") (== $moved "false")) {
    UPDATE (robotBDirection) (robotBDirection "up");
    ASSIGN $moved "true";
};

WHEN: TEST(&& (== $left "true") (== $moved "false")) {
    UPDATE (robotBDirection) (robotBDirection "left");
    ASSIGN $moved "true";
};

WHEN: TEST(&& (== $down "true") (== $moved "false")) {
    UPDATE (robotBDirection) (robotBDirection "down");
    ASSIGN $moved "true";
};

RETRIEVE robotBDirection $direction;
PERFORM move "robotB" $direction;
PERFORM store "robotB";
```

```
        PERFORM collect "robotB";

        UPDATE (robotBSteps) (robotBSteps (+ $iteration 1));
    };

FAILURE:
    EXECUTE println "System: Failed to Run - " $robot;
}
/* END: PLAN run $robot */
```

Excerpt 2.3: Plan to achieve the sub-goal to make a move and clean the dusts

2.4.3 Explanation

The flowchart uncovers the algorithm used to implement the heuristics in navigating the robot. This plan is the key enabler of the program, whereby the plan decides the best move to make, then advances the robot by one grid towards the direction decided. Subsequently, the presence of the dusts will be checked, and the robots will collect the dusts found at the location. Then, the plan updates the world model to mark that the location was visited by the robots. On a side note, the code above will move either robot A or B based on the argument passed, whereby passing *"robotA"* will move robot A while passing *"robot B"* will move robot B.

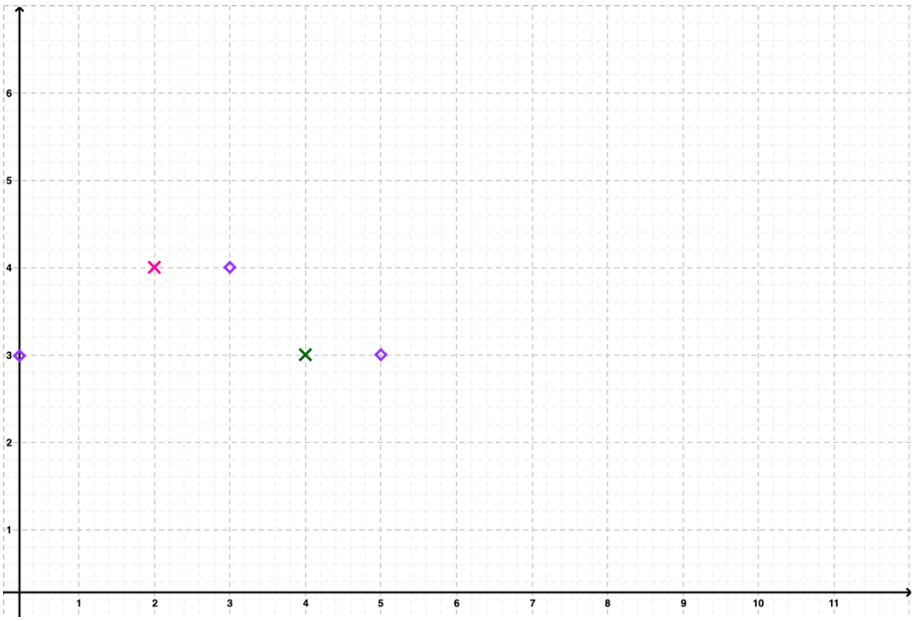

Figure 2.4: Forbidden movement towards the left

Referring to the source code, three conditions were checked to determine if the moving towards a direction is plausible. The three conditions are the presence of obstacles in the adjacent cell, whether if the adjacent cell was previously visited and whether if the move brings the robot out of the boundaries. Any of the matching condition will prevent the robot from moving towards the direction. For example, as shown in Figure 2.4, the purple diamonds represent the robots, while the red and green crosses represent the obstacle and the location marked as visited respectively. The robots cannot move to the left as moving left exceeds the boundaries on the x-axis, blocked by an obstacle and the cell has been visited previously. This part was important to aid the robots in selecting the best adjacent cell to navigate to, which prioritizes the cells which are unvisited first so that most of the cells will be scanned once.

Then, if there is no plausible movement to be made, a random movement will be made by the robot, ignoring whether if the cells were visited in hope of finding the uncollected dusts. Lastly, as the robot moves to the adjacent cell, the location will be marked as visited if the location was not previously marked in the world model (facts), and the presence of dusts will be checked and collected, and the procedures of the plan to move and clean completes.

As for the heuristics, with the introduction of obstacles on the grid, there exists various possibilities which can lead to undesirable outcomes such as unreachable dusts and exit. Therefore, the algorithm to navigate the robots around tries to scan through all the grids at least once before introducing elements of randomness in the movement of the robots. This was achieved by adding a check to prevent the robots from revisiting a visited cell during the first run, which will effectively ensure that the robots progress forward to the unvisited cells only.

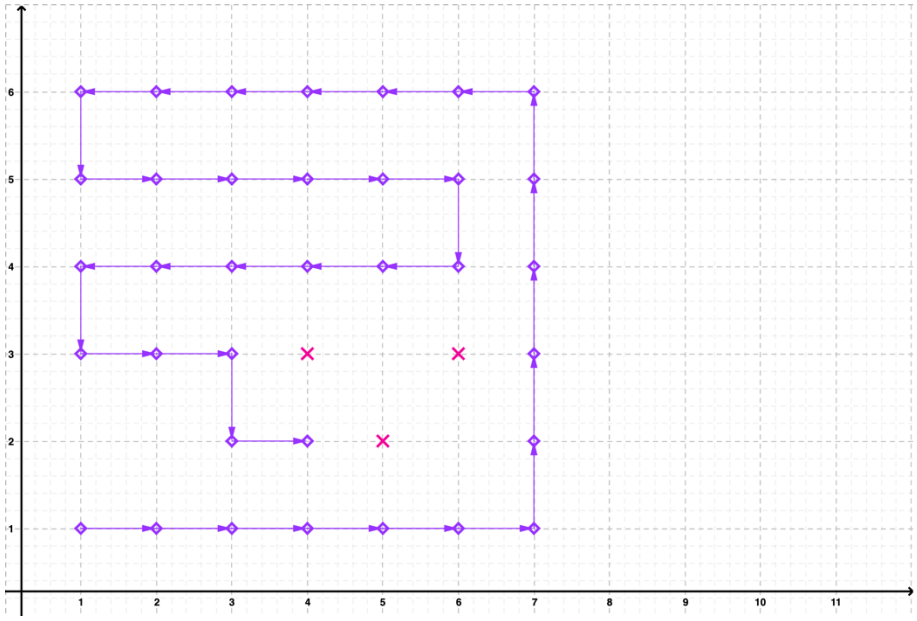

*Figure 2.5: The path taken by **one robot** (deterministically) on first run*

However, should some of the cells be obstructed by the obstacles, some cells may be missed on the first run. To showcase the operation, the initial path for one robot was plotted in Figure 2.5. With the deterministic movement and the placement of the obstacles, the cells (5, 3), (6, 2), (1, 2) and (2, 2) will not be visited on the first run. With the robot reaching (4, 2) where there is no better adjacent cell to navigate to, the robot will start to attempt to move randomly, either to the left or bottom since the top and right sides were blocked by obstacles.

Thus, with the robots being able to move randomly when no better movement can be made, the unvisited cells which are usually placed together will be visited with a very high chance. This is because the unvisited cells which are placed together will be prioritized by the robots once the first unvisited cell is encountered when the robots move randomly as a best move is unidentified. In short, the heuristics involved should help in effectively navigating the robots to visit all the cells in the grid at least once, hereafter collecting all the dusts that are not entirely blocked by the obstacles. Otherwise, the robots retry until the time limit expires.

2.5 Plan: Achieve Move

2.5.1 Flowchart

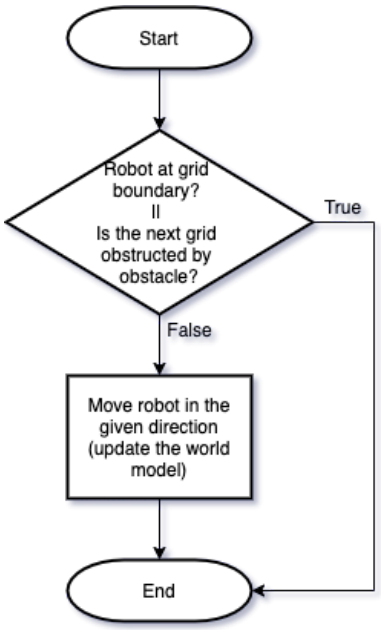

Figure 2.6: Flowchart to validate and move the robot

2.5.2 Source Code

```
/* BEGIN: PLAN move $robot $direction */
PLAN:
{
NAME:
    "Move the given robot one step on the given direction"

DOCUMENTATION:
    "The plan attempts to move the given robot in the given direction for one step, with the
validation,
        - To ensure that the direction to advance does not contain an obstacle.
        - To ensure that the robot does not go out of the boundaries of the grid."

GOAL:
    ACHIEVE move $robot $direction;

BODY:
    RETRIEVE MAX_X $MAX_X;
    RETRIEVE MAX_Y $MAX_Y;

    // Move Robot A
    WHEN: TEST(== $robot "robotA") {
        RETRIEVE robotA $x $y;

        // Check and ensure that the right side does not contain an obstacle and is within
the boundaries
        // The same applies for the following validations
        WHEN: TEST(&& (== $direction "right") (!(FACT obstacle (+ $x 1) $y)) (<= (+ $x 1)
$MAX_X)) {
            EXECUTE println "Robot A: Going right";
            UPDATE (robotA) (robotA (+ $x 1) $y);
        };

        WHEN: TEST(&& (== $direction "left") (!(FACT obstacle (- $x 1) $y)) (>= (- $x 1) 0))
{
            EXECUTE println "Robot A: Going left";
            UPDATE (robotA) (robotA (- $x 1) $y);
        };

        WHEN: TEST(&& (== $direction "up") (!(FACT obstacle $x (+ $y 1))) (<= (+ $y 1)
$MAX_Y)) {
            EXECUTE println "Robot A: Going up";
            UPDATE (robotA) (robotA $x (+ $y 1));
        };

        WHEN: TEST(&& (== $direction "down") (!(FACT obstacle $x (- $y 1))) (>= (- $y 1) 0))
{
            EXECUTE println "Robot A: Going down";
            UPDATE (robotA) (robotA $x (- $y 1));
        };
```

```
        RETRIEVE robotA $x $y;
        EXECUTE println "Robot A: (" $x ", " $y ")";
    };

    // Move Robot B
    // Please refer to the section above as the logic is identical but applicable for Robot B
    WHEN: TEST(== $robot "robotB") {
        RETRIEVE robotB $x $y;

        WHEN: TEST(&& (== $direction "right") (!(FACT obstacle (+ $x 1) $y)) (<= (+ $x 1)
$MAX_X)) {
            EXECUTE println "Robot B: Going right";
            UPDATE (robotB) (robotB (+ $x 1) $y);
        };

        WHEN: TEST(&& (== $direction "left") (!(FACT obstacle (- $x 1) $y)) (>= (- $x 1) 0))
{
            EXECUTE println "Robot B: Going left";
            UPDATE (robotB) (robotB (- $x 1) $y);
        };

        WHEN: TEST(&& (== $direction "up") (!(FACT obstacle $x (+ $y 1))) (<= (+ $y 1)
$MAX_Y)) {
            EXECUTE println "Robot B: Going up";
            UPDATE (robotB) (robotB $x (+ $y 1));
        };

        WHEN: TEST(&& (== $direction "down") (!(FACT obstacle $x (- $y 1))) (>= (- $y 1) 0))
{
            EXECUTE println "Robot B: Going down";
            UPDATE (robotB) (robotB $x (- $y 1));
        };

        RETRIEVE robotB $x $y;
        EXECUTE println "Robot B: (" $x ", " $y ")";
    };

FAILURE:
    EXECUTE println "System: Failed to Move - " $robot;
}
/* END: PLAN move $robot $direction */
```

Excerpt 2.4: Plan to achieve the sub-goal to move the robot

2.5.3 Explanation

Before moving the robot one step forward in the given direction, validations will first be performed to ensure that the robot moving towards the direction will not exceed the boundaries of the grid and collide with an obstacle. After the validation, the fact on the location of the robot will be updated, denoting that the robot had moved one step successfully in the direction specified by the invoker of the plan. Similarly, this plan determines the robot to be moved through the arguments, whereby passing *"robotA"* and *"robotB"* to *$robot* denotes Robot A and B respectively while the direction to move one step is specified by *$direction*.

2.6 Plan: Achieve Collect

2.6.1 Flowchart

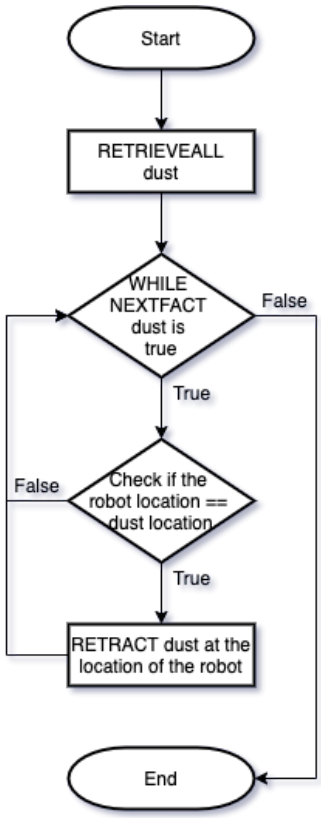

Figure 2.7: Flowchart to check and collect the dusts

2.6.2 Source Code

```
/* BEGIN: PLAN collect $robot */
PLAN:
{
NAME:
    "Check and collect dust at the location of the given robot"

DOCUMENTATION:
    "The plan checks and collects the dust at the location of the given robot.
        - The fact of the collected dust will be retracted as a method of communication
between the two robots."

GOAL:
    ACHIEVE collect $robot;

BODY:

    // Retrieve all the dusts from the world model (fact)
    RETRIEVEALL $FACTS dust $x $y;

    // Check the location of Robot A with the location of the dusts
    // If the robot is at the location of a dust, collect the dust and retract the fact
    WHEN: TEST(== $robot "robotA") {
        RETRIEVE robotA $rX $rY;
        WHILE: NEXTFACT $FACTS dust $x $y {
            WHEN: TEST(&& (== $x $rX) (== $y $rY)) {
                RETRACT dust $x $y;
                EXECUTE println "Robot A: Dust Collected at (" $x ", " $y ")";
            };
        };
    };

    WHEN: TEST(== $robot "robotB") {
        RETRIEVE robotB $rX $rY;
        WHILE: NEXTFACT $FACTS dust $x $y {
            WHEN: TEST(&& (== $x $rX) (== $y $rY)) {
                RETRACT dust $x $y;
                EXECUTE println "Robot B: Dust Collected at (" $x ", " $y ")";
            };
        };
    };

FAILURE:
    EXECUTE println "System: Failed to Collect Dust - " $robot;
}
/* END: PLAN collect $robot */
```

Excerpt 2.5: Plan to achieve the sub-goal to check and collect the dusts

2.6.3 Explanation

When the robots advance one step, the plan to check the location of the dusts and collect if the robots are above the dust. The linear search was performed by looping through the list of dusts in the world model. If the robots were not on any of the dusts in the world model, the plan terminates without performing any action. Otherwise, the fact on the location of the dust will be retracted to communicate on the dust has been cleaned with the other robots. In a nutshell, this plan forms the backbone of the robots to achieve the objectives of the robots to clean up all the dusts by checking and collecting the dusts that are possible to be collected.

2.7 Plan: Achieve Store

2.7.1 Flowchart

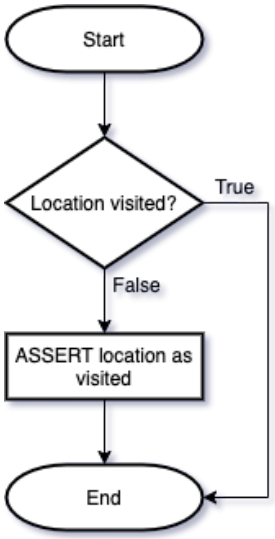

Figure 2.8: Flowchart to mark a location as visited

2.7.2 Source Code

```
/* BEGIN: PLAN store $robot */
PLAN:
{
NAME:
    "Store the visited location of the given robot"

DOCUMENTATION:
    "The plan stores the location visited by the given robot in the world model (fact)"

GOAL:
    ACHIEVE store $robot;

BODY:

    // Check if the location was previously added to the visited List
    // If it is not, then add it to the visited list
    WHEN: TEST(== $robot "robotA") {
        RETRIEVE robotA $x $y;
        WHEN: TEST(!(FACT visited $x $y)) {
            ASSERT visited $x $y;
        };
    };

    WHEN: TEST(== $robot "robotB") {
        RETRIEVE robotB $x $y;
        WHEN: TEST(!(FACT visited $x $y)) {
            ASSERT visited $x $y;
        };
    };

FAILURE:
    EXECUTE println "System: Failed to Store Visited Location - " $robot;
}
/* END: PLAN store $robot */
```

Excerpt 2.6: Plan to achieve the sub-goal to mark a location as visited

2.7.3 Explanation

The plan simply checks if a location was previously visited using the *FACT visited $x $y* which will return either a Boolean value denoting if the fact exists. Hence, if the fact does not exist, the program adds the location into the list of visited coordinates using *ASSERT visited $x $y*. This step is also crucial in the navigation of the robots within the grid as marking a location as visited allows the robots to effectively scan through the grid. This is because the robot avoid repeatedly scanning the locations which have previously been visited which can waste the time limit, resulting in a wide range of grid cells missed by the end of the program.

2.8 Plan: Achieve Check

2.8.1 Flowchart

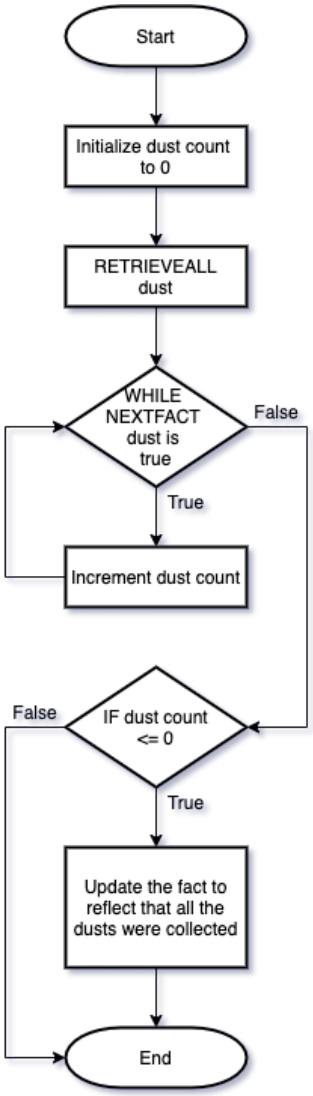

Figure 2.9: Flowchart to check if all the dusts were cleaned

2.8.2 Source Code

```
/* BEGIN: PLAN check */
PLAN:
{
NAME:
    "Check whether if all the dusts were cleaned"

DOCUMENTATION:
    "Check and update if all the dusts were cleaned in the world model (fact)."

GOAL:
    ACHIEVE check;

BODY:

    // Go through the list of dusts
    ASSIGN $dustCount 0;
    RETRIEVEALL $FACTS dust $x $y;
    WHILE: NEXTFACT $FACTS dust $x $y {
        ASSIGN $dustCount (+ $dustCount 1);
    };

    // If the dust list is empty, update the dustCleaned fact to true
    WHEN: TEST(<= $dustCount 0) {
        UPDATE (dustCleaned) (dustCleaned "true");
    };

FAILURE:
    EXECUTE println "System: Failed to Check If the Dusts were Cleaned";
}
/* END: PLAN check */
```

Excerpt 2.7: Plan to achieve the sub-goal to check if all the dusts were cleaned

2.8.3 Explanation

The plan checks and update whether if the dusts in the grid were all collected. From the source code, all the dusts were retrieved using *RETRIEVEALL*, then the plan loops through the dusts using *NEXTFACT $FACTS dust $x $y* and count the number of dusts remaining. If the number of dusts remaining was zero, then the world model (fact) should be updated to denote that no dust is present in the grid currently, which in turn allows the robots to exit from the grid. This plan will be performed for every iteration to ensure that there is at least one dust remaining in the grid, only then the robots will continue to search for and clean up the dusts in the grid.

In conclusion, this plan finalize the sub-goals to achieve the desire to clean up the dusts. As a recap over the flow to clean up the dusts, firstly, the top-level plan, *Clean* from section 2.3 will first be invoked. Then, the top-level plan then invokes the *Run* plan from section 2.4 which contains the algorithms for a robot to perform the tasks. Through the plan, the further sub-goals for the robots to make a move on the grid, check and collect the dusts and mark the location as visited will be performed by invoking the *Move*, *Collect* and *Store* plans respectively. The following sections will discuss on the plans to achieve the top-level goal (desire) to navigate the robot to the exit after completing the dust cleaning process. As the logics were mostly similar in the following sections, the flowchart section will be omitted.

2.9 Plan: Achieve Return

2.9.1 Source Code

```
/* BEGIN: PLAN return */
PLAN:
{
NAME:
    "Navigate the robots to the exit"

DOCUMENTATION:
    "The plan navigates the robots to the exit after,
        - All the dusts were cleaned.
        - The time limit to clean had expired.
    If there exists obstacles blocking the pathway to the exit,
        - The plan continues to try to navigate the robots to the exit.
        - The plan stops after the time limit had expired, whereby the exit was not
achievable in time."

GOAL:
    ACHIEVE return;

BODY:

    // Assign the maximum iterations (time limit) to try to navigate the robots to the exit
    EXECUTE println "System: Exit Initiated";
    RETRIEVE MAX_X $MAX_X;
    RETRIEVE MAX_Y $MAX_Y;
    ASSIGN $MAX_TRIES (* $MAX_X $MAX_Y 10);
    ASSIGN $iteration 0;

    // Retrieve the location of the robots and exit
    RETRIEVE robotA $aX $aY;
    RETRIEVE robotB $bX $bY;
    RETRIEVE exit $exitX $exitY;

    // Loop to keep the robot walking towards the exit until,
    //  - Both robots have exited.
    //  - The time limit had expired.
    WHILE: TEST(&& (|| (!= $aX $exitX) (!= $aY $exitY) (!= $bX $exitX) (!= $bY $exitY)) (<
$iteration $MAX_TRIES)) {

        // Advances one step for both robots towards the exit
        PERFORM quit "robotA";
        PERFORM quit "robotB";

        // Retrieve the location of the robots
        RETRIEVE robotA $aX $aY;
        RETRIEVE robotB $bX $bY;
        ASSIGN $iteration (+ $iteration 1);
```

```
    };

    EXECUTE println "System: Terminated";

FAILURE:
    EXECUTE println "System: Failed to Exit";
}
/* END: PLAN return */
```

Excerpt 2.8: Plan to achieve the top-level goal, return which brings the robots to the exit

2.9.2 Explanation

The logical implementation of the plan for the robots to return to the exit was largely similar to the plan to clean the dusts in the grid in section 2.3. Firstly, a maximum number of tries that can be made by the robots were calculated as the ten times of the number of cells in the grid, which was the statement *ASSIGN $MAX_TRIES (* $MAX_X $MAX_Y 10)* in the code above. Then, until both robots exit successfully or the time limit had expired, the loop will continue to invoke the plan, *quit* which will try to navigate the robot to the exit by using similar heuristics aforementioned in section 2.4 which will be discussed in depth in the next section.

2.10 Plan: Achieve Quit

2.10.1 Source Code

```
/* BEGIN: PLAN quit $robot */
PLAN:
{
NAME:
    "Decide and advances the selected robot for one step towards the exit"

DOCUMENTATION:
    "The plan attempts to navigate the robot towards the exit using heuristics,
        - The heuristic tries to move the robot along the obstacles or boundaries.
        - The movement prioritize the horizontal motion towards the right.
        - This helps in ensuring that the robot gets closer to the exit even when blocked by
obstacles."

GOAL:
    ACHIEVE quit $robot;

BODY:
    RETRIEVE MAX_X $MAX_X;
    RETRIEVE MAX_Y $MAX_Y;
    RETRIEVE exit $exitX $exitY;

    // Robot A selected
    WHEN: TEST(== $robot "robotA") {
        RETRIEVE robotA $x $y;

        // Check if the robot is already at the exit
        // Perform the movement only if the robot is not already at the exit
        WHEN: TEST(|| (!= $x $exitX) (!= $y $exitY)) {


            /* BEGIN: Robot Movement Logic */
            // Variables to store the valid moves
            ASSIGN $up "true";
            ASSIGN $down "true";
            ASSIGN $left "true";
            ASSIGN $right "true";
            ASSIGN $moved "false";

            // Check which are the valid moves
            WHEN: TEST(|| (<= $x 0) (FACT robotAVisited (- $x 1) $y) (FACT obstacle (- $x 1)
$y)) {
                ASSIGN $left "false";
            };

            WHEN: TEST(|| (>= $x $MAX_X) (FACT robotAVisited (+ $x 1) $y) (FACT obstacle (+
$x 1) $y)) {
```

```
                ASSIGN $right "false";
            };

            WHEN: TEST(|| (<= $y 0) (FACT robotAVisited $x (- $y 1)) (FACT obstacle $x (- $y
1))) {
                ASSIGN $down "false";
            };

            WHEN: TEST(|| (>= $y $MAX_Y) (FACT robotAVisited $x (+ $y 1)) (FACT obstacle $x
(+ $y 1))) {
                ASSIGN $up "false";
            };

            // If there is no other move that leads better to the exit,
            // Perform a random move
            WHEN: TEST(&& (== $up "false") (== $down "false") (== $left "false") (== $right
"false")) {

                EXECUTE println "Robot A: No Better Move to Exit, Attempting a Random Move";

                // If the robot is at the boundaries, move in the opposite direction
                WHEN: TEST(<= $x 0) {
                    UPDATE (robotADirection) (robotADirection "right");
                    ASSIGN $moved "true";
                };

                WHEN: TEST(<= $y 0) {
                    UPDATE (robotADirection) (robotADirection "up");
                    ASSIGN $moved "true";
                };

                WHEN: TEST(>= $x $MAX_X) {
                    UPDATE (robotADirection) (robotADirection "left");
                    ASSIGN $moved "true";
                };

                WHEN: TEST(>= $y $MAX_Y) {
                    UPDATE (robotADirection) (robotADirection "down");
                    ASSIGN $moved "true";
                };

                // Attempt to make a random move if the robot is not at the boundaries
                WHEN: TEST(== $moved "false") {
                    DO_ANY {
                        UPDATE (robotADirection) (robotADirection "right");
                        ASSIGN $moved "true";
                    } {
                        UPDATE (robotADirection) (robotADirection "up");
                        ASSIGN $moved "true";
                    } {
                        UPDATE (robotADirection) (robotADirection "left");
```

```
                    ASSIGN $moved "true";
                } {
                    UPDATE (robotADirection) (robotADirection "down");
                    ASSIGN $moved "true";
                };
            };
        };

        // The following logic helps to navigate the robot in moving along the obstacles
or boundaries
        // Attempt to move towards the right if possible (prioritizing the movement to
the right)
        WHEN: TEST(&& (== $right "true") (== $moved "false")) {
            UPDATE (robotADirection) (robotADirection "right");
            ASSIGN $moved "true";
        };

        // Otherwise, attempt to move upwards
        WHEN: TEST(&& (== $up "true") (== $moved "false")) {
            UPDATE (robotADirection) (robotADirection "up");
            ASSIGN $moved "true";
        };

        // Otherwise, try to move towards the left
        WHEN: TEST(&& (== $left "true") (== $moved "false")) {
            UPDATE (robotADirection) (robotADirection "left");
            ASSIGN $moved "true";
        };

        // If all else fail, move towards the bottom
        // This is a heuristic in hope to find a path to the exit
        WHEN: TEST(&& (== $down "true") (== $moved "false")) {
            UPDATE (robotADirection) (robotADirection "down");
            ASSIGN $moved "true";
        };
        /* END: Robot Movement Logic */


        // Move the robot in the decided direction for one step
        // Mark the current location as visited for the robot
        RETRIEVE robotADirection $direction;
        ASSERT robotAVisited $x $y;
        PERFORM move "robotA" $direction;

        RETRIEVE robotA $x $y;
        WHEN: TEST(&& (== $x $exitX) (== $y $exitY)) {
            EXECUTE println "Robot A: Exited at (" $exitX ", " $exitY ")";
        };
    };
};
```

```
    // Please refer to the comments in the section above
    // The code is identical but applicable for Robot B
    WHEN: TEST(== $robot "robotB") {
        RETRIEVE robotB $x $y;

        WHEN: TEST(|| (!= $x $exitX) (!= $y $exitY)) {
            ASSIGN $up "true";
            ASSIGN $down "true";
            ASSIGN $left "true";
            ASSIGN $right "true";
            ASSIGN $moved "false";

            WHEN: TEST(|| (<= $x 0) (FACT robotBVisited (- $x 1) $y) (FACT obstacle (- $x 1)
$y)) {
                ASSIGN $left "false";
            };

            WHEN: TEST(|| (>= $x $MAX_X) (FACT robotBVisited (+ $x 1) $y) (FACT obstacle (+
$x 1) $y)) {
                ASSIGN $right "false";
            };

            WHEN: TEST(|| (<= $y 0) (FACT robotBVisited $x (- $y 1)) (FACT obstacle $x (- $y
1))) {
                ASSIGN $down "false";
            };

            WHEN: TEST(|| (>= $y $MAX_Y) (FACT robotBVisited $x (+ $y 1)) (FACT obstacle $x
(+ $y 1))) {
                ASSIGN $up "false";
            };

            WHEN: TEST(&& (== $up "false") (== $down "false") (== $left "false") (== $right
"false")) {

                EXECUTE println "Robot B: No Better Move to Exit, Attempting a Random Move";

                WHEN: TEST(<= $x 0) {
                    UPDATE (robotBDirection) (robotBDirection "right");
                    ASSIGN $moved "true";
                };

                WHEN: TEST(<= $y 0) {
                    UPDATE (robotBDirection) (robotBDirection "up");
                    ASSIGN $moved "true";
                };

                WHEN: TEST(>= $x $MAX_X) {
                    UPDATE (robotBDirection) (robotBDirection "left");
                    ASSIGN $moved "true";
                };
```

```
    WHEN: TEST(>= $y $MAX_Y) {
        UPDATE (robotBDirection) (robotBDirection "down");
        ASSIGN $moved "true";
    };

    WHEN: TEST(== $moved "false") {
        DO_ANY {
            UPDATE (robotBDirection) (robotBDirection "right");
            ASSIGN $moved "true";
        } {
            UPDATE (robotBDirection) (robotBDirection "up");
            ASSIGN $moved "true";
        } {
            UPDATE (robotBDirection) (robotBDirection "left");
            ASSIGN $moved "true";
        } {
            UPDATE (robotBDirection) (robotBDirection "down");
            ASSIGN $moved "true";
        };
    };
};

WHEN: TEST(&& (== $right "true") (== $moved "false")) {
    UPDATE (robotBDirection) (robotBDirection "right");
    ASSIGN $moved "true";
};

WHEN: TEST(&& (== $up "true") (== $moved "false")) {
    UPDATE (robotBDirection) (robotBDirection "up");
    ASSIGN $moved "true";
};

WHEN: TEST(&& (== $left "true") (== $moved "false")) {
    UPDATE (robotBDirection) (robotBDirection "left");
    ASSIGN $moved "true";
};

WHEN: TEST(&& (== $down "true") (== $moved "false")) {
    UPDATE (robotBDirection) (robotBDirection "down");
    ASSIGN $moved "true";
};

RETRIEVE robotBDirection $direction;
ASSERT robotBVisited $x $y;
PERFORM move "robotB" $direction;

RETRIEVE robotB $x $y;
WHEN: TEST(&& (== $x $exitX) (== $y $exitY)) {
    EXECUTE println "Robot B: Exited at (" $exitX ", " $exitY ")";
};
```

```
        };
    };

FAILURE:
    EXECUTE println "System: Failed to Quit - " $robot;
}
/* END: PLAN quit $robot */
```

Excerpt 2.9: Plan to achieve the sub-goal to make a best move towards the exit

2.10.2 Explanation

The navigation algorithm in leading the robots toward the exit is similar to the algorithm used in section 2.4 to perform a move with minor adjustments. By referring to the flowchart in Figure 2.3, the robots will attempt to make a move, prioritizing the movement towards the top right corner, which is the location of the exit. However, the list of visited locations will be separated for both robots as the robots should not be interfering each other when attempting to move towards the exit. The algorithm developed ensures that the robots selects a best cell which was not visited, within the boundaries and not blocked by obstacles to help in exploring a wider range of cells in the grid so that there is a higher chance to reach the location of the exit.

From the heuristics involved, the robots should gradually move towards the exit even with the obstacles in place. Similarly, when a best move cannot be determined, the robots will move randomly ignoring whether if a cell was previously visited in hope of searching for a path that leads to the exit. Should the obstacles block the exit completely, the robots shall go into an infinite loop while retrying to search for the exit. Under such circumstance, the time limit kicks in, whereby the robots give up searching and the exit process fails. This mimics the real-world scenario whereby the base station is blocked and the robots keep trying to search for the base station until they eventually run out of battery and the return process fails.

2.11 Conclusion

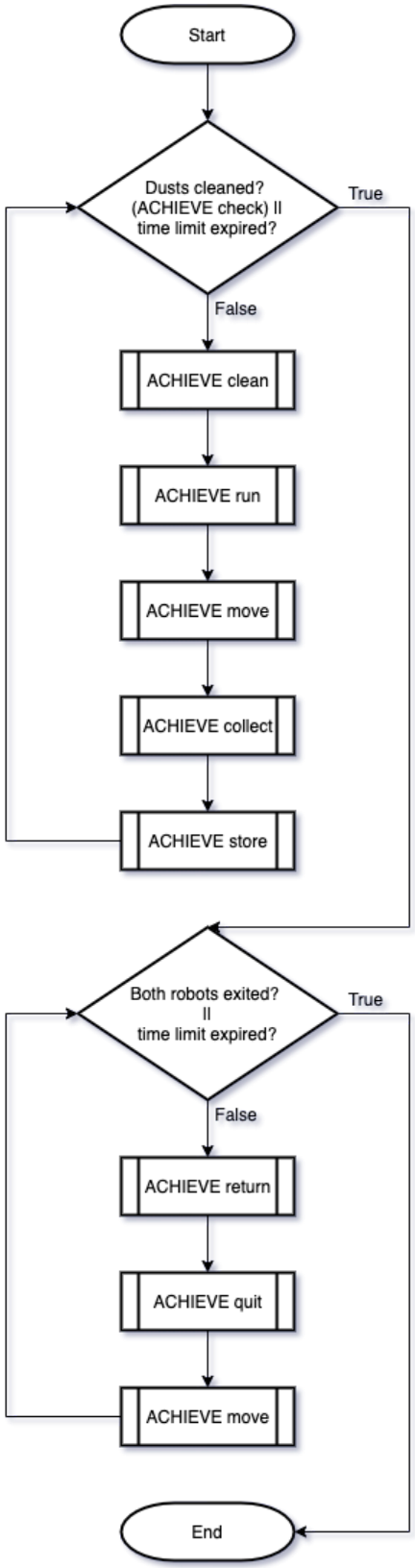

Figure 2.10: Flowchart to show the plan invocation sequence of the program

In conclusion, the flowchart above shows the high-level view of the plan invocation, in which the first part shows the plans involved in achieving the goal to clean up the dusts in the grid. Firstly, the plan achieving *clean* will be invoked, through the plan, the plans, *check*, *run*, *move*, *collect*, and *store* are invoked in order by using *PERFORM*. The plans check if the all the dusts in the grid were cleaned, attempt to run the robots by deciding an adjacent cell to move to, then further invoking *move*, *collect* and *store* to move the robots towards the selected direction, check and collect the dusts if the robot is above a dust and mark the location as visited.

Besides that, the invocation sequence of the plans for the top-level goal to navigate the robot to exit from the grid is shown at the bottom part of the flowchart. The plan achieving *return* will first be called, then the plan further invokes the plan to achieve *quit* to decide the best move to take while advancing towards the exit. Then, the plan achieving *move* will be further invoked to update the world model on the location of the robots, denoting that the robots have moved. The explanation above concludes the sequence of the plans being invoked in the process of achieving the goals for the robots to clean up the dusts and navigate to the exit.

3.0 Testing and Results

3.1 Introduction

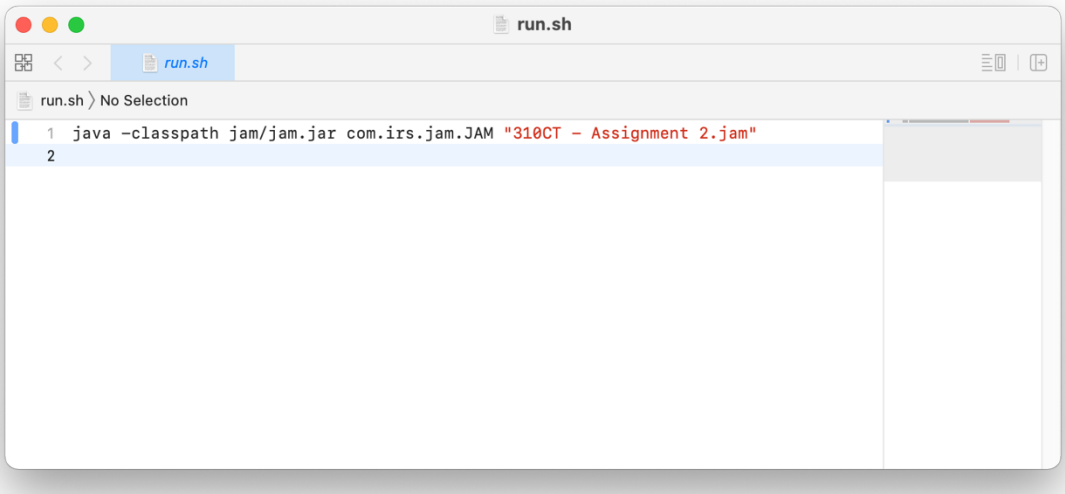

Figure 3.1: Content of run.sh shell script

To simplify the process of executing the JAM code, a shell script was written with the content shown above. In the testing below, the manipulative variables will include the dusts, obstacles, initial location of the robots and size of the grid, etc. For each of the manipulative variables, at most three test cases will be included to account for different scenarios that may happen during execution. The initial placement of the dusts, obstacles, robots and exit will be plotted on a graph, followed by the output from executing the code, then an explanation will be included on instances such as the turn taken by the robots, dusts collection, etc. if necessary. As the output can be very lengthy, only the important instances will be included in the report. On a side note, the *Program Output* header will be marked as *Excerpted* if the screenshots contain only the important sections instead of showing all the output from the start till the end.

3.2 Test Cases (Location of the Dusts)

3.2.1 Test Case 1

3.2.1.1 Initial Setting

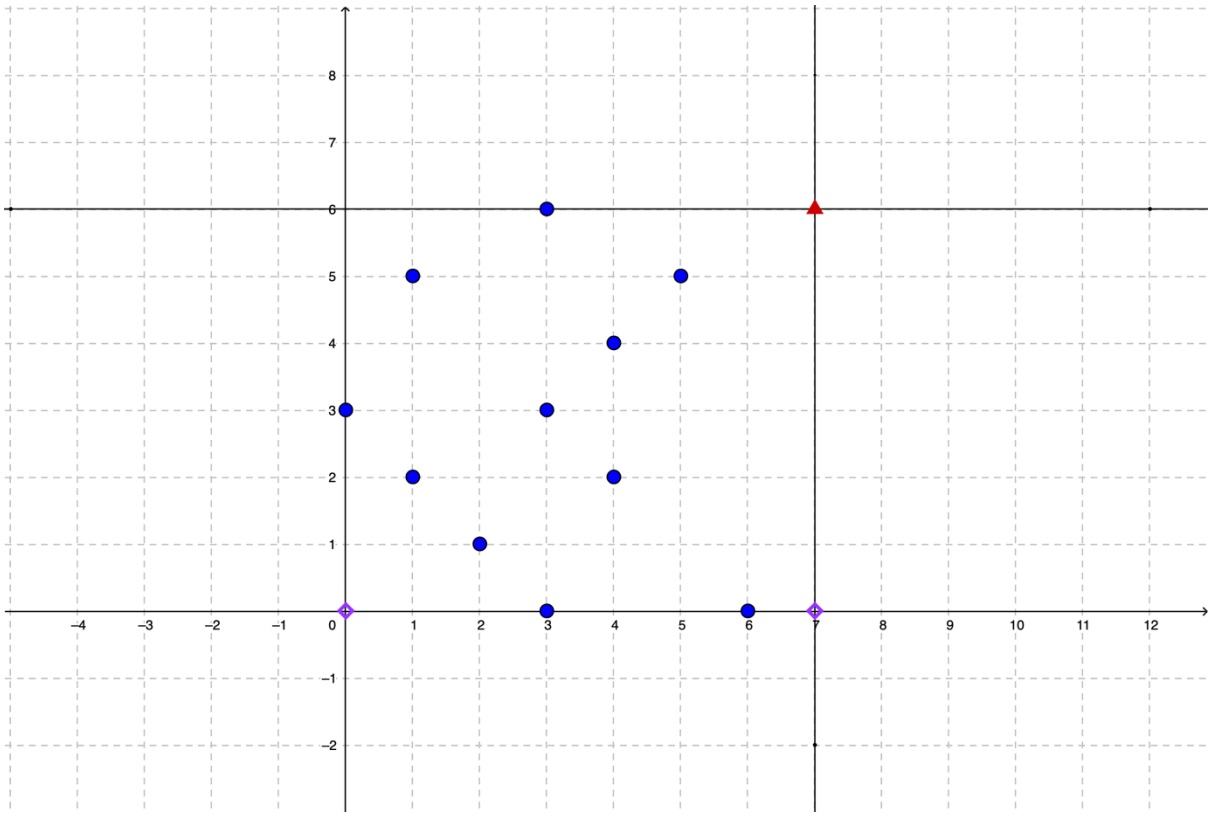

Figure 3.2: Initial placement of the dusts, robots and exit

The first test case focuses on the default 11 dusts given in the question, without obstacles in the middle of the grid. The blue points on the grid denotes the dusts to be collected, while the purple diamonds at (0, 0) and (7, 0) are the initial location of the two robots. Lastly, the red triangle at (7, 6) denotes the exit on the grid which must be reached by the robots after collecting all the dusts successfully. The actions taken by the robots will be discussed in the following section which includes the screenshots after running the program with the settings.

3.2.1.2 Program Output

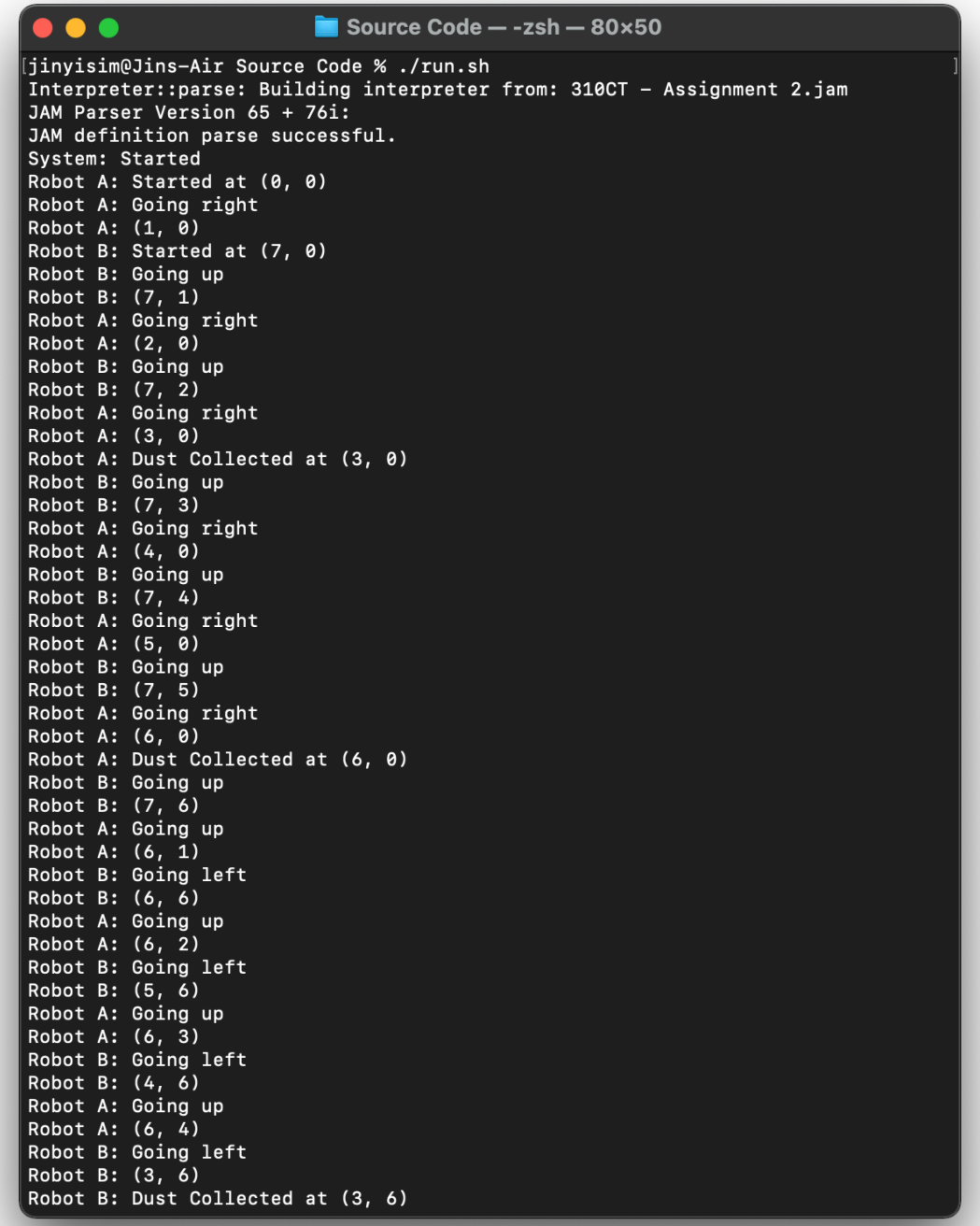

Figure 3.3: Test case 1 output (1)

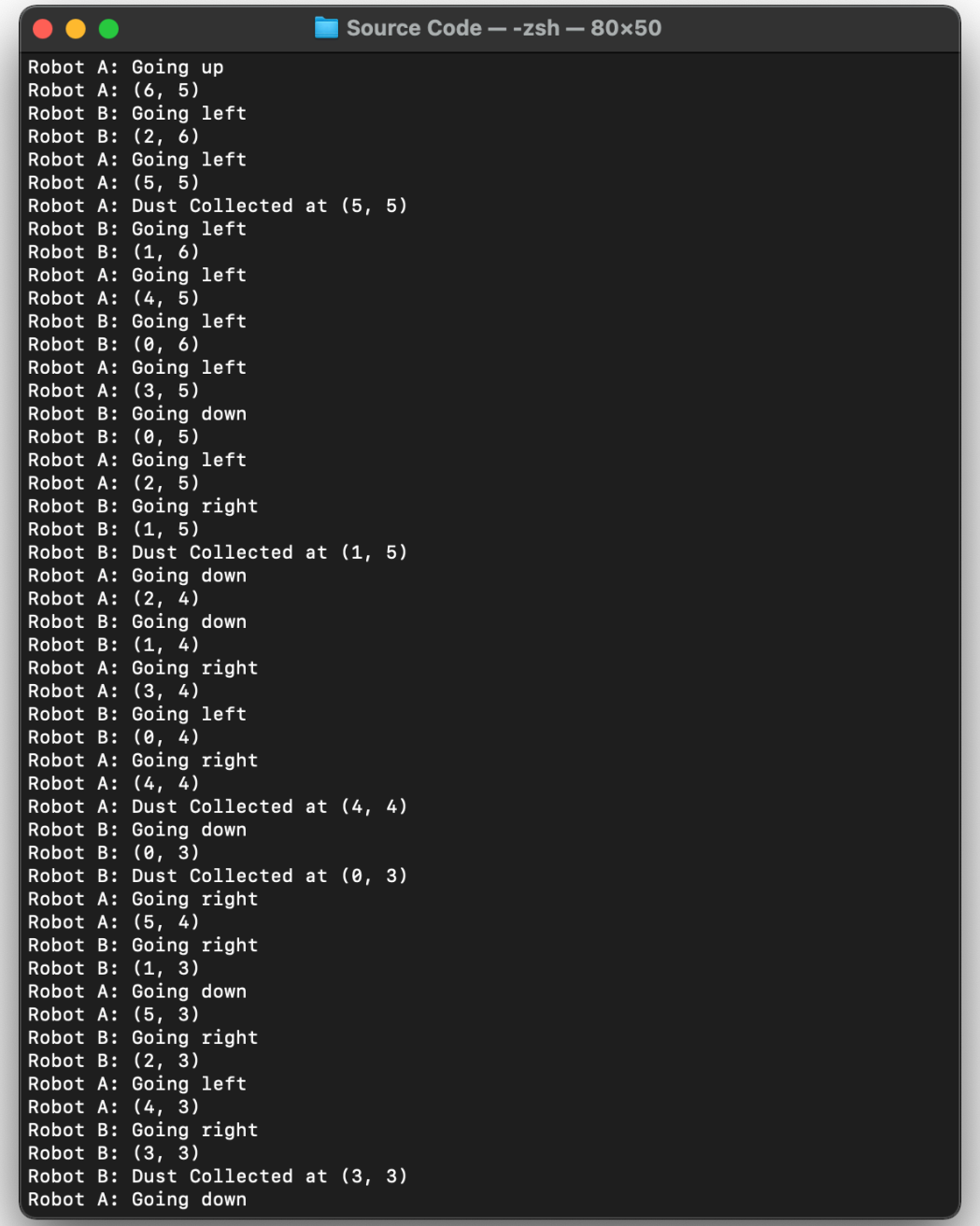

Figure 3.4: Test case 1 output (2)

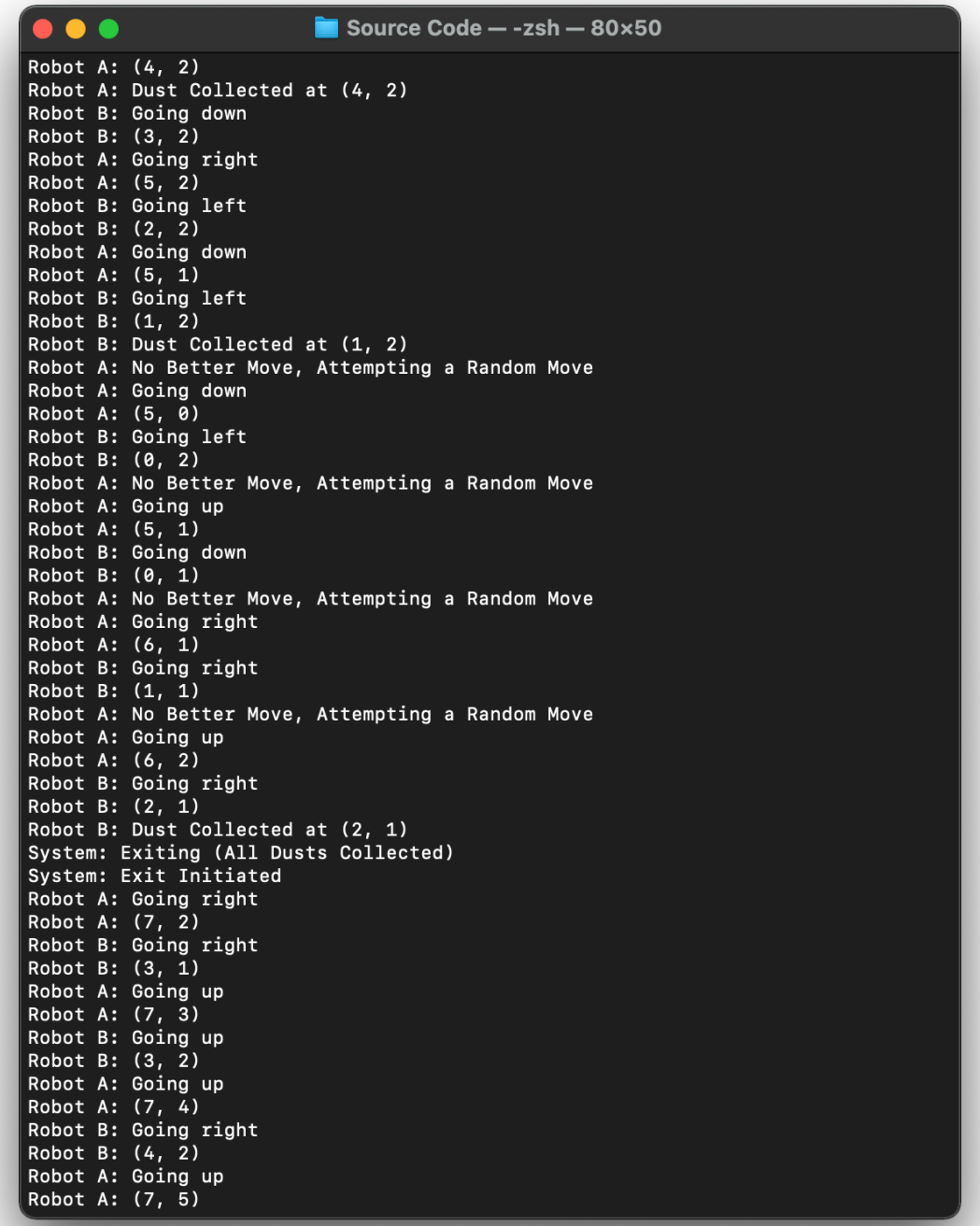

Figure 3.5: Test case 1 output (3)

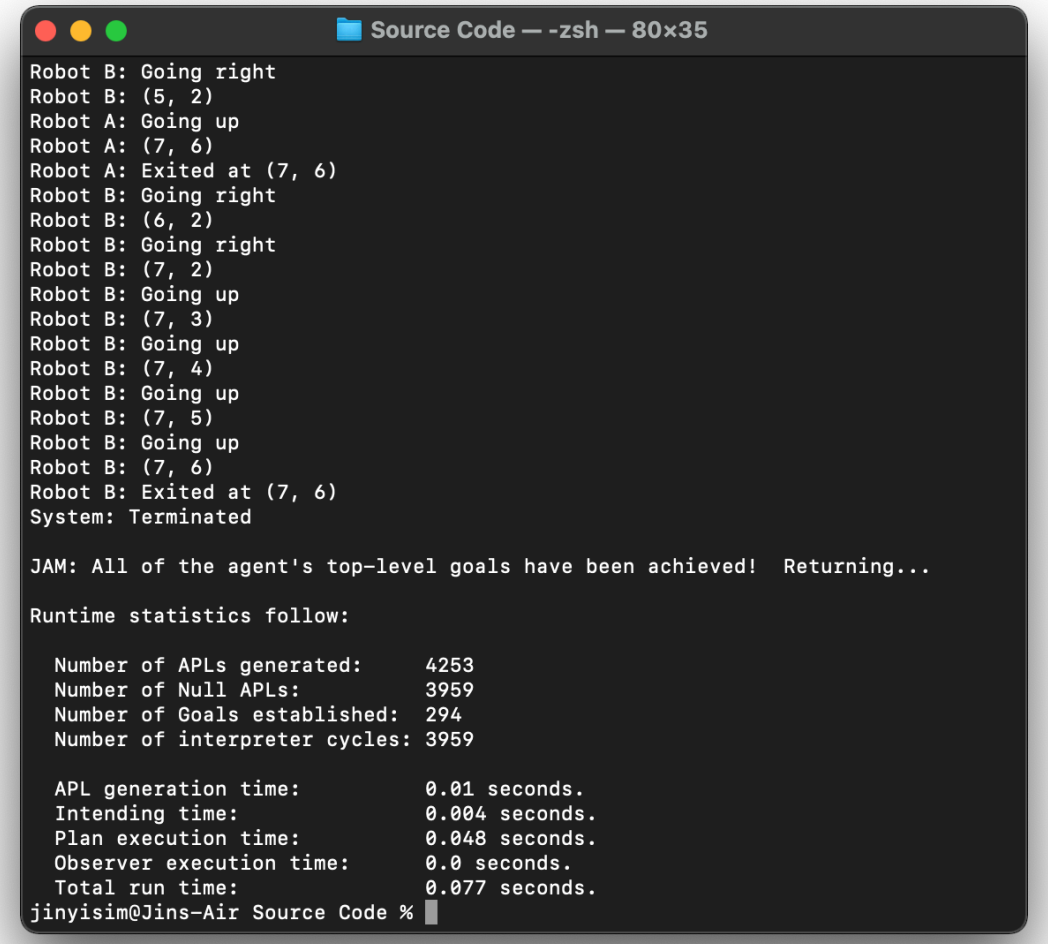

Figure 3.6: Test case 1 output (4)

3.2.1.3 Explanation

As discussed previously, whereby the algorithm to navigate the robot around the grid will prioritize the movement in the following order: right, up, left and down. This process is guaranteed to be deterministic on the first run as all the grid cells are unvisited and the algorithm will always find a best move to make. From Figure 3.3 and Figure 3.4, it was clear that the pattern of the paths taken by robot A and B adhere to the explanation of the algorithm. Firstly, the robot A starts at (0, 0), then gradually move to the right, while robot B starts at (7, 0) and move upwards over the iterations. Note that the robot A moves from (0, 0) to (6, 0), then proceed to (6, 1) instead of (7, 0). This is because robot B first starts at (7, 0), marking (7, 0) as visited, which means robot A should not duplicate and revisit the cell that was visited unless robot A faces a dead end with no better choice other than revisiting the visited cells in the grid.

The pattern repeats for both robots to continue visiting all the unvisited cells on the first round. Then, on the third screenshot, there is no better move that can be made by robot A, hence a random move was made. Lastly, after collecting all the dusts as shown in the third screenshot, the goal to clean is considered to be achieved. By then, the goal to exit from the grid will be initiated. The procedures were shown in the last two screenshots, whereby robot A and B both attempts to make a move in every iteration towards the exit located at (7, 6). In short, the output above shows the steps taken by the robots and exit after cleaning all the dusts.

3.2.2 Test Case 2

3.2.2.1 Initial Setting

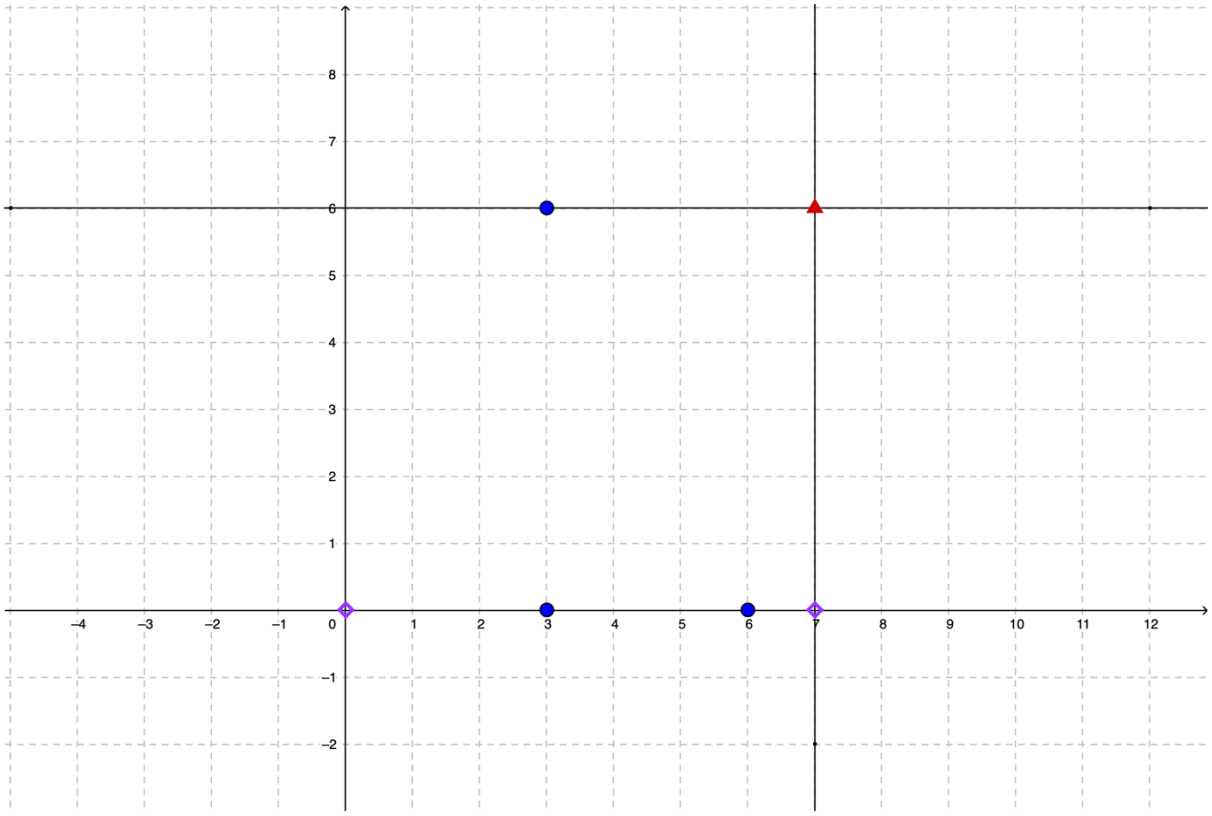

Figure 3.7: Initial placement of the dusts, robots and exit

To show the robots handling the dusts at (3, 0), (6, 0) and (3, 6) which should be done fairly quickly by the cooperating robots since the dusts were placed on the path of the robots. Once the robots completed collecting all the dusts, the goal to exit from the grid should be initiated immediately which leads to the robots completing the entire operations without the possibility of navigating to a point where there is no better option to make for the next move.

3.2.2.2 Program Output

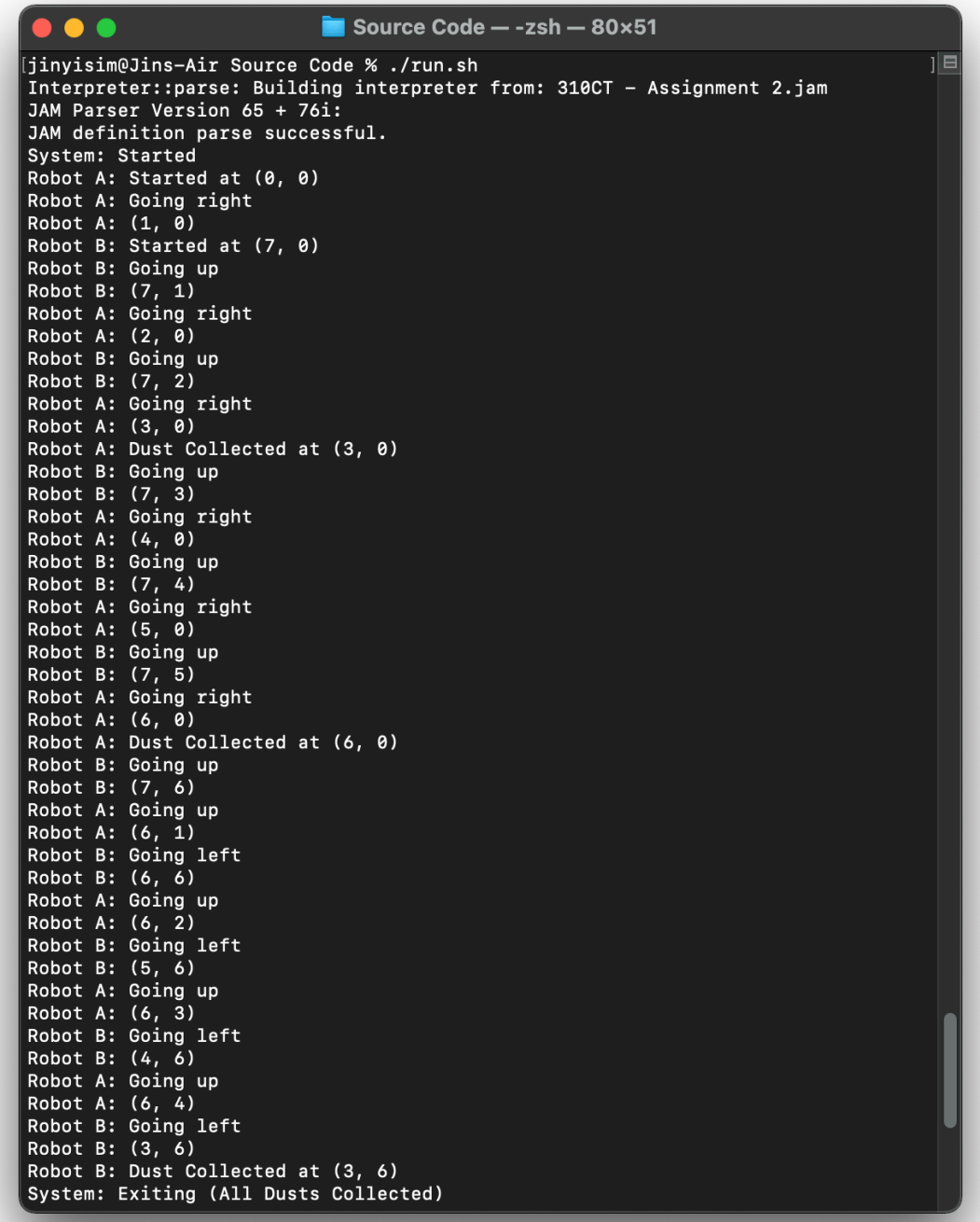

Figure 3.8: Test case 2 output (1)

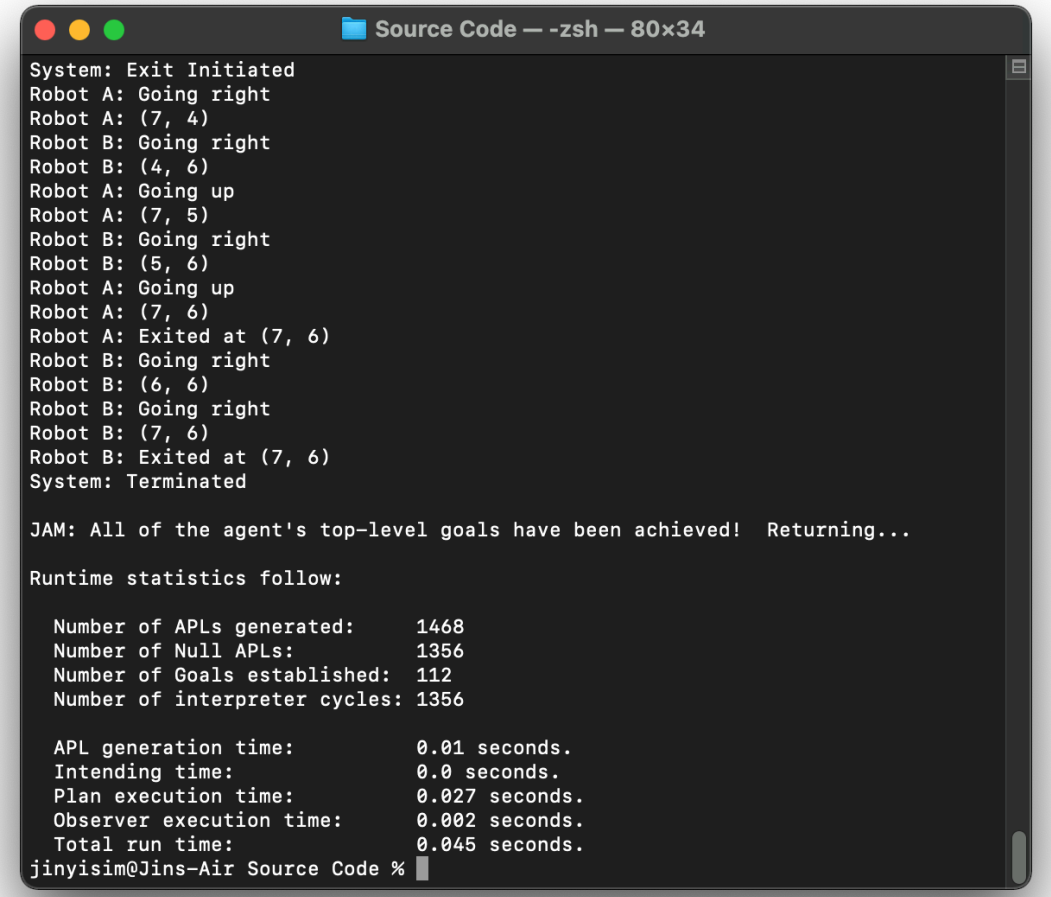

Figure 3.9: Test case 2 output (2)

3.2.3 Test Case 3

3.2.3.1 Initial Setting

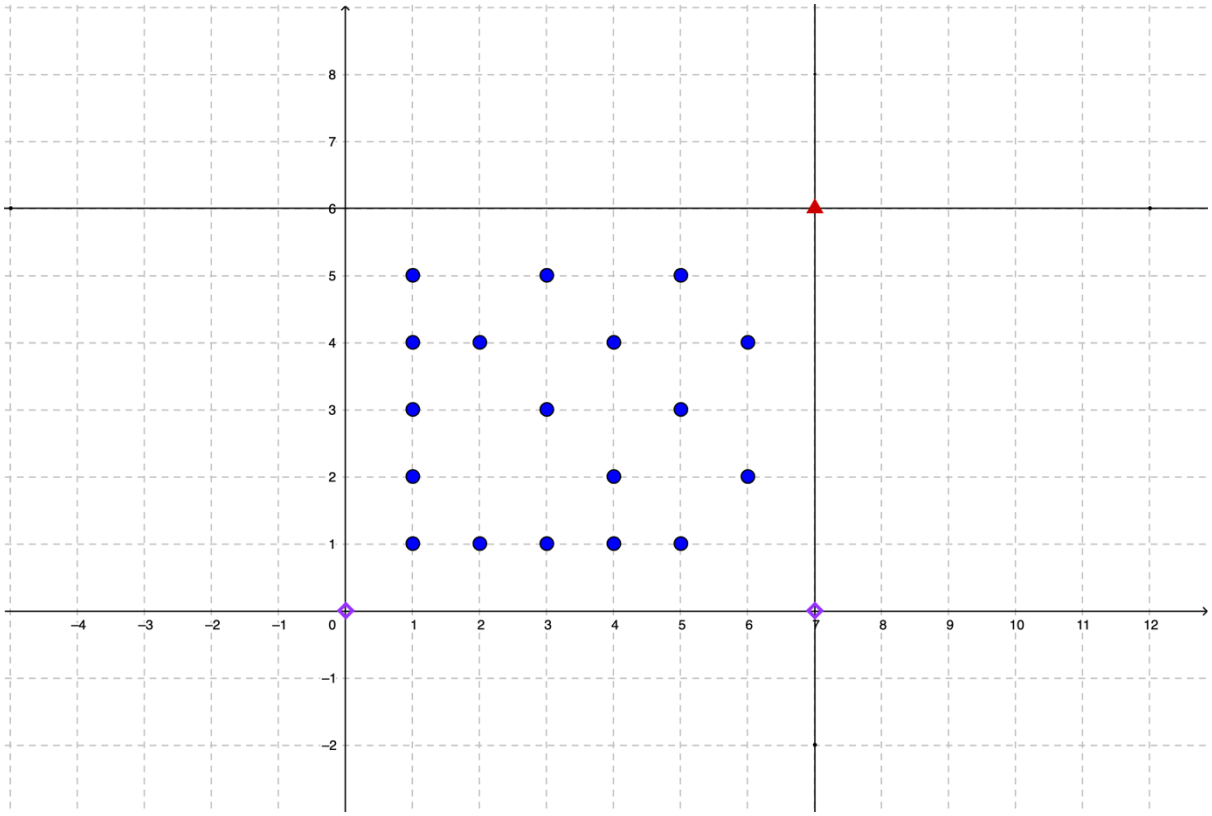

Figure 3.10: Initial placement of the dusts, robots and exit

Without obstacles on the grid, the robots should collect all the dusts on the grid before the running out of time even with a grid full of dusts. The grid above contains 18 dusts placed at random around the grid, which means that the robots should be able to move to collect all the dusts deterministically. The output below shows the result in line with the expectation.

3.2.3.2 Program Output

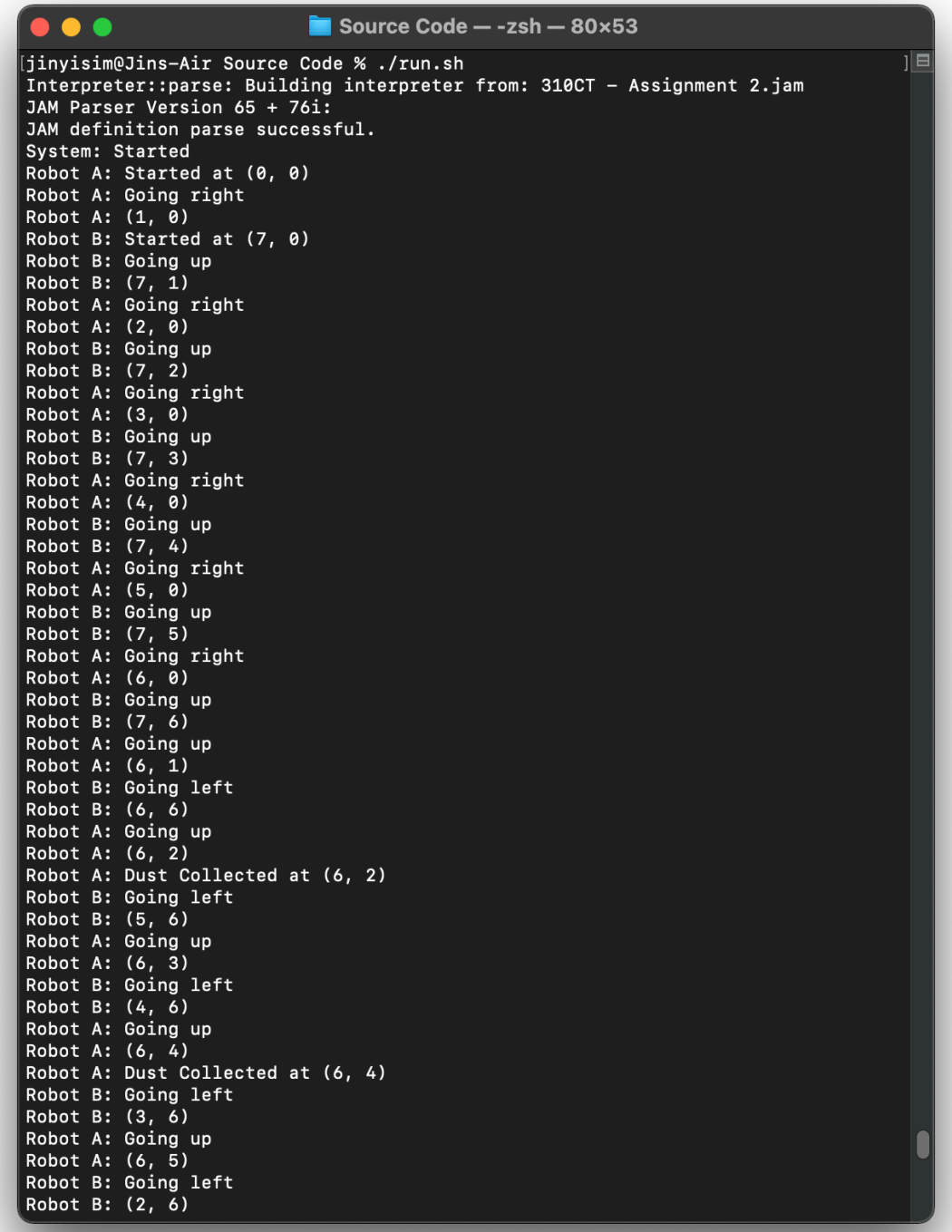

Figure 3.11: Test case 3 output (1)

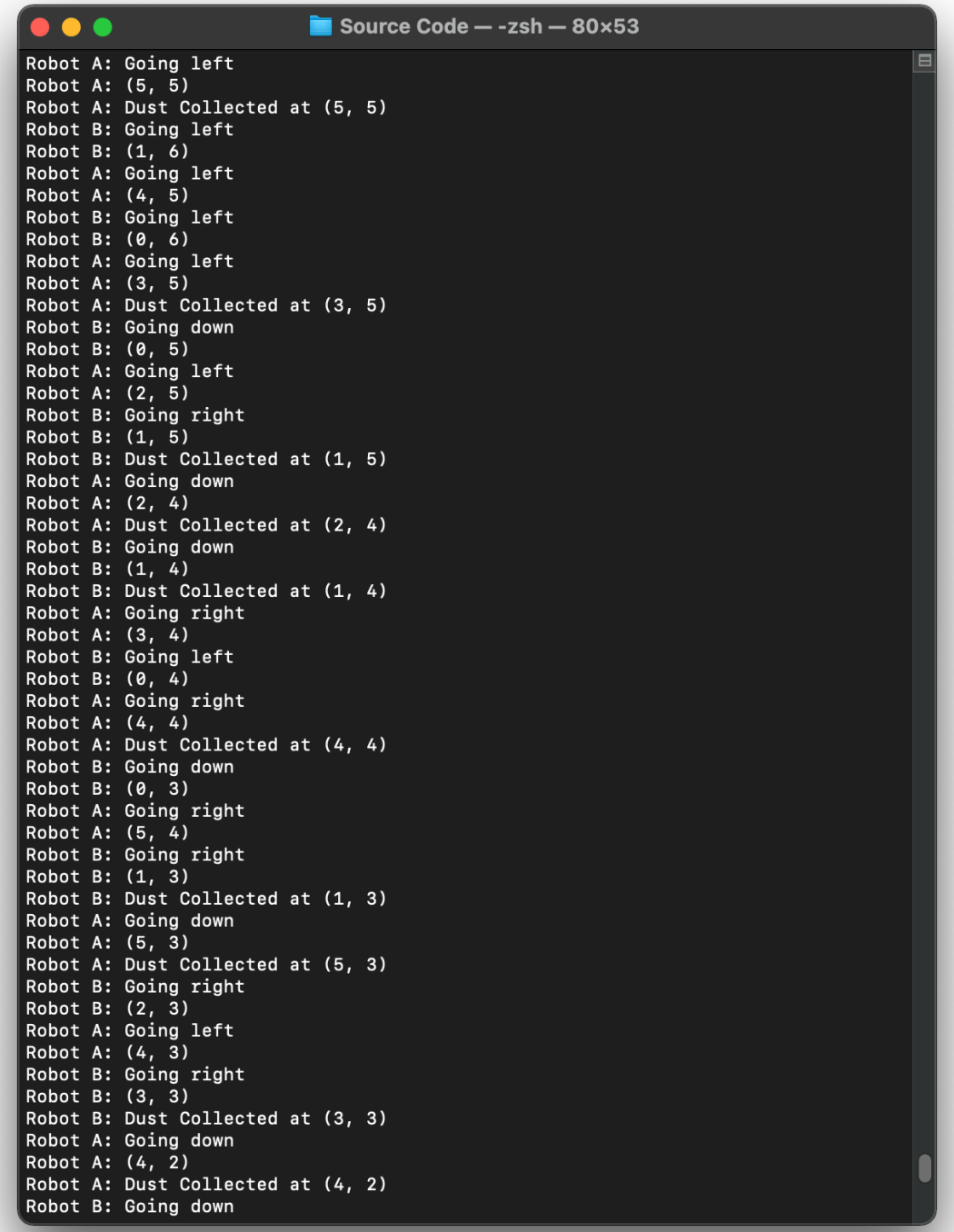

Figure 3.12: Test case 3 output (2)

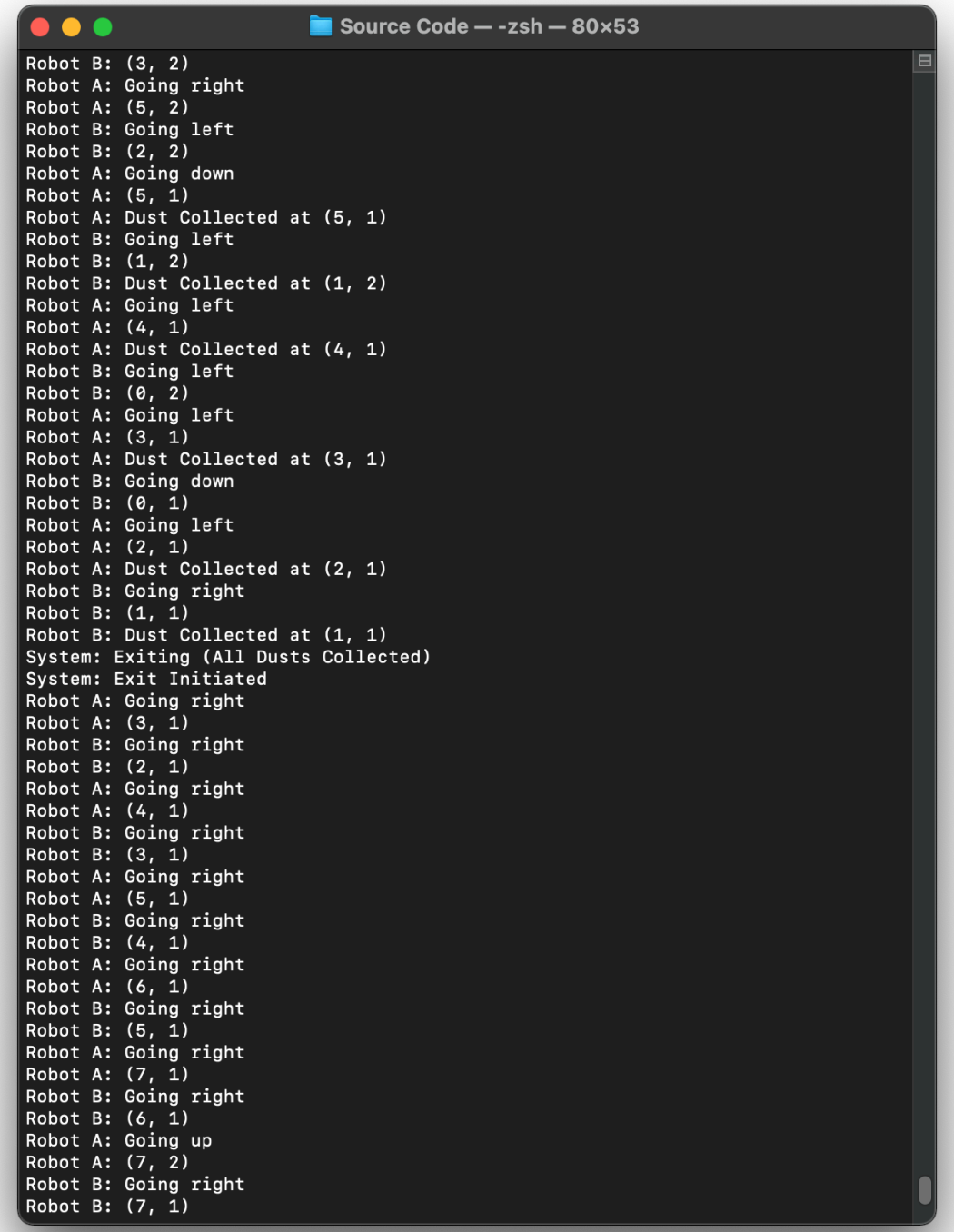

Figure 3.13: Test case 3 output (3)

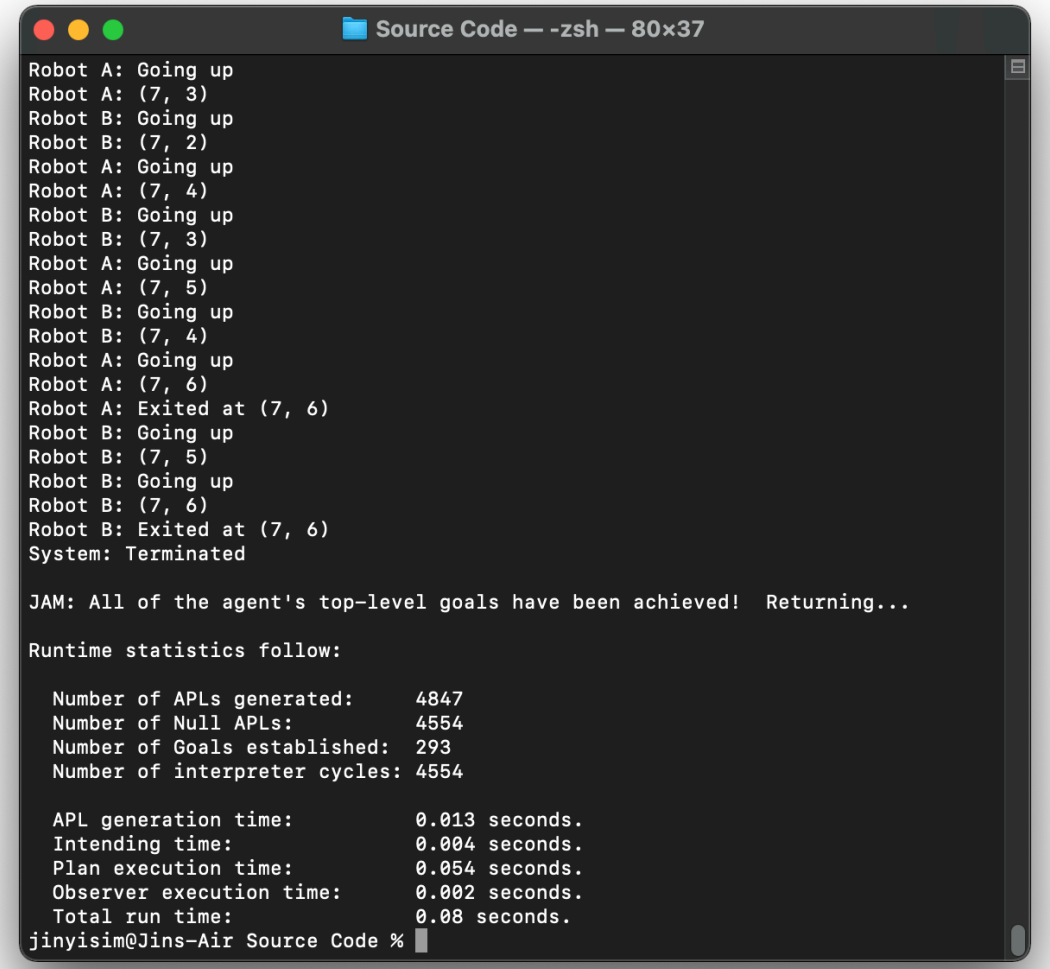

Figure 3.14: Test case 3 output (4)

3.3 Test Cases (Location of the Obstacles)

3.3.1 Test Case 1

3.3.1.1 Initial Setting

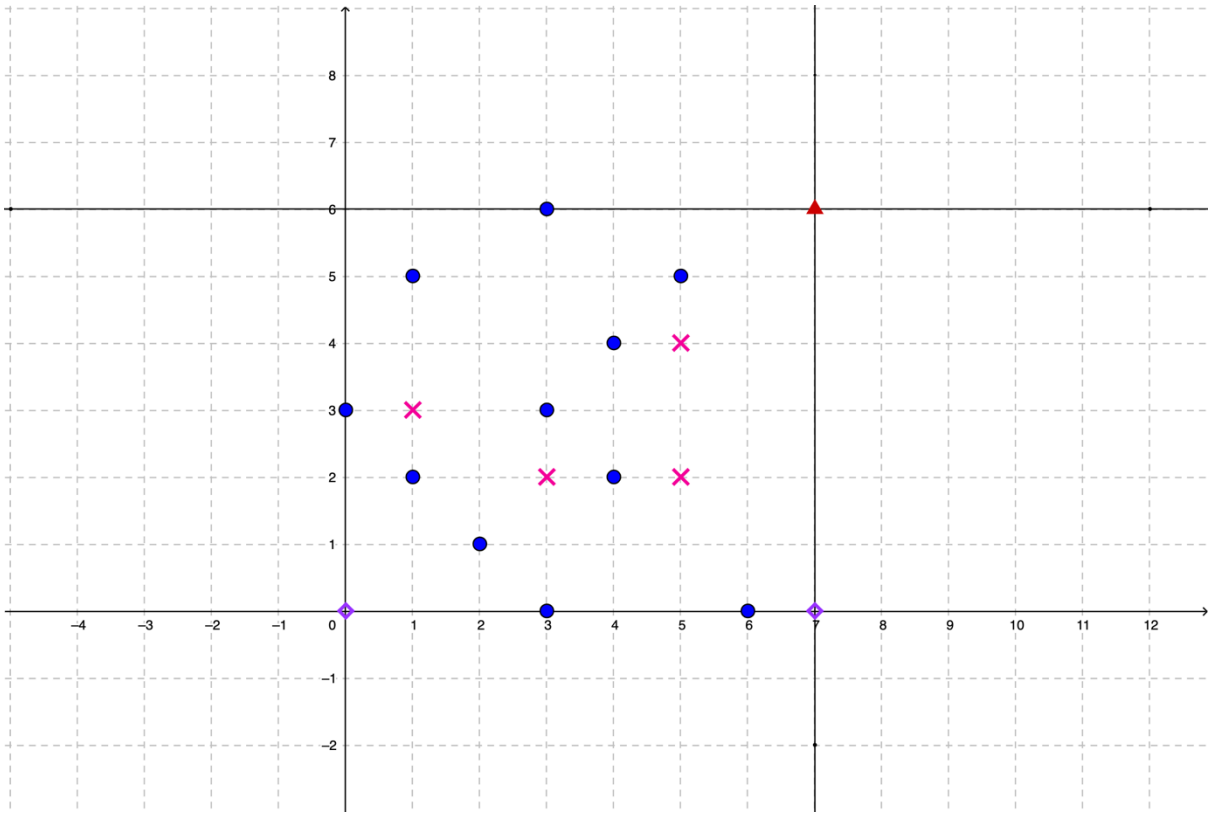

Figure 3.15: Initial placement of the dusts, obstacles, robots and exit

From the initial placement, the location of the dusts were those given in the question, which was also used in section 3.2.1. However, obstacles were added which were marked in red crosses in which the robots shall not pass. In this test, it was expected that the robots will still collect all the dusts and exit from the grid within the time limit since none of the dusts was blocked completely by the obstacles. Nonetheless, the robots will surely encounter a dead end in this case and a random move must be made to reach some of the blocked locations.

3.3.1.2 Program Output

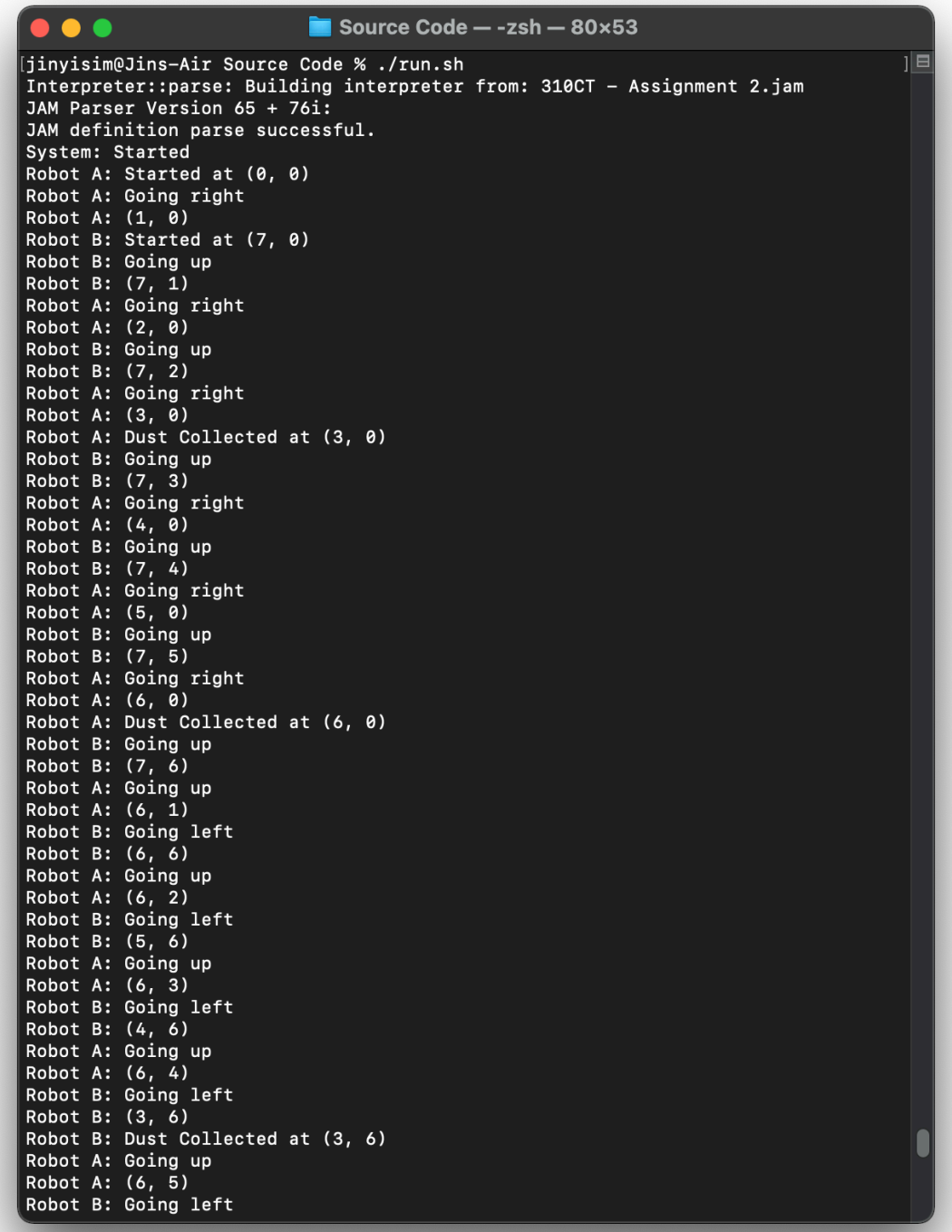

Figure 3.16: Test case 1 output (1)

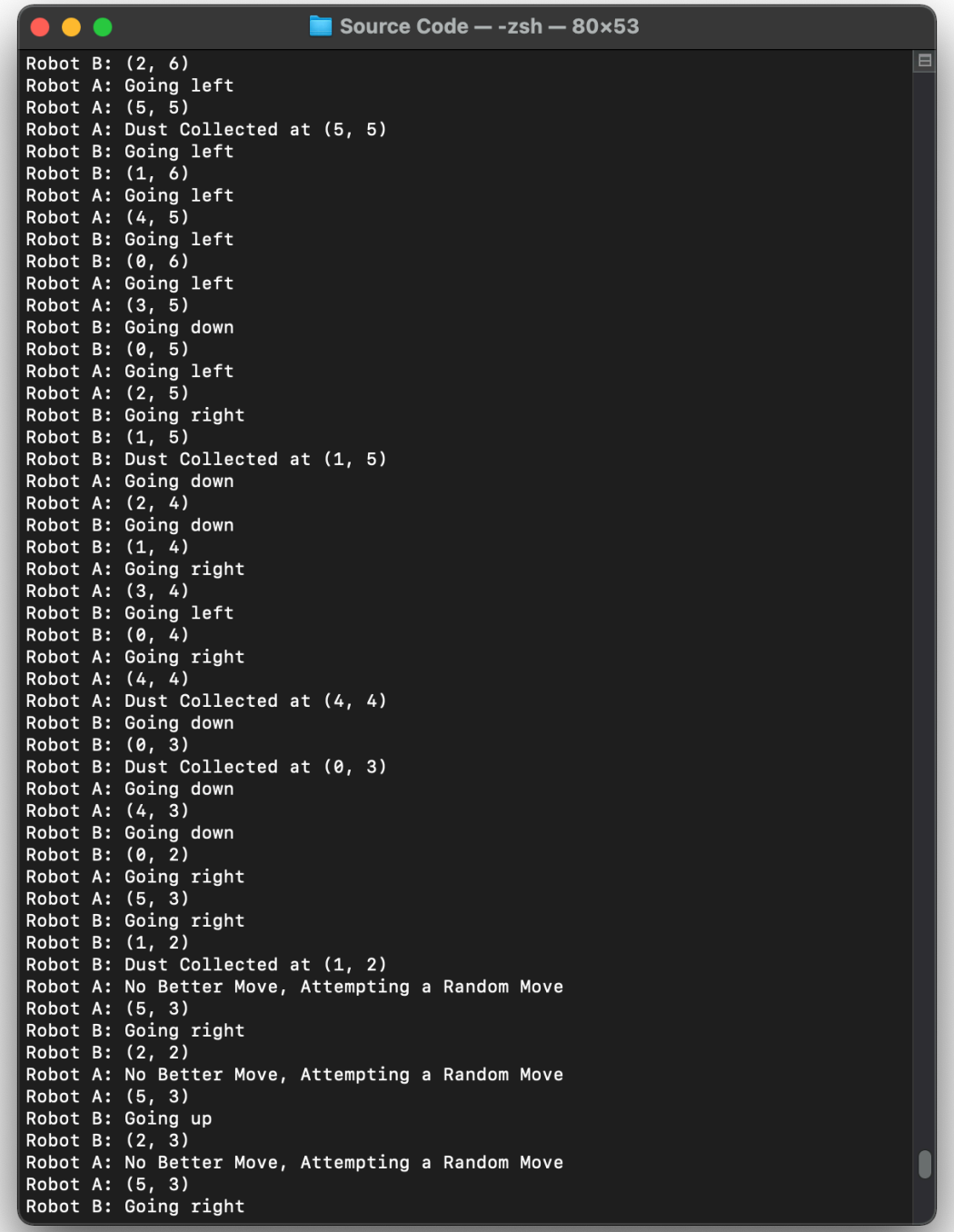

Figure 3.17: Test case 1 output (2)

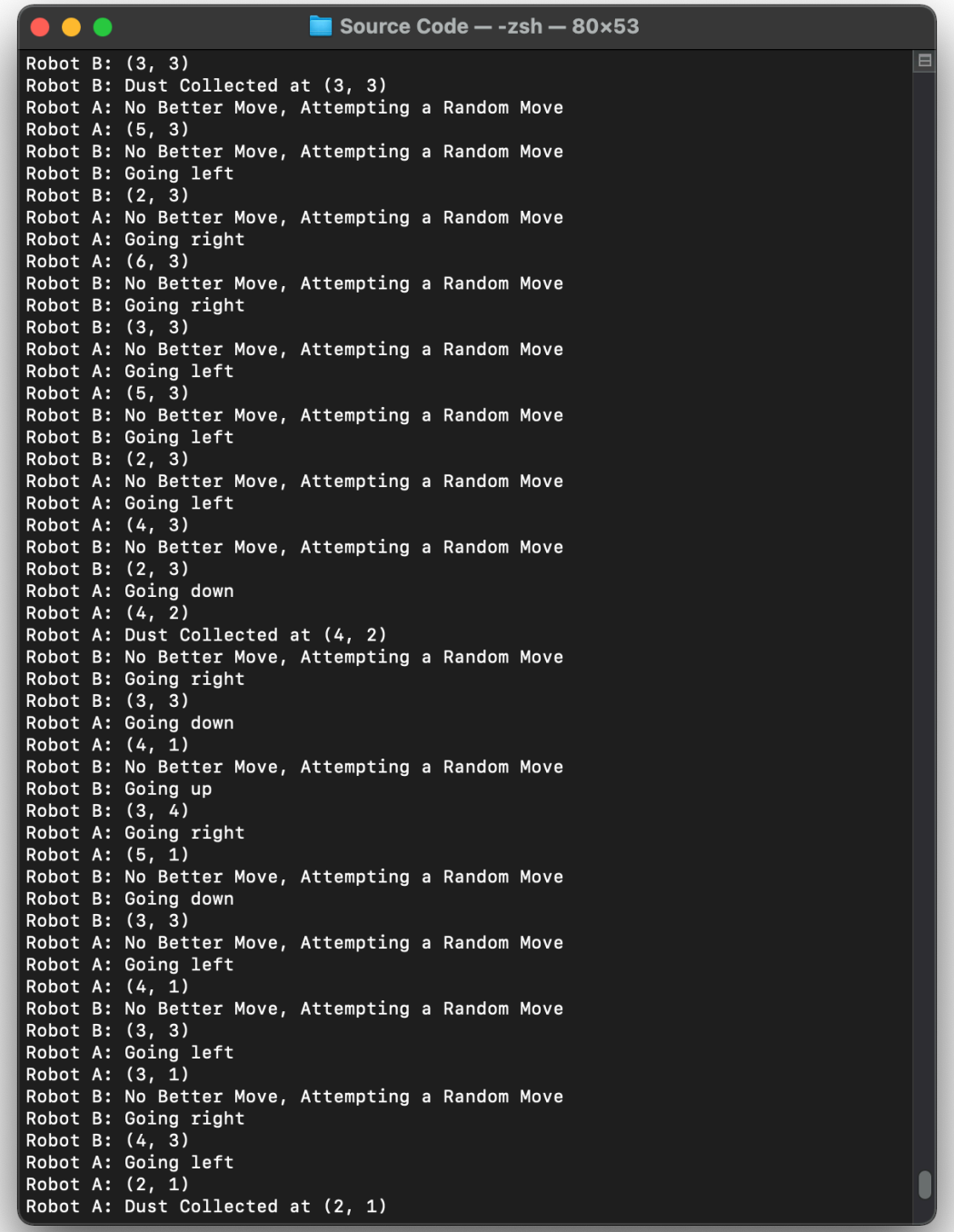

Figure 3.18: Test case 1 output (3)

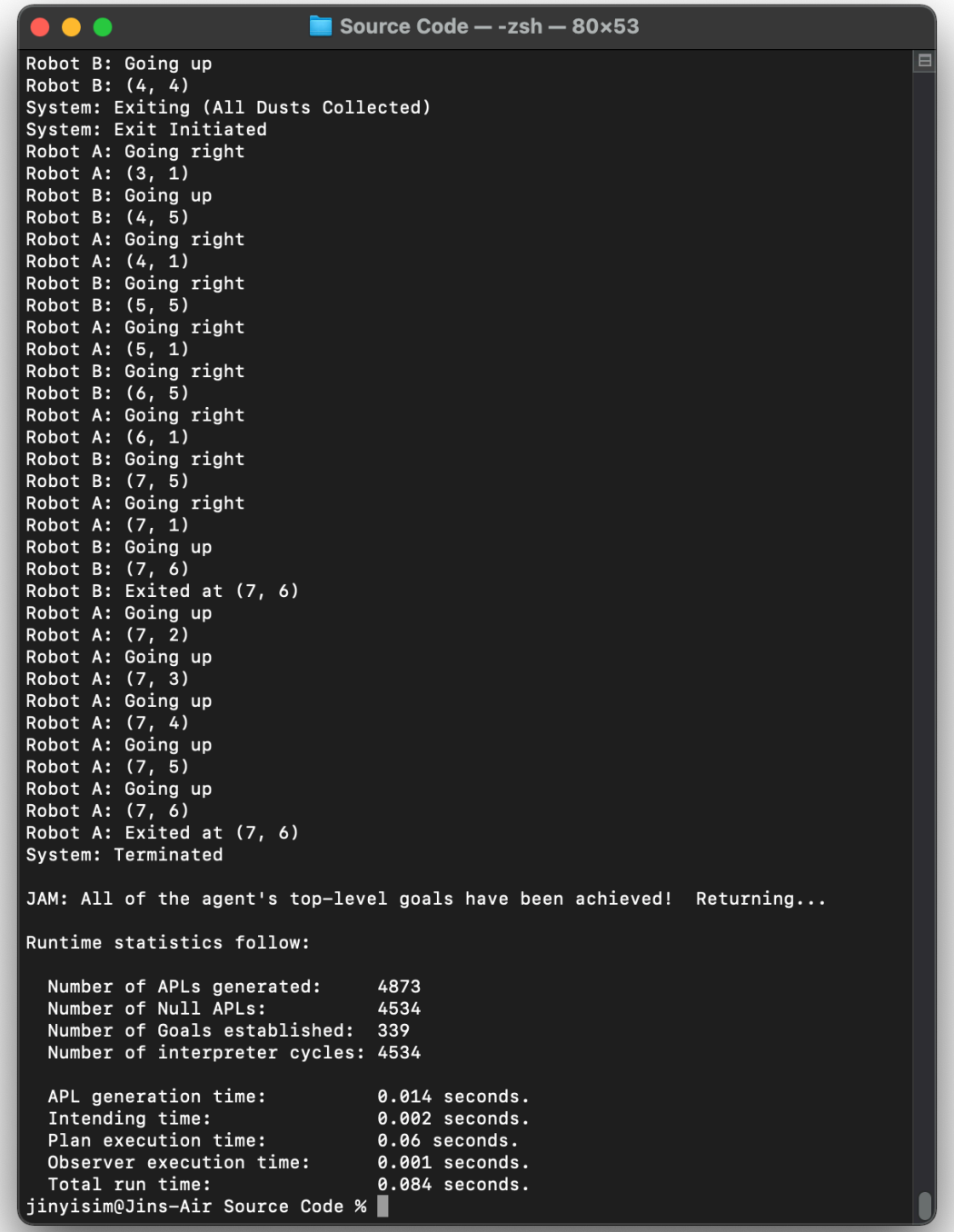

Figure 3.19: Test case 1 output (4)

3.3.1.3 Explanation

From the screenshots of the output above, it was clear that the robots did not run into the cell with obstacle (a search was performed in the terminal to ensure that the coordinate of the obstacles were not found). With the obstacles in place, the first deterministic movement of the robots missed a number of cells in the grid, which results in a number of missed dusts. Referring to Figure 3.18, both robots attempt to move randomly since no better move was determined to search for the remaining dusts. As shown in the last statement, after moving randomly for a period, robot A finally found and collected the last dust in the grid.

With all the dusts collected, the system printed that all the robots are exiting because all the dusts were collected in the beginning of Figure 3.19. With that, the robots switched their goal to return to the exit. This was shown in the output following the exit statement, whereby both robots attempt to move towards the exit, with robot B reaching (7, 6) which was where the exit was located first, followed by five more moves from robot A before reaching the exit.

In a nutshell, this shows that with the obstacles in place, the robots will still attempt to deterministically scan through the grid for the first time. If all the dusts were found at the first scan, the robots will exit immediately. Otherwise, the robots will attempt to find the remaining dusts within the grid while avoiding the obstacles in a random fashion should there be no better move to make. The output above shows that the heuristics were successful in allowing the robots to navigate effectively and collect all the dusts within the time limit calculated initially.

3.3.2 Test Case 2

3.3.2.1 Initial Setting

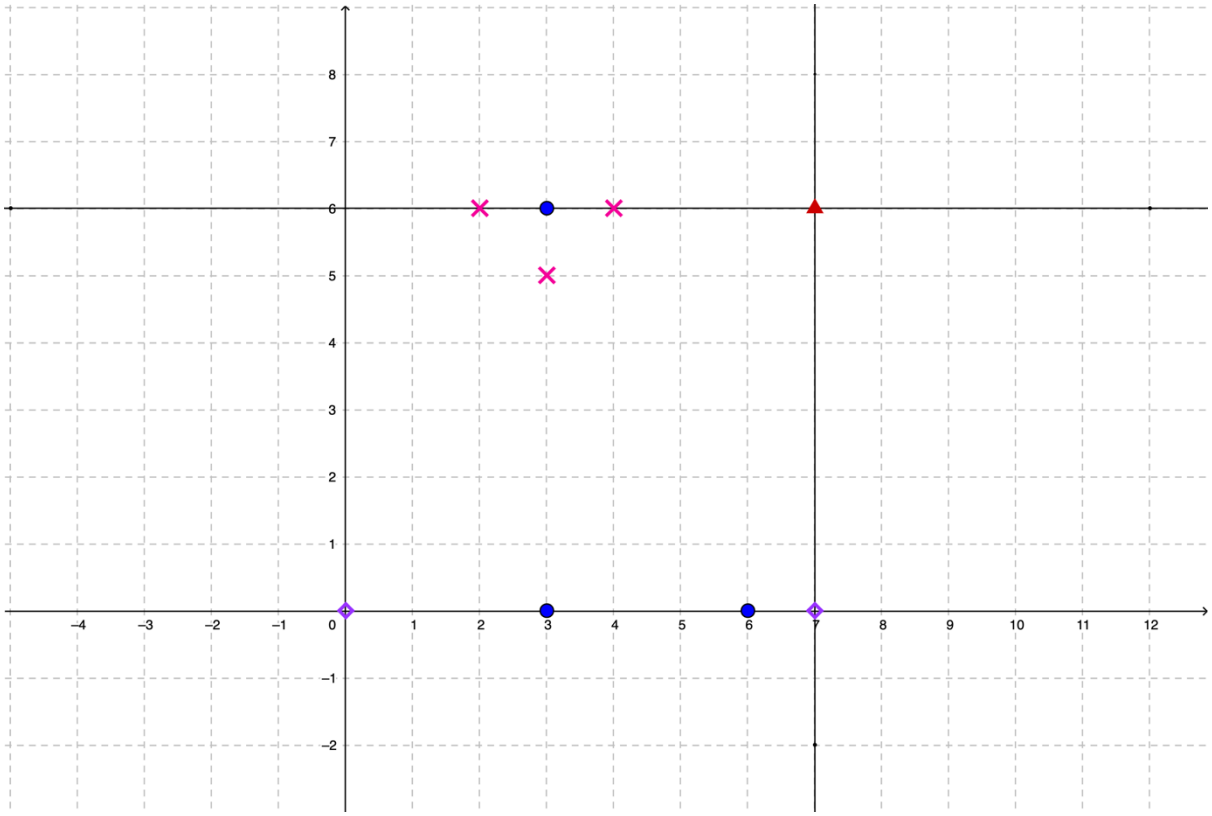

Figure 3.20: Initial placement of the dusts, obstacles, robots and exit

The setup shown in Figure 3.20 will return in an infinte loop. This is because the obstacles have blocked the dust at (3, 6), which means that the robots will never have the chance to reach the location to clean up the dust. In this case, the robots will attempt to search until the time limit eventually expires, in which the robots will then give up and navigate to the exit.

The setup above was done deliberately in showing that the program handles situations when the dusts were out of reach of the robots. This also mimics the real-world whereby the doors for some of the rooms may be closed, and the robots will have to keep trying to search for an entry to the room for cleaning until the battery becomes low and the robots will be forced to navigate back to the base station to recharge. The output below will only include the screenshots when the program starts (with the dusts at (3, 0) and (6, 0) collected) and the section where the robots keep trying until they eventually give up as the output is very lengthy.

3.3.2.2 Program Output (Excerpt)

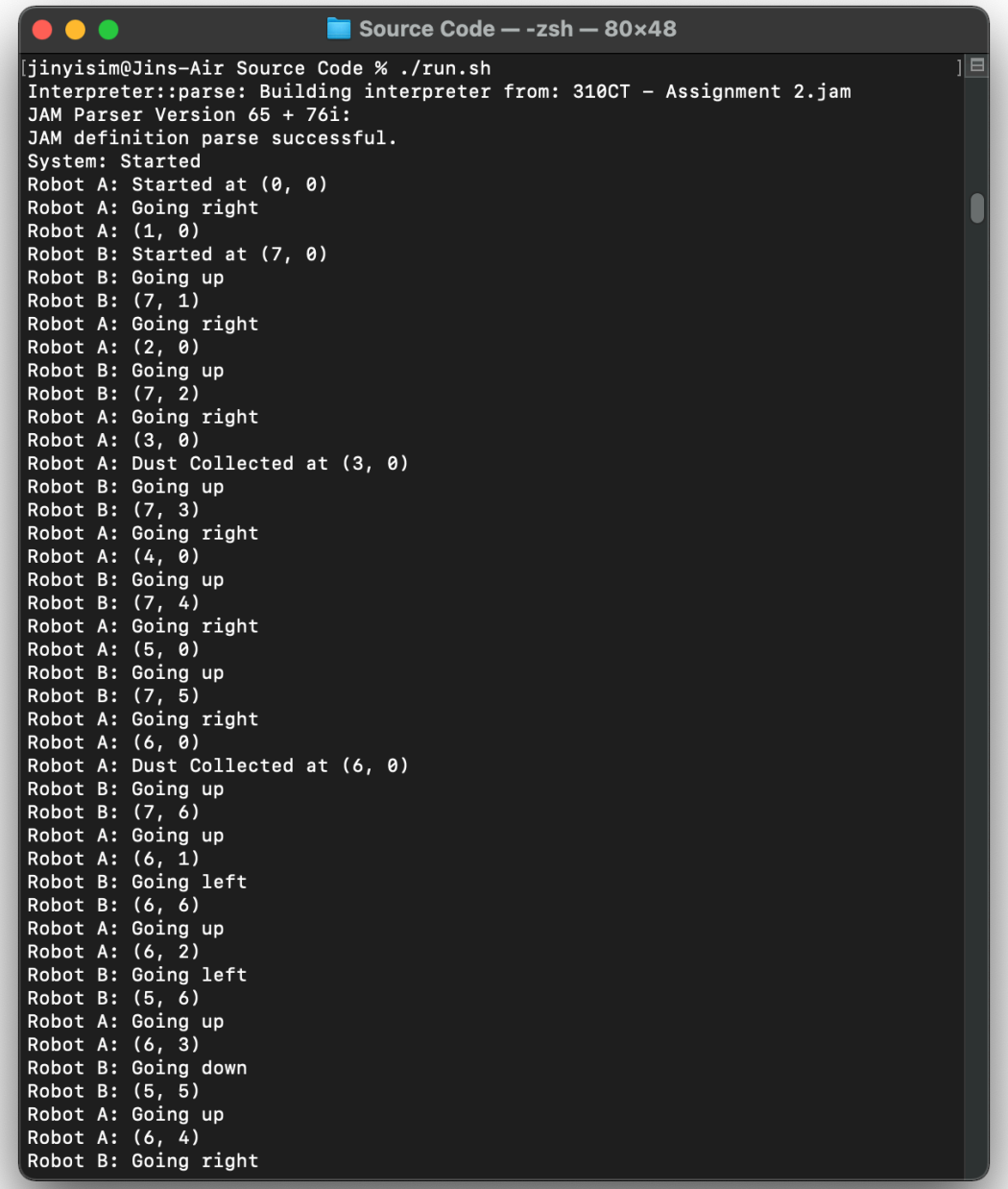

Figure 3.21: Test case 2 output (1), dusts at (3, 0) and (6, 0) collected

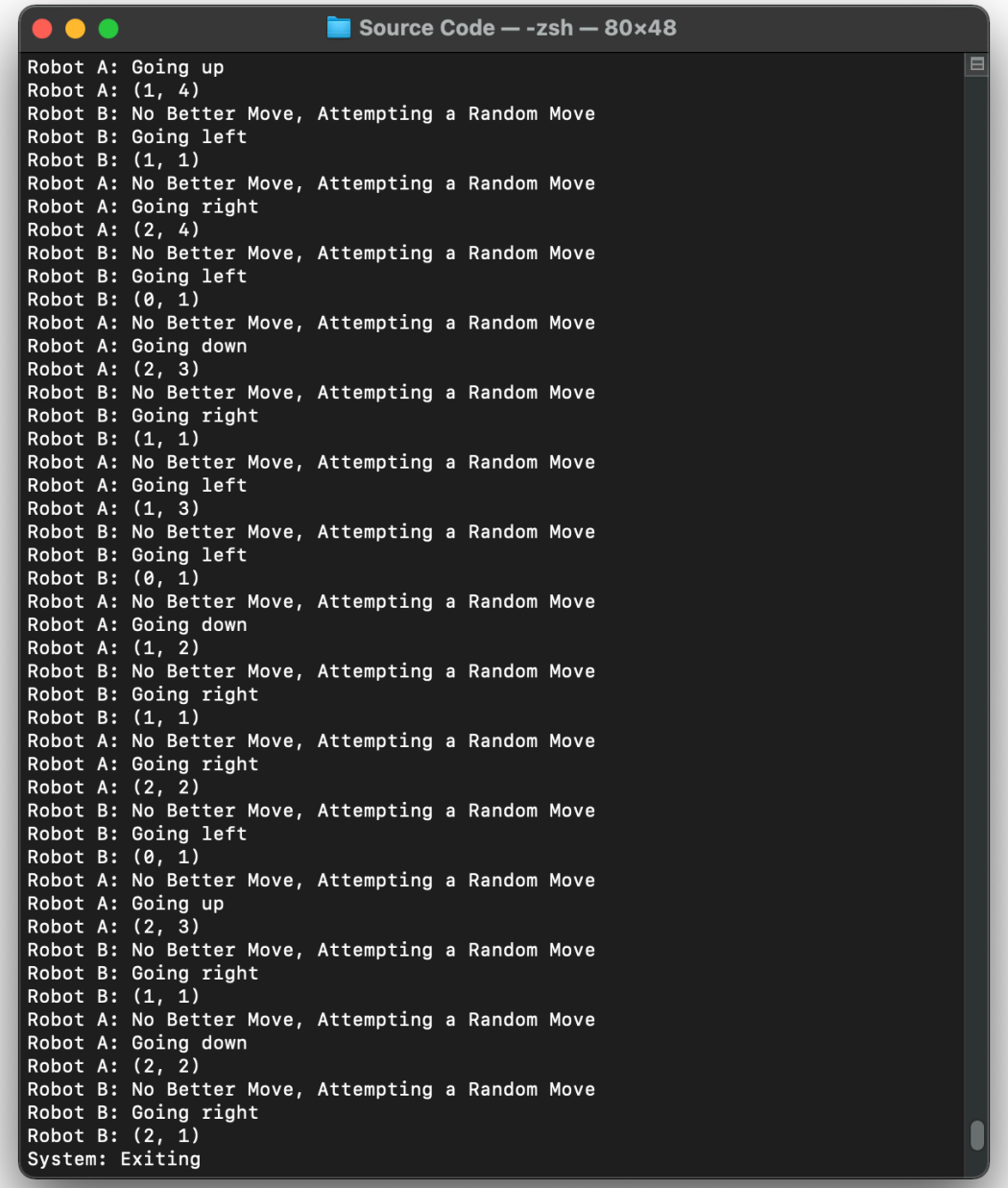

Figure 3.22: Test case 2 output (2), retrying until the time limit expires and exits without collecting all the dusts in the grid

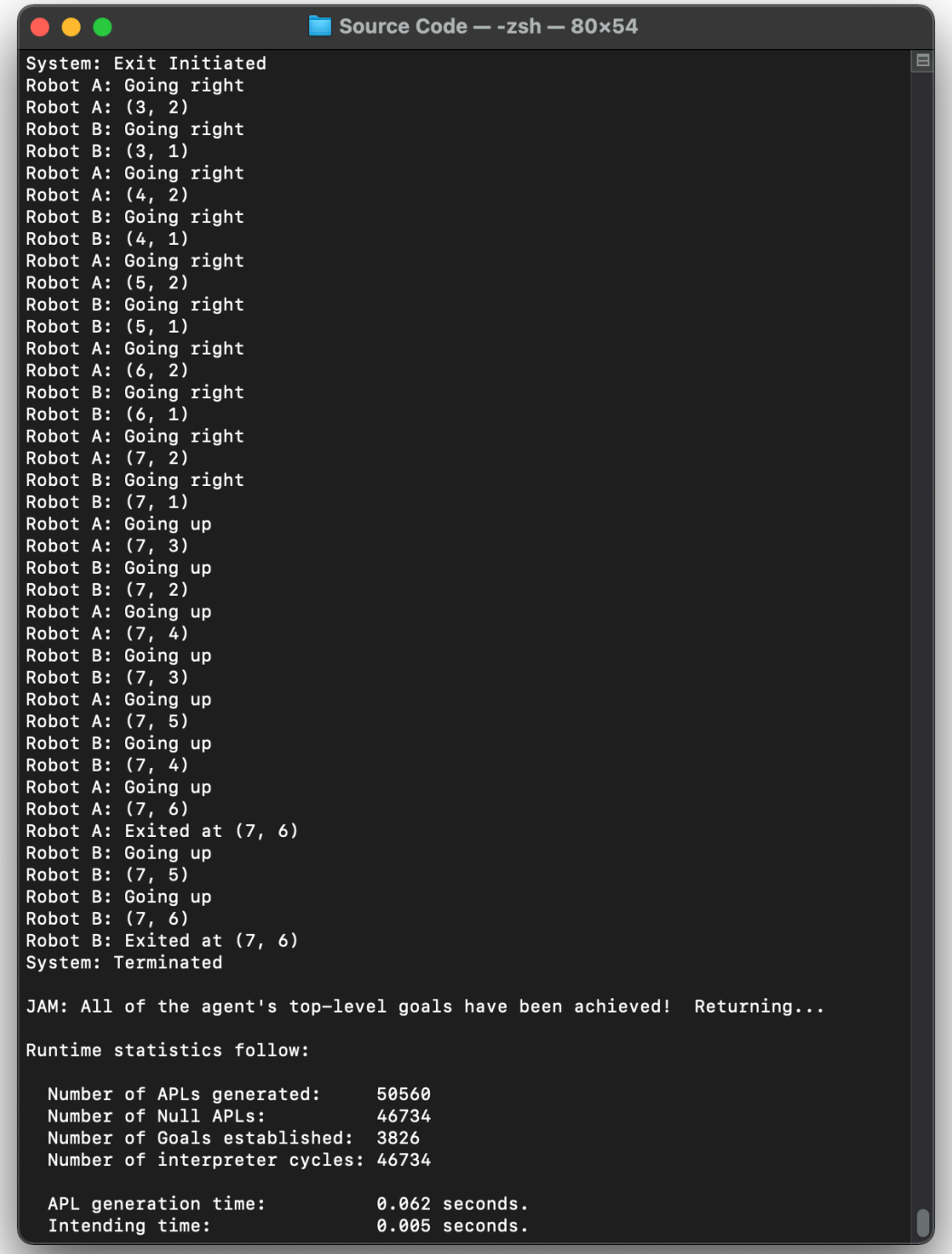

Figure 3.23: Test case 2 output (3), both robots navigated to the exit at (7, 6)

3.3.3 Test Case 3

3.3.3.1 Initial Setting

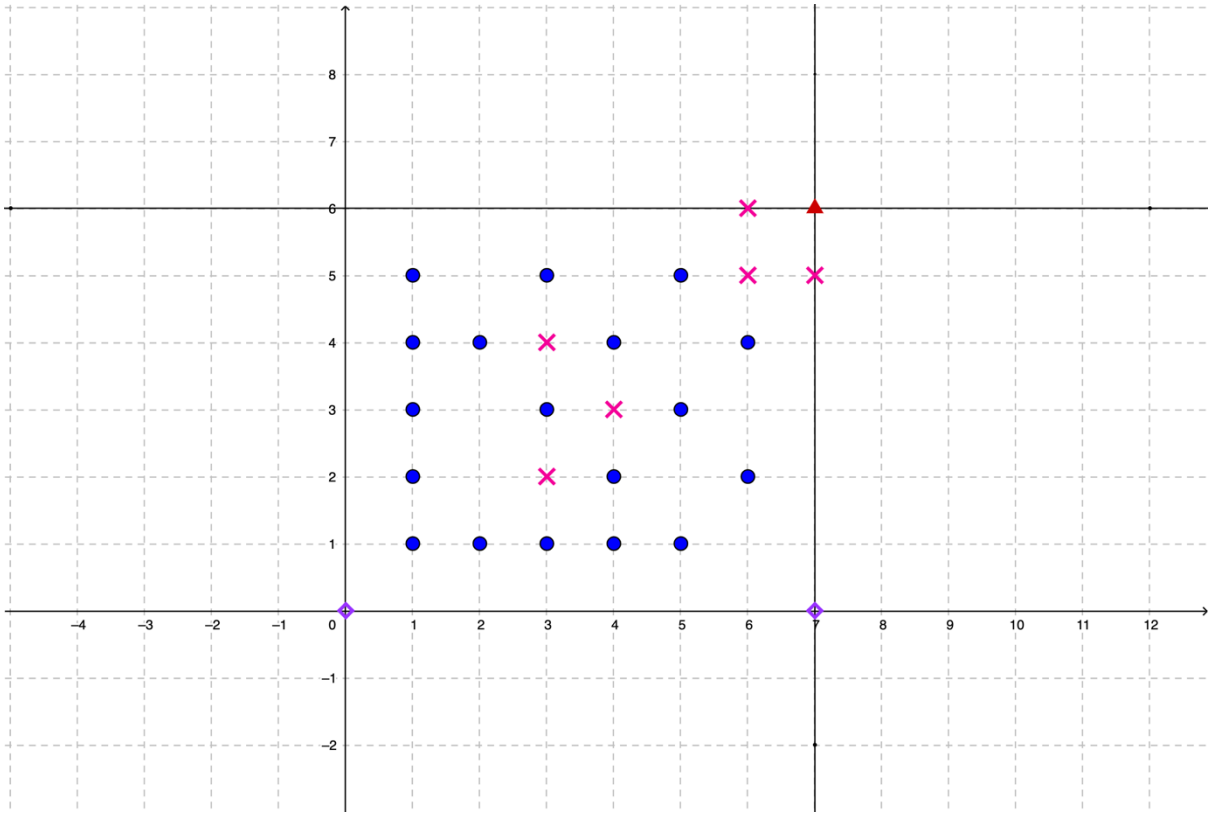

Figure 3.24: Initial placement of the dusts, obstacles, robots and exit

The placement of the obstacles above will result in a longer period of time required to clean up the dusts in the grid since the dust at (3, 3) can only be accessed from a single direction. On the other hand, the exit is inaccessible as it was entirely blocked by the obstacles. The placement was deliberately done to showcase the plan to achieve the goal to return to the exit handling the scenario where the exit cannot be reached. The robots will keep trying to search for the exit until the time limit runs out, where the robots run out of battery in real-world.

Similarly, only the screenshots showing all the dusts were collected successfully and when the robots fail to exit and the program finally terminates will be included in the report since the process of the robots retrying in searching for the exit is too lengthy to be included.

3.3.3.2 Program Output (Excerpt)

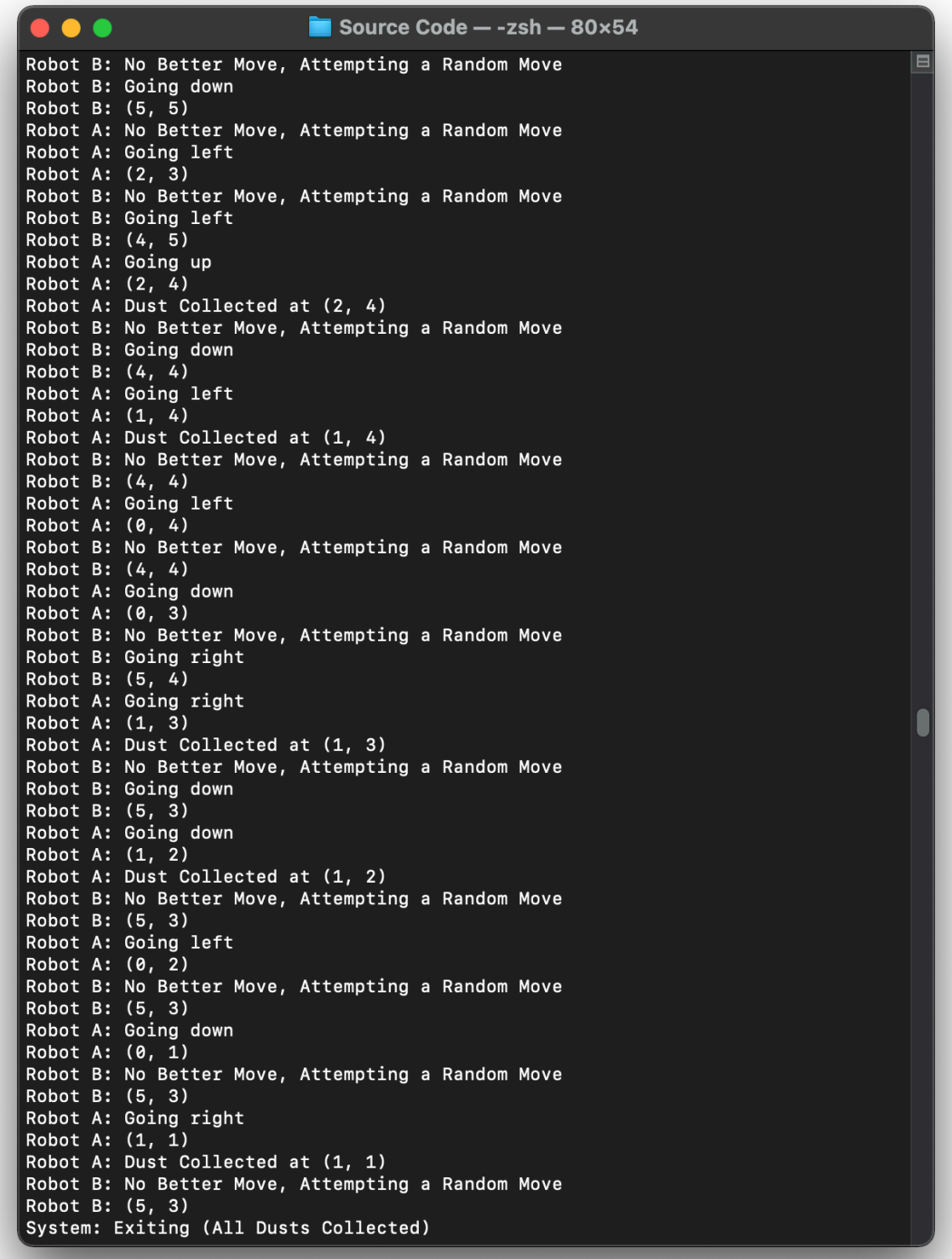

Figure 3.25: Test case 3 output (1), all the dusts were collected

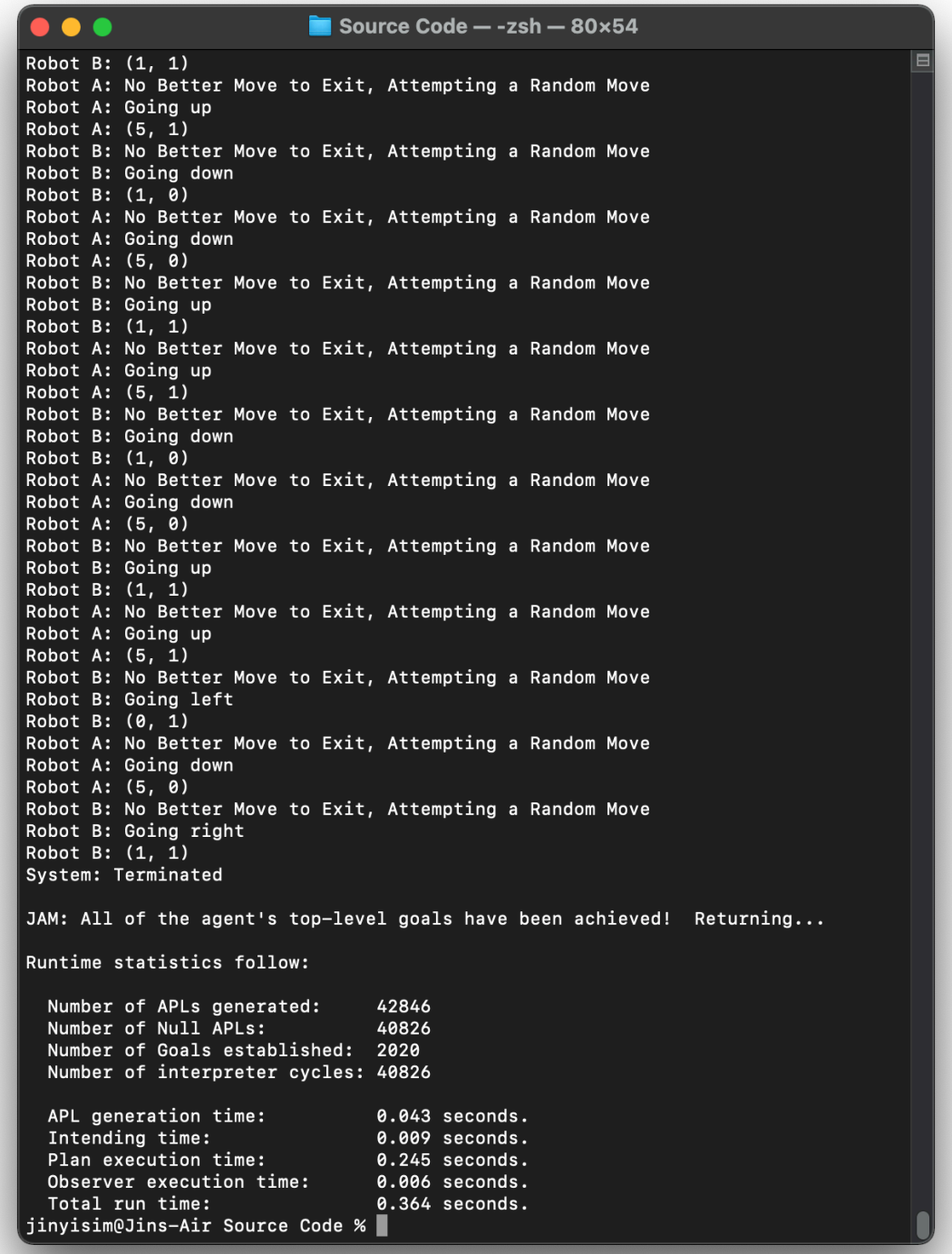

Figure 3.26: Test case 3 output (2), the program terminates without both robots reaching the exit (blocked entirely by obstacles)

3.4 Test Cases (Initial Location of the Robots)

3.4.1 Test Case 1

3.4.1.1 Initial Setting

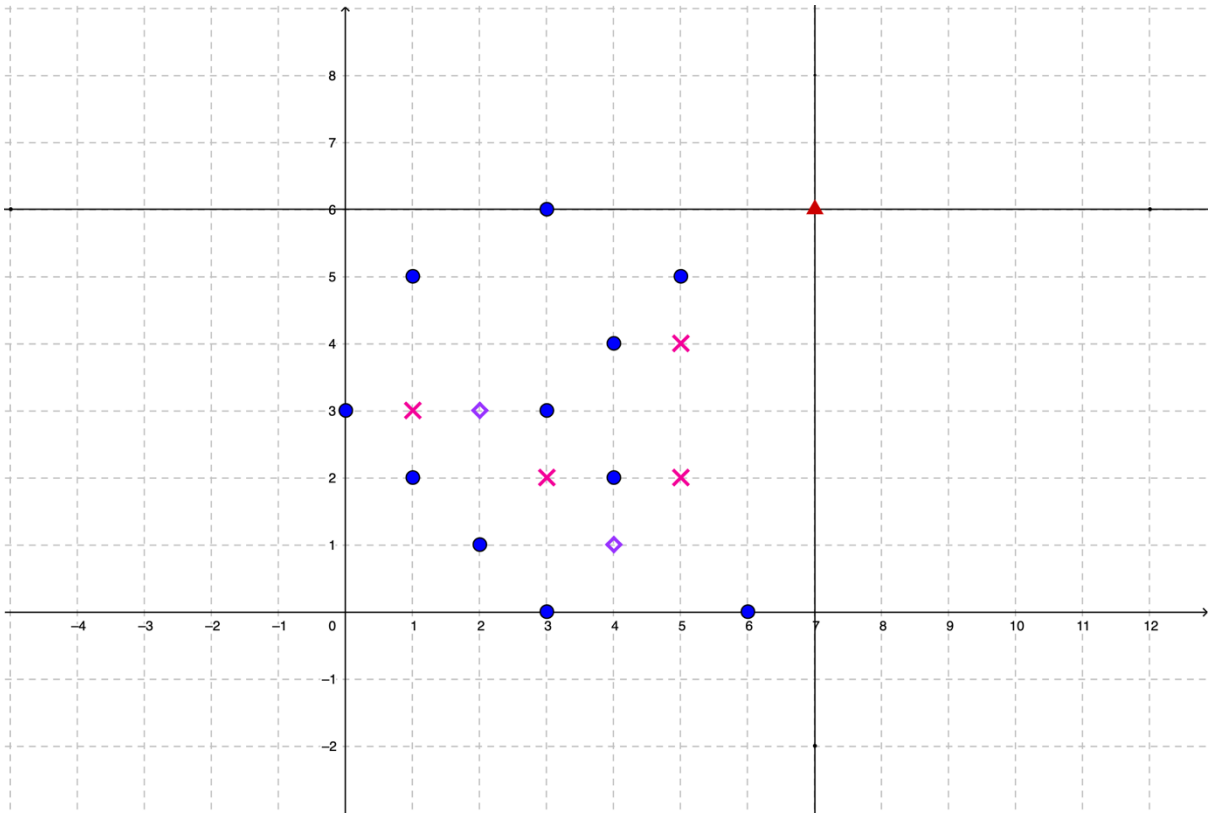

Figure 3.27: Initial placement of the dusts, obstacles, robots and exit

The test in this section will showcase the program in handling the scenarios when the robots were placed at different location initially. Figure 3.27 shows the location of the dusts and obstacles which were the same as the test performed in section 3.3.1. However, the location of robot A and B were shifted to (2, 3) and (4, 1) respectively. As the algorithm prioritizes the movement towards the right and top, the initial deterministic movement will be affected, whereby the robots will run into a situation where there is no better move to make quickly.

However, the element of randomness kicks in to help the robots to navigate towards an unvisited grid cell, which will eventually lead the robots into discovering more adjacent unvisited grids. Hence, the overall program should not be affected by the initial location of the robots since the algorithm allows the robots to move randomly when no better move can be made. The screenshots for the output in the following section will include the instances when the robots begin initially, collecting all the dusts and exits from the grid successfully.

3.4.1.2 Program Output (Excerpt)

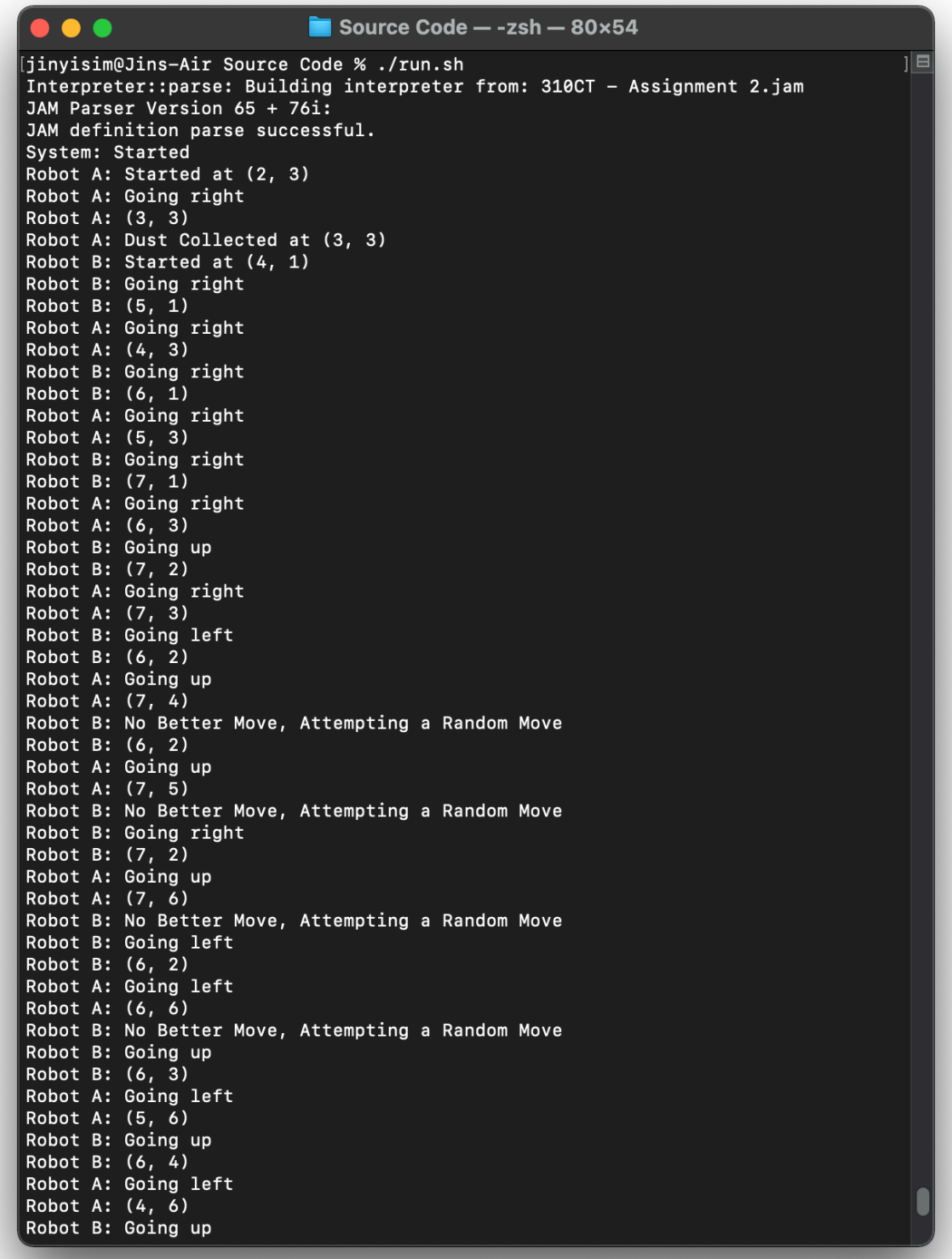

Figure 3.28: Test case 1 output (1), robot A begins at (2, 3), robot B begins at (4, 1)

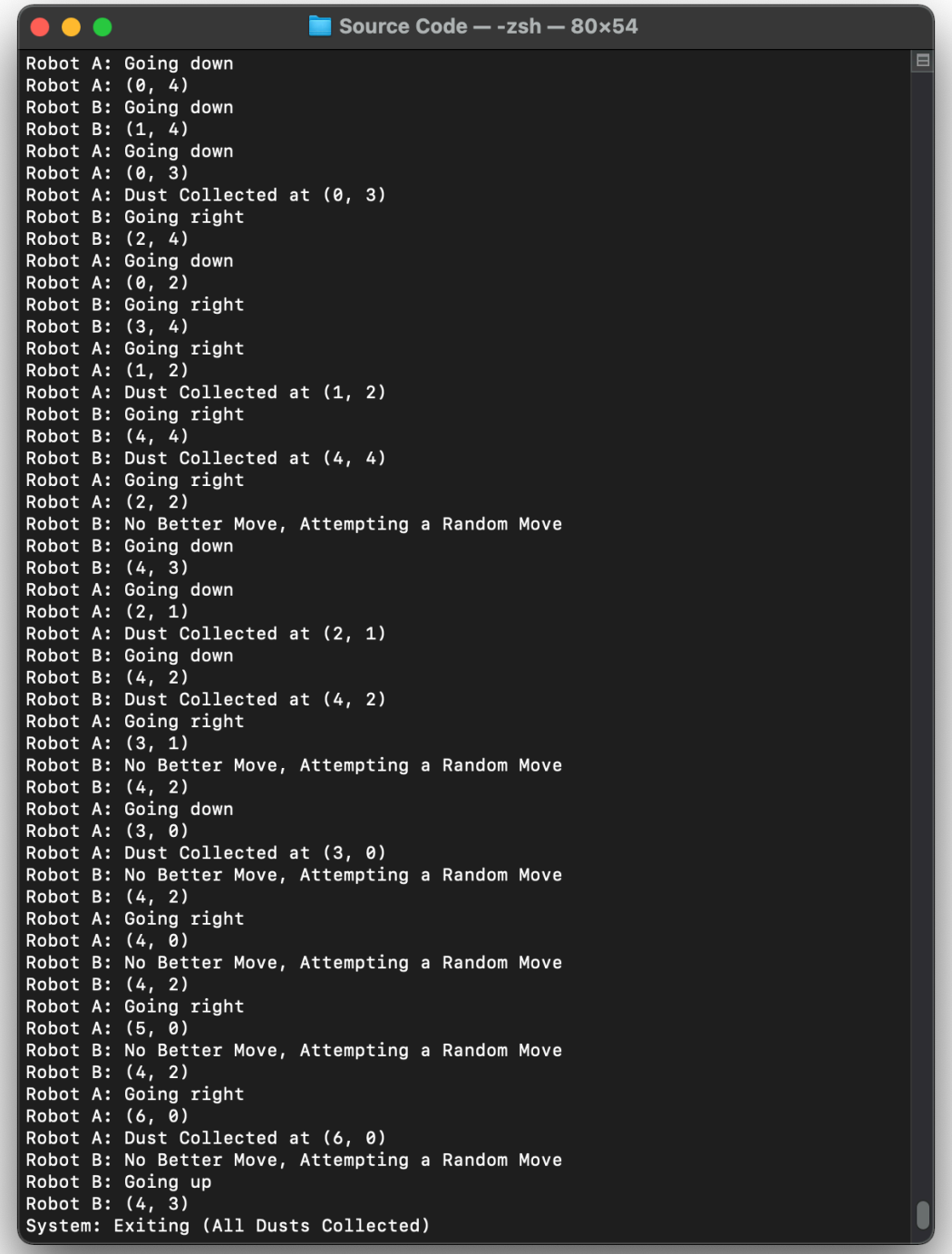

Figure 3.29: Test case 1 output (2), all the dusts were cleaned

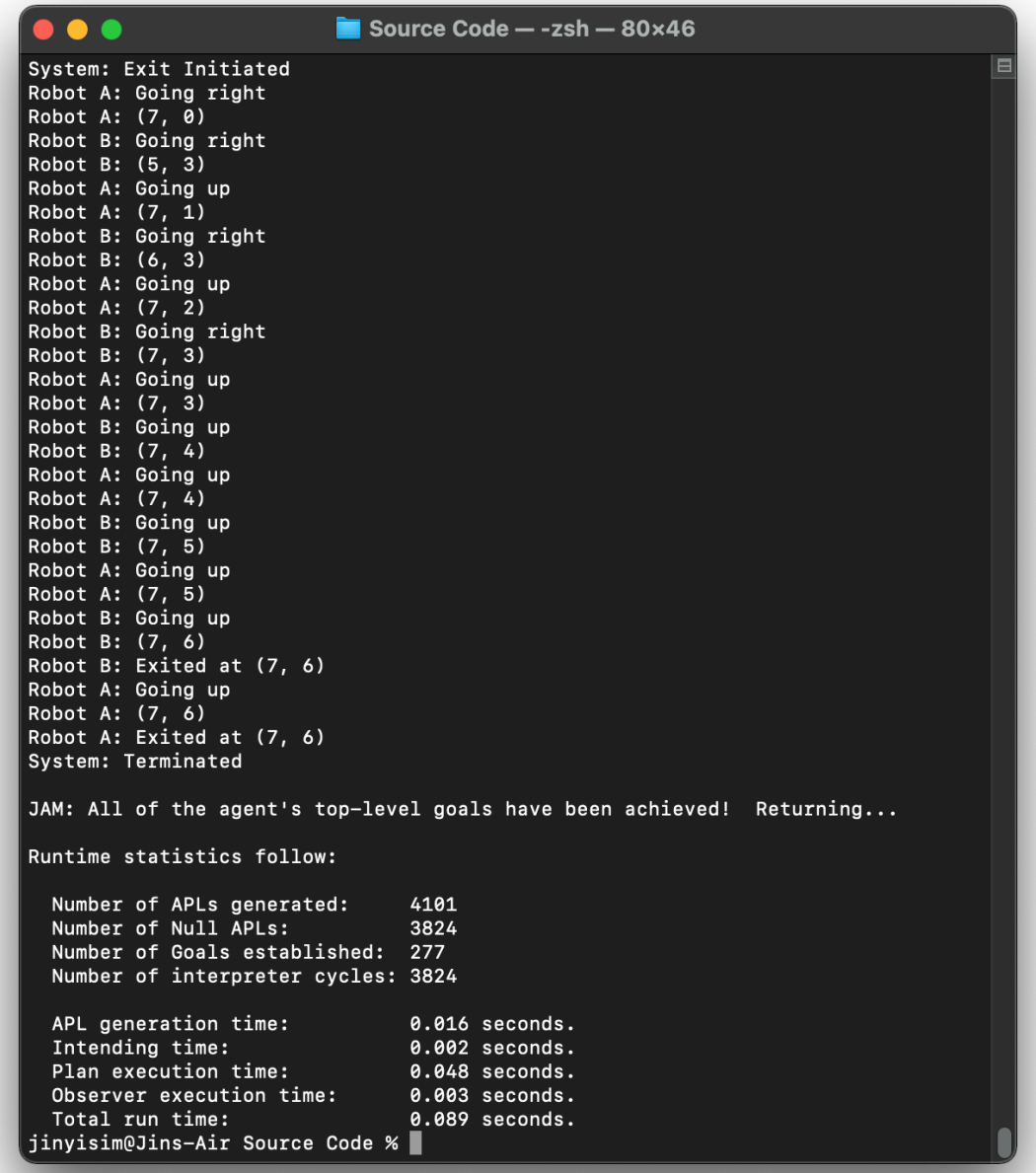

Figure 3.30: Test case 1 output (3), both robots exited successfully

3.4.1.3 Explanation

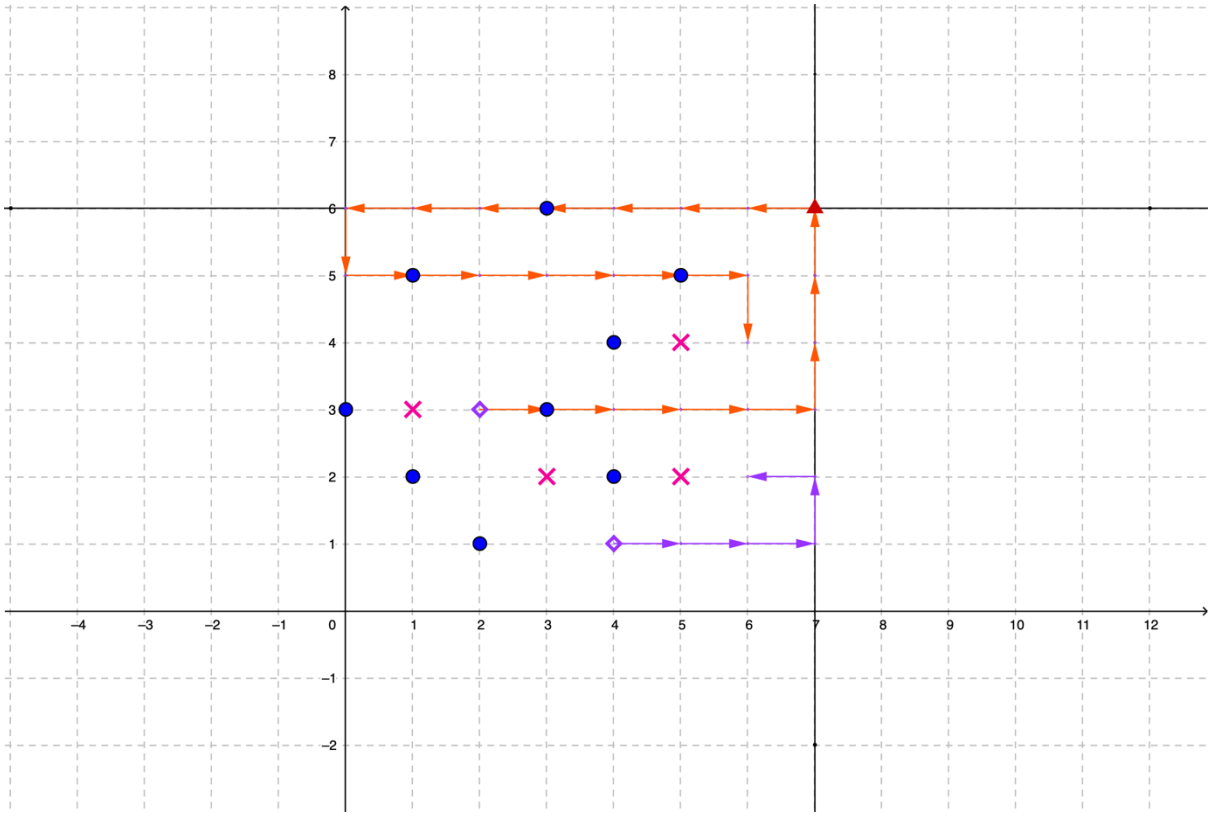

Figure 3.31: Deterministic steps taken by robot A and B

Figure 3.31 shows the deterministic steps made by the robot A and B drawn in orange and purple arrows respectively. This revealed that when the robots were placed higher in the y-axis, the robots tend to miss out a lot of cells when the robots first attempt to scan through all the cells in the grid. However, the ability for the robots to move to a random location allows the robots to eventually progress to the unvisited cells. As the unvisited cells are clustered together, the robots will start to scan through all the unvisited cells until all the cells were visited. This process repeats until all the cells in the grid were visited, hence even with the robots placed randomly in the grid, all the cells that are not blocked will eventually be visited.

3.4.2 Test Case 2

3.4.2.1 Initial Setting

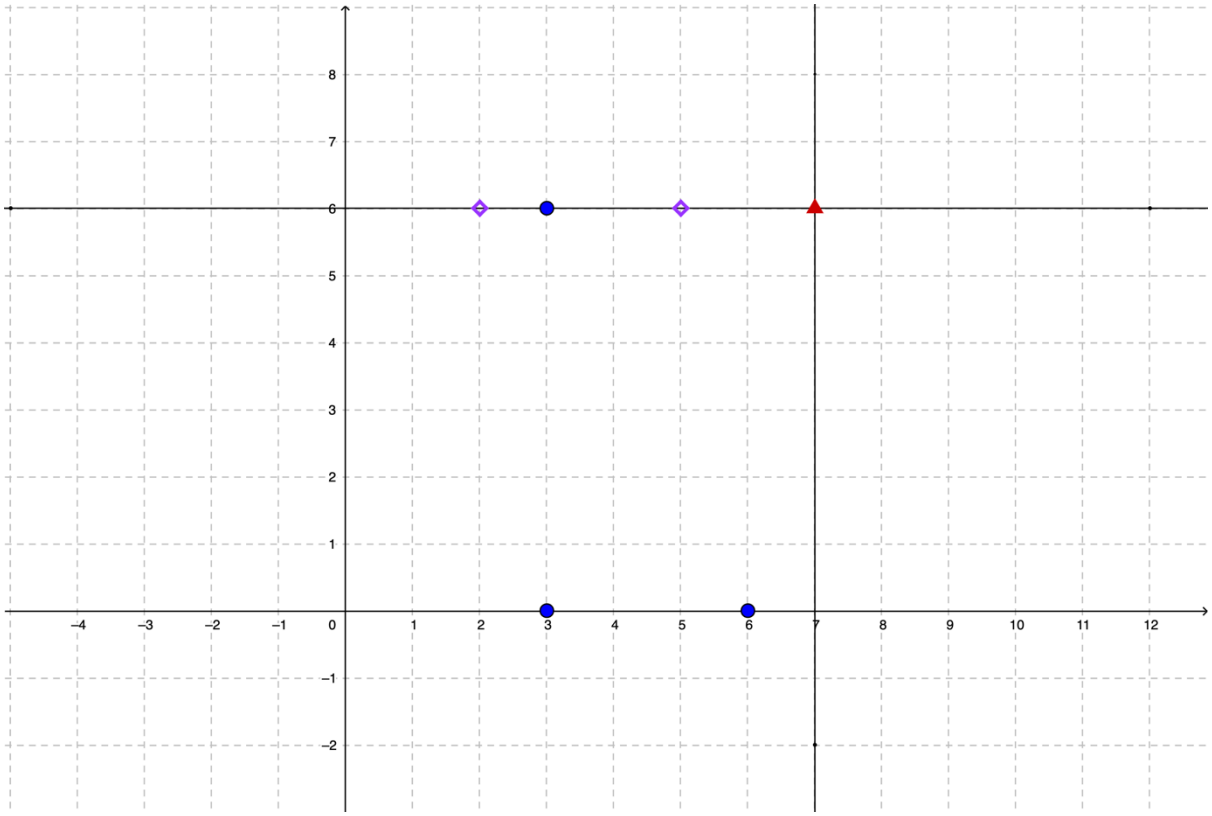

Figure 3.32: Initial placement of the dusts, robots and exit

To showcase the robots being able to complete the scan over the entire grid even when placed at the top of the grid, whereby robot A was placed at (2, 6) while robot B was placed at (5, 6). The placements for the dusts, robots and exit were plotted in the cartesian plane above. The output below will include the screenshots when the robot first starts, when the robot finished collecting all the dusts followed by the screenshot of the robots exit successfully.

3.4.2.2 Program Output (Excerpt)

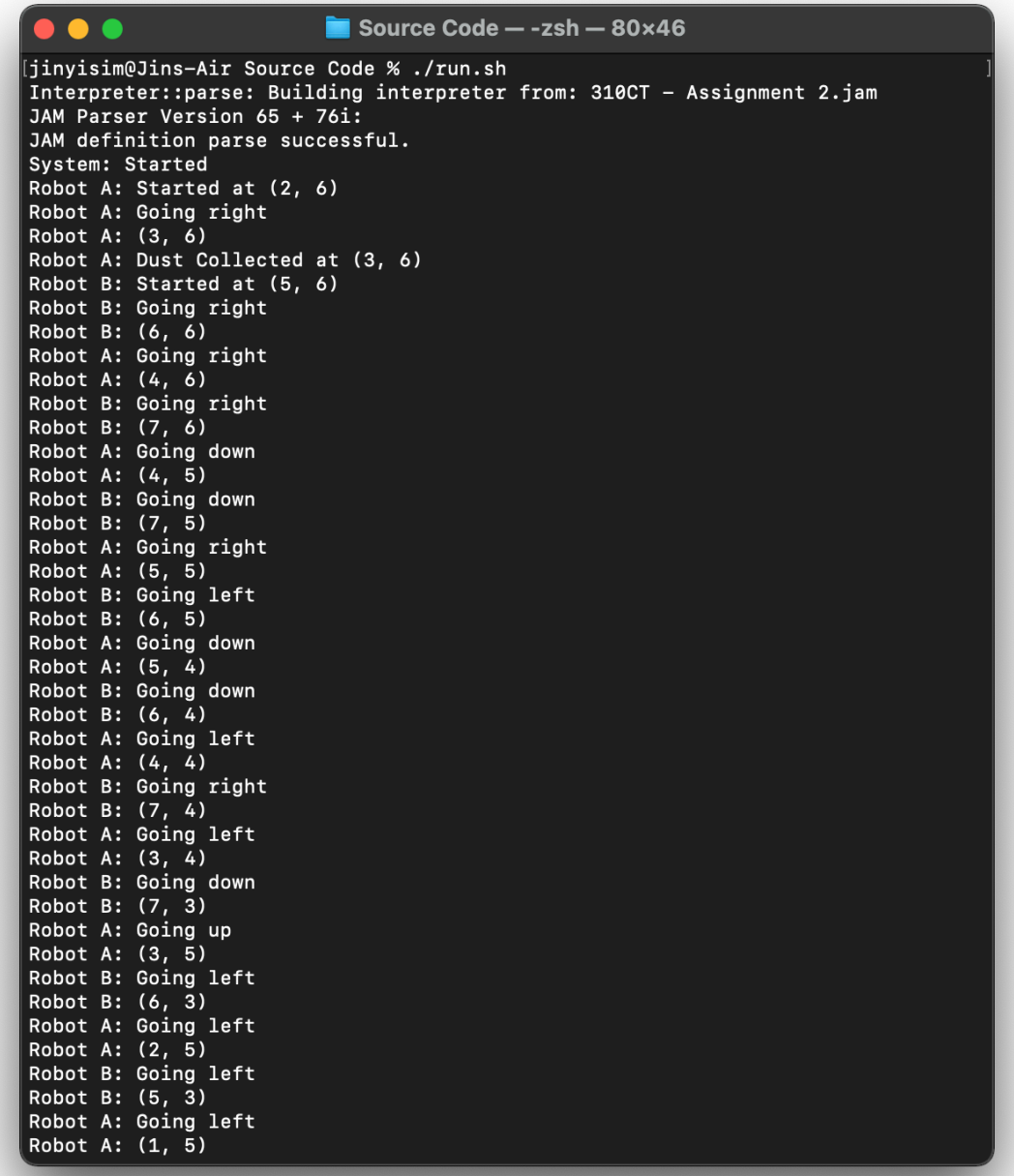

Figure 3.33: Test case 2 output (1), robot A begins at (2, 6), robot B begins at (5, 6)

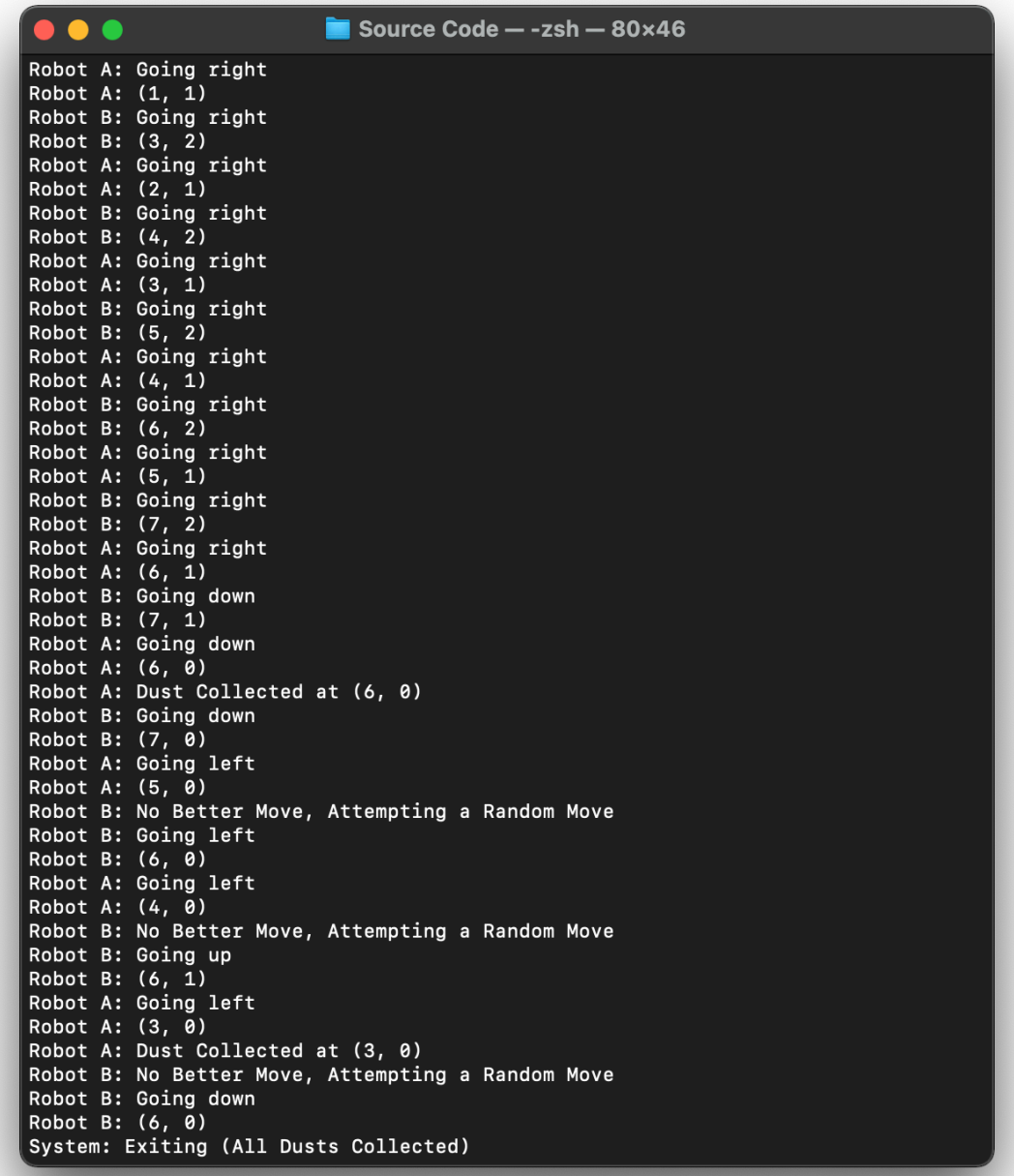

Figure 3.34: Test case 2 output (1), all the dusts were collected

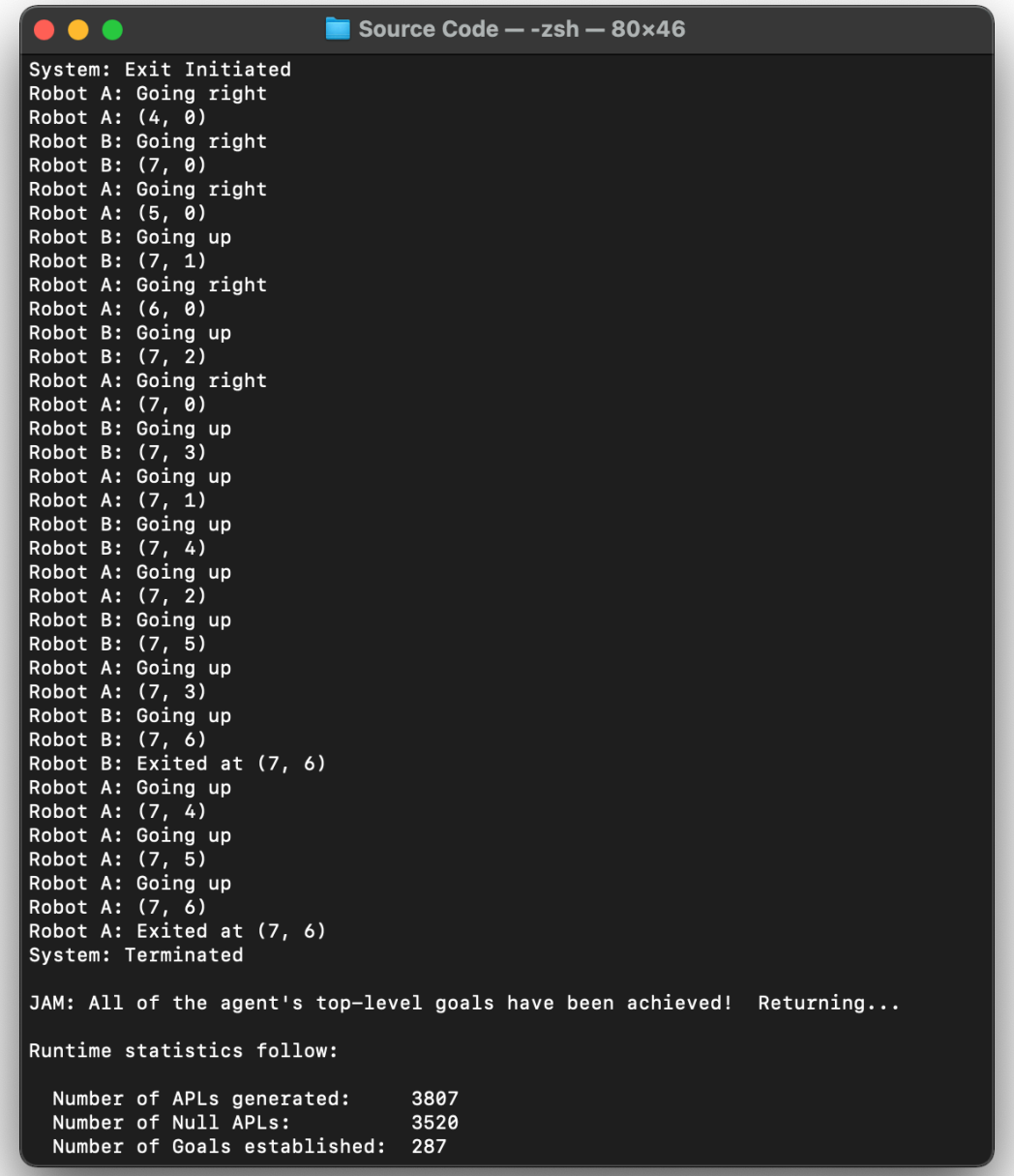

Figure 3.35: Test case 2 output (3), both robots exited from the grid at (7, 6)

3.4.3 Test Case 3

3.4.3.1 Initial Setting

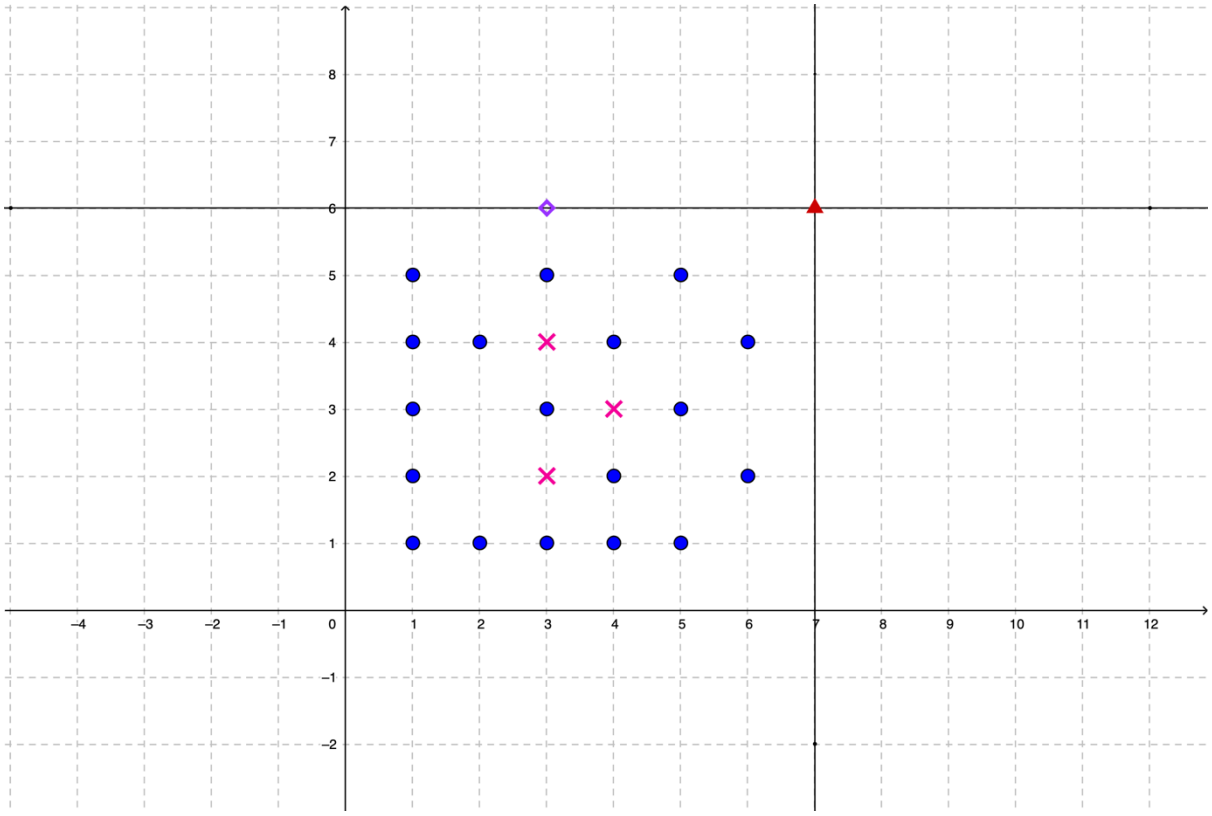

Figure 3.36: Initial placement of the dusts, obstacles, robots and exit

To test the behaviour of the robots when both robot A and B were placed together, the placement of the dusts, obstacles, robots and exit were used in running the program. In the setting above, both robot A and B were placed on (3, 6) overlapping each other to ensure that the program behaves properly even when both are initiated at the same location. The output below includes the excerpt of the output which shows the starting location, the instances when the robots clean up all the dusts in the grid and proceed to the exit successfully by the end.

3.4.3.2 Program Output (Excerpt)

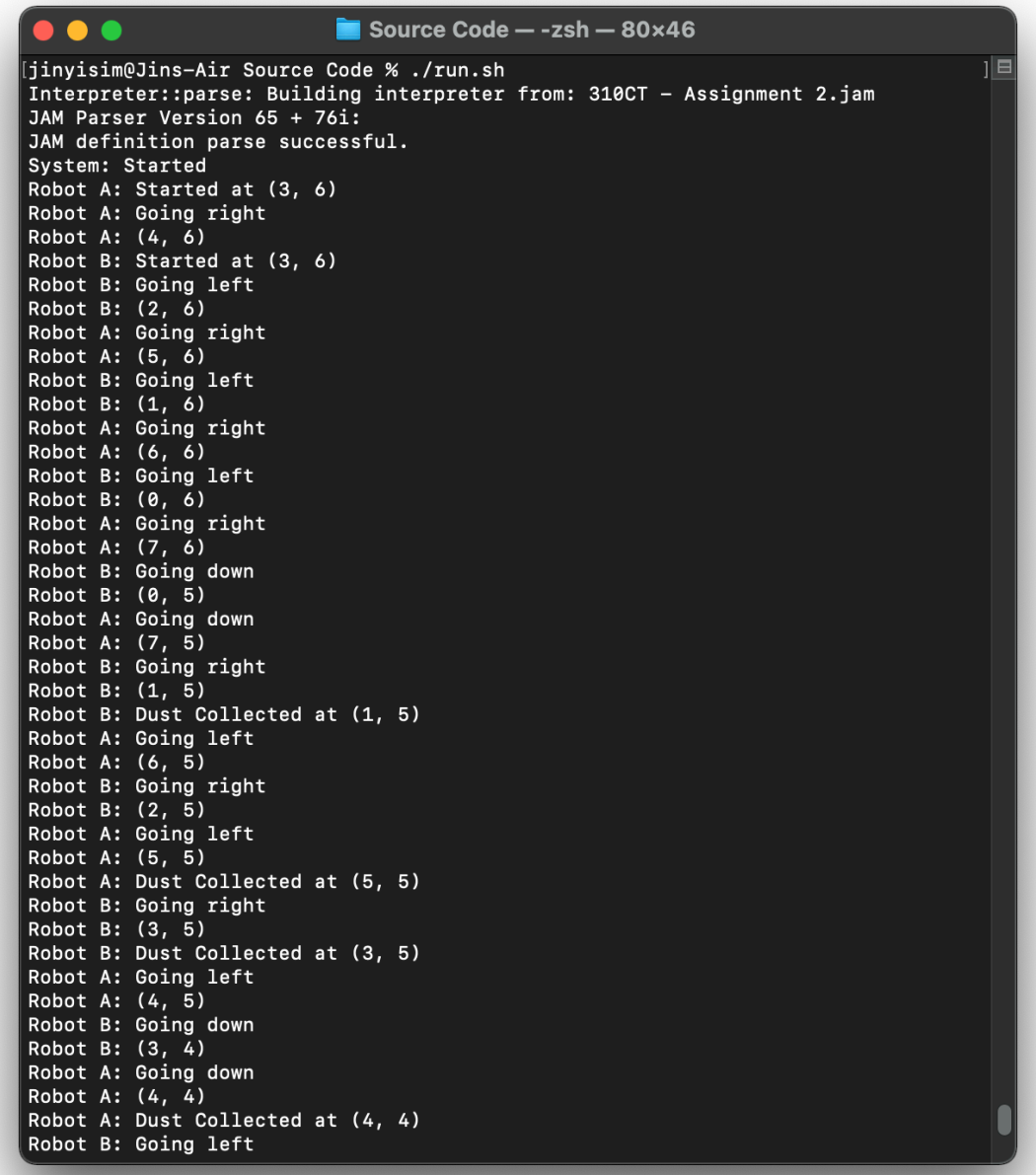

Figure 3.37: Test case 3 output (1), robot A and B both begin at (3, 6)

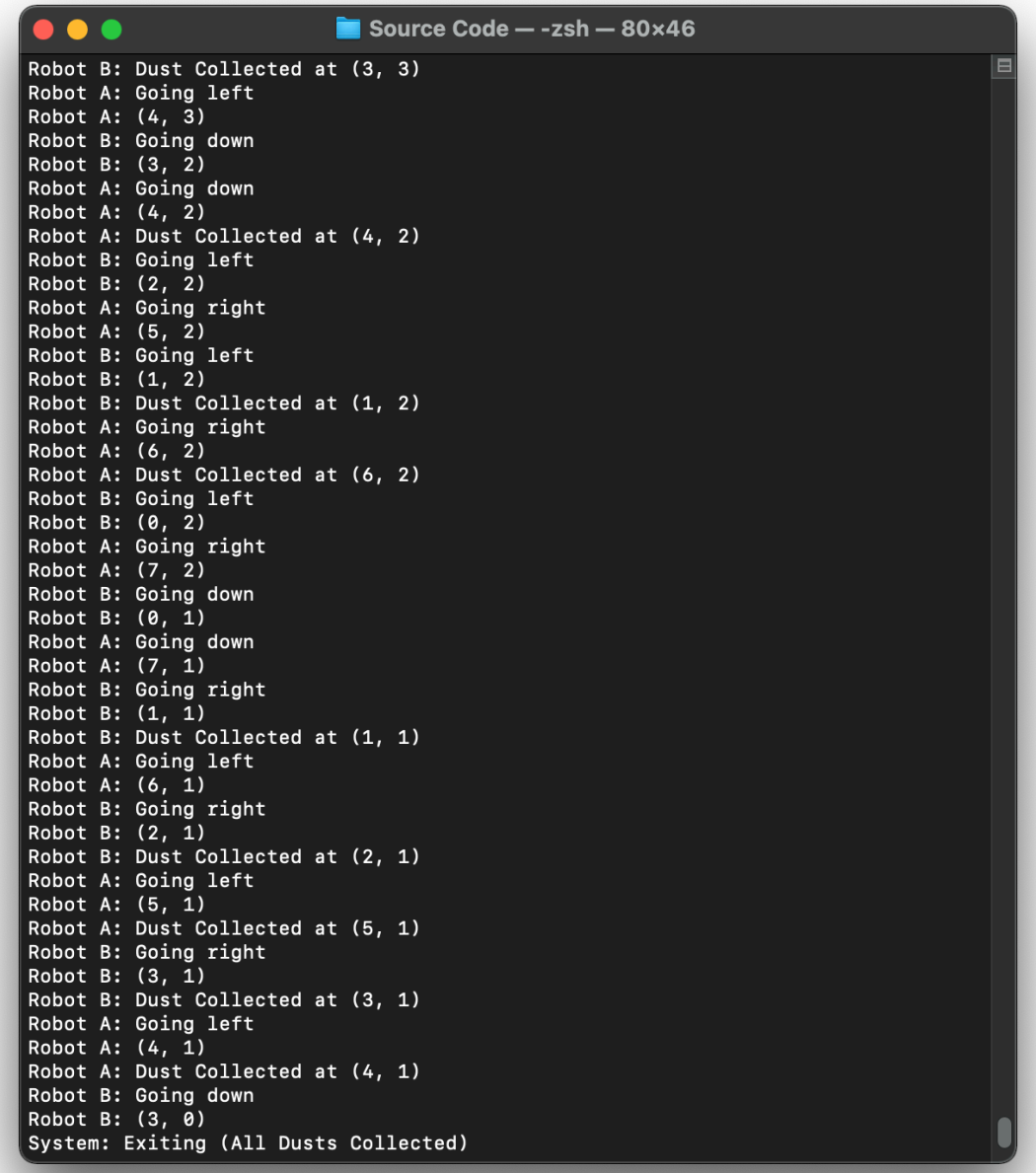

Figure 3.38: Test case 3 output (2), all the dusts were collected

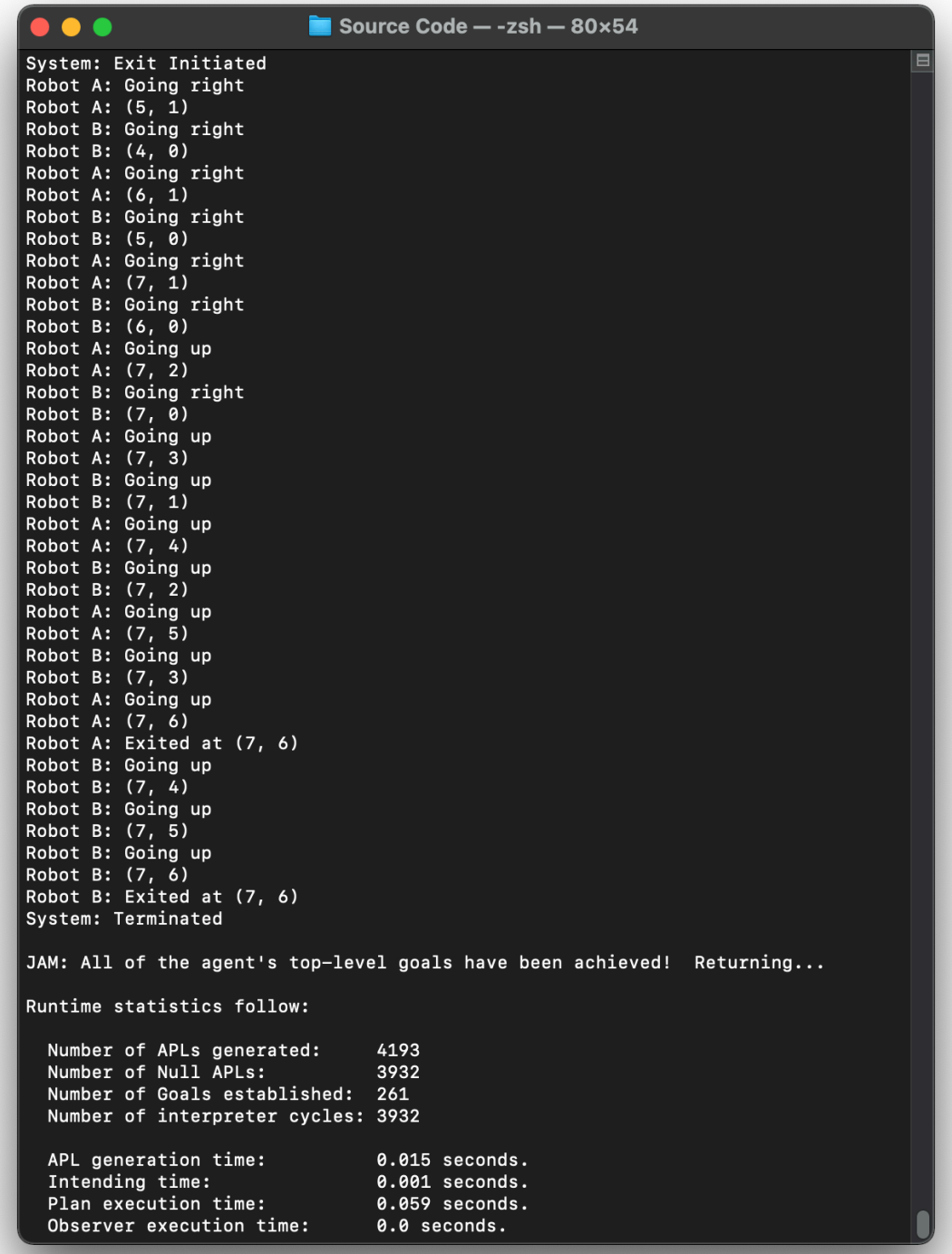

Figure 3.39: Test case 3 output (3), both robots exited from the grid at (7, 6)

3.5 Test Case (Grid Size and Exit Location)

3.5.1 Test Case 1

3.5.1.1 Initial Setting

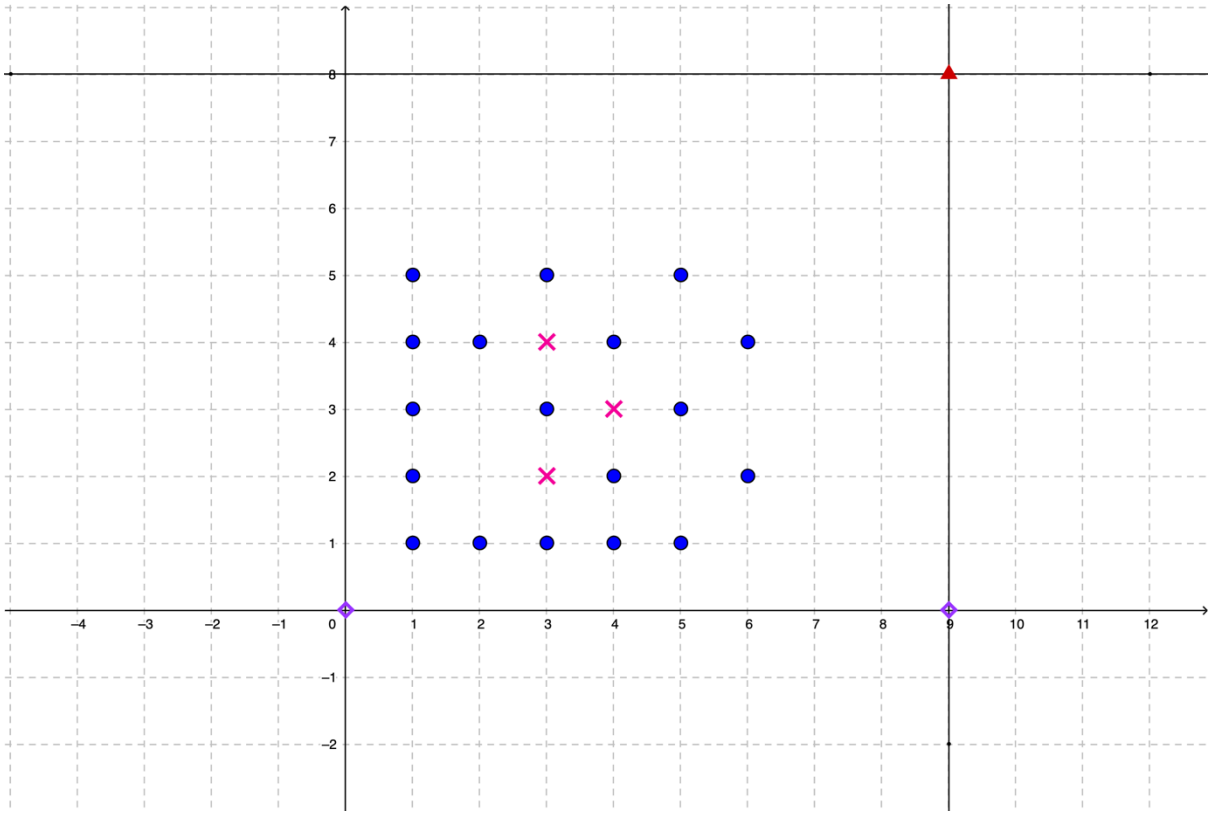

Figure 3.40: Initial placement of the dusts, obstacles, robots and exit

The boundaries of the grid were changed to $x = 9$ and $y = 8$ to test the system with different grid sizes. Robot A and B were initially placed at (0, 0) and (9, 0) respectively and the exit was placed at (9, 8). Therefore, the program will be tested by manipulating two variables, which are the boundaries of the grid and the location of the exit point changed.

The program caters to the manipulation in the grid size through the *MAX_X* and *MAX_Y* facts in the world model, whereby manipulating the facts will denote that the size of the grid has changed. This is also the case for the exit, whereby the robots will seek for the exit based on the *exit* fact in the world model. Similarly, the output below will include the three screenshots, which includes the instances when the robots begins to run, followed by when the robots completed the cleaning and when the robots have successfully returned to the exit.

3.5.1.2 Program Output (Excerpt)

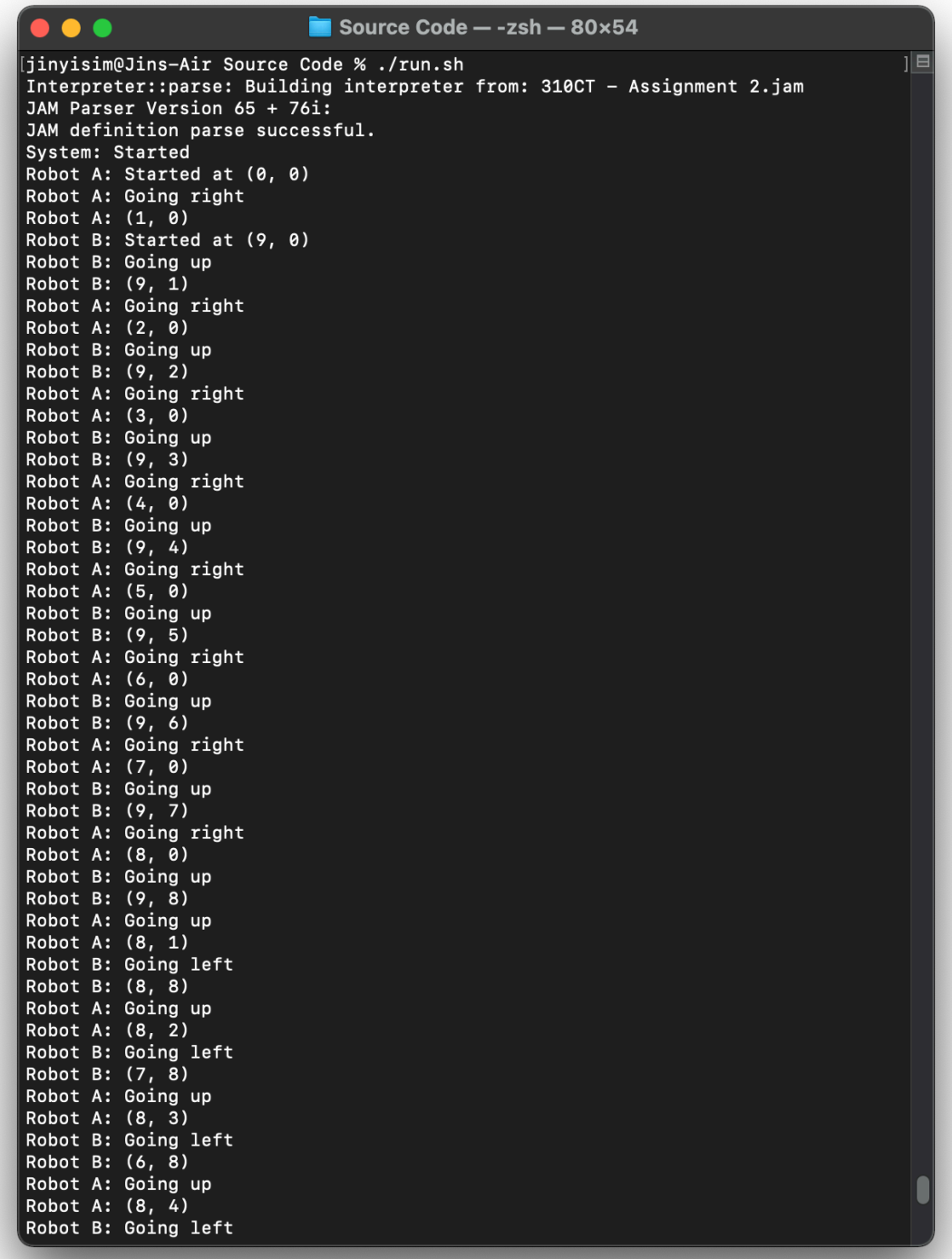

Figure 3.41: Test case 1 output (1), robot A starts at (0, 0) while robot B starts at (9, 0)

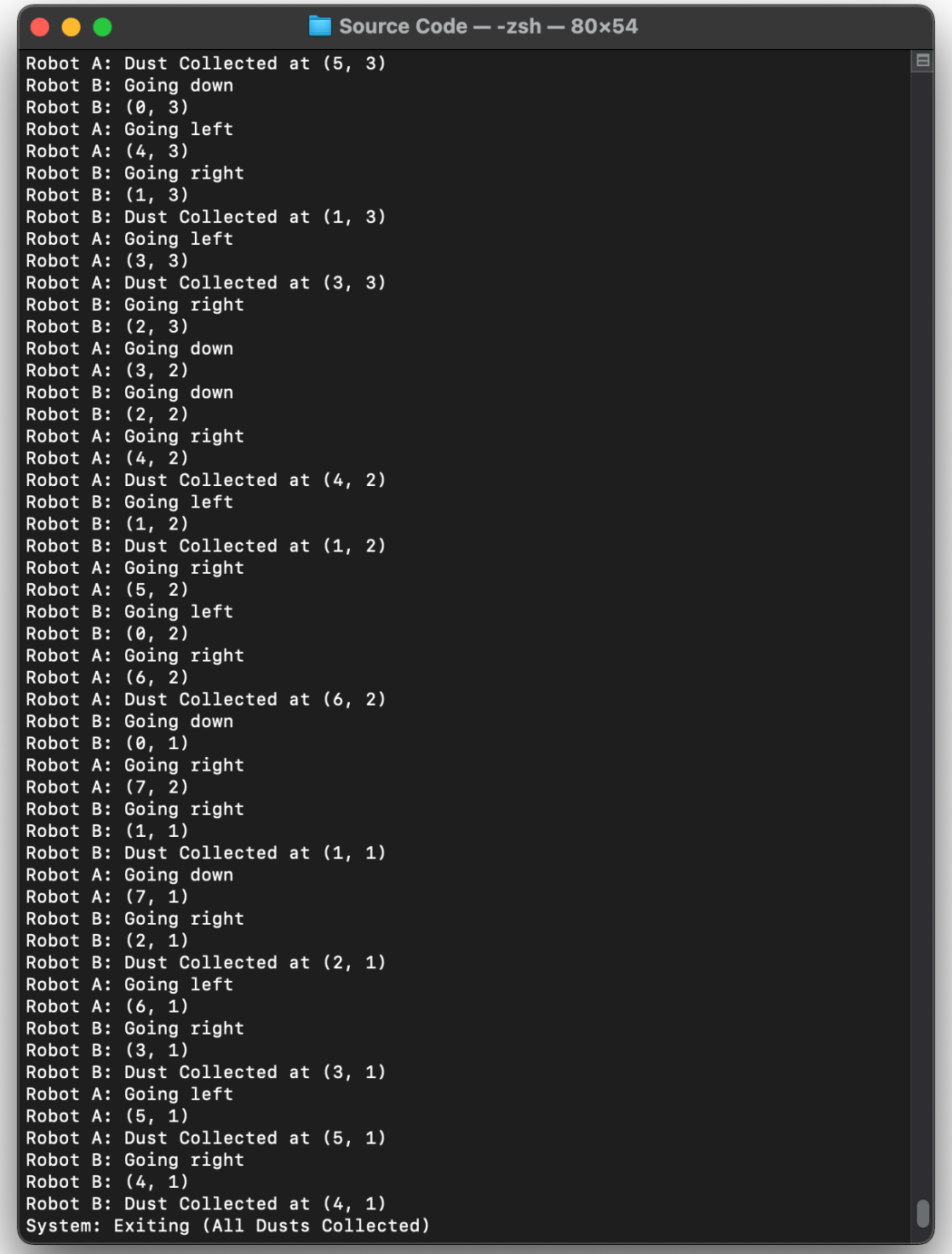

Figure 3.42: Test case 1 output (2), all the dusts were collected

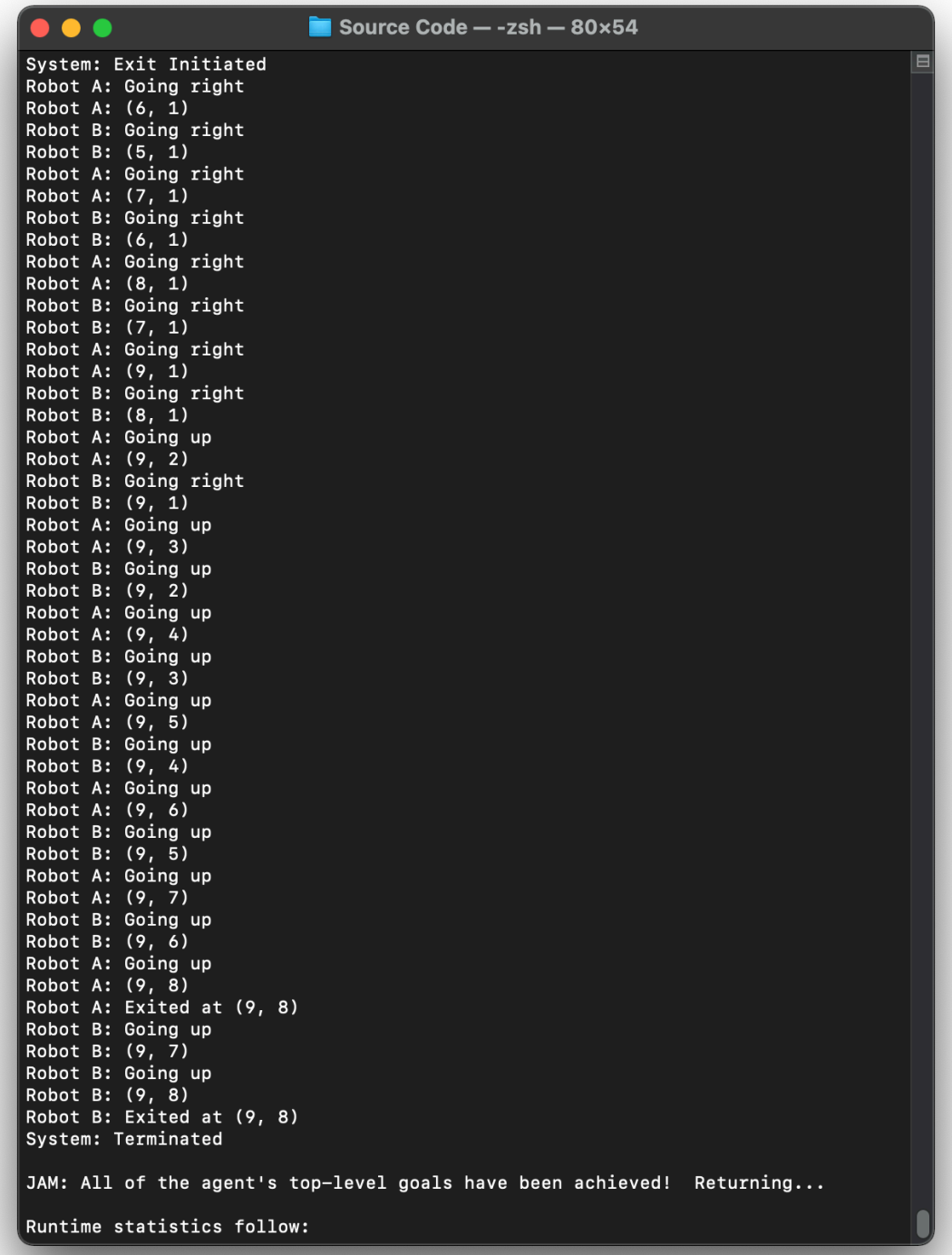

Figure 3.43: Test case 1 output (3), both robots exited successfully at (9, 8)

3.6 Test Case (Number of Robots)

3.6.1 Test Case 1

3.6.1.1 Initial Setting

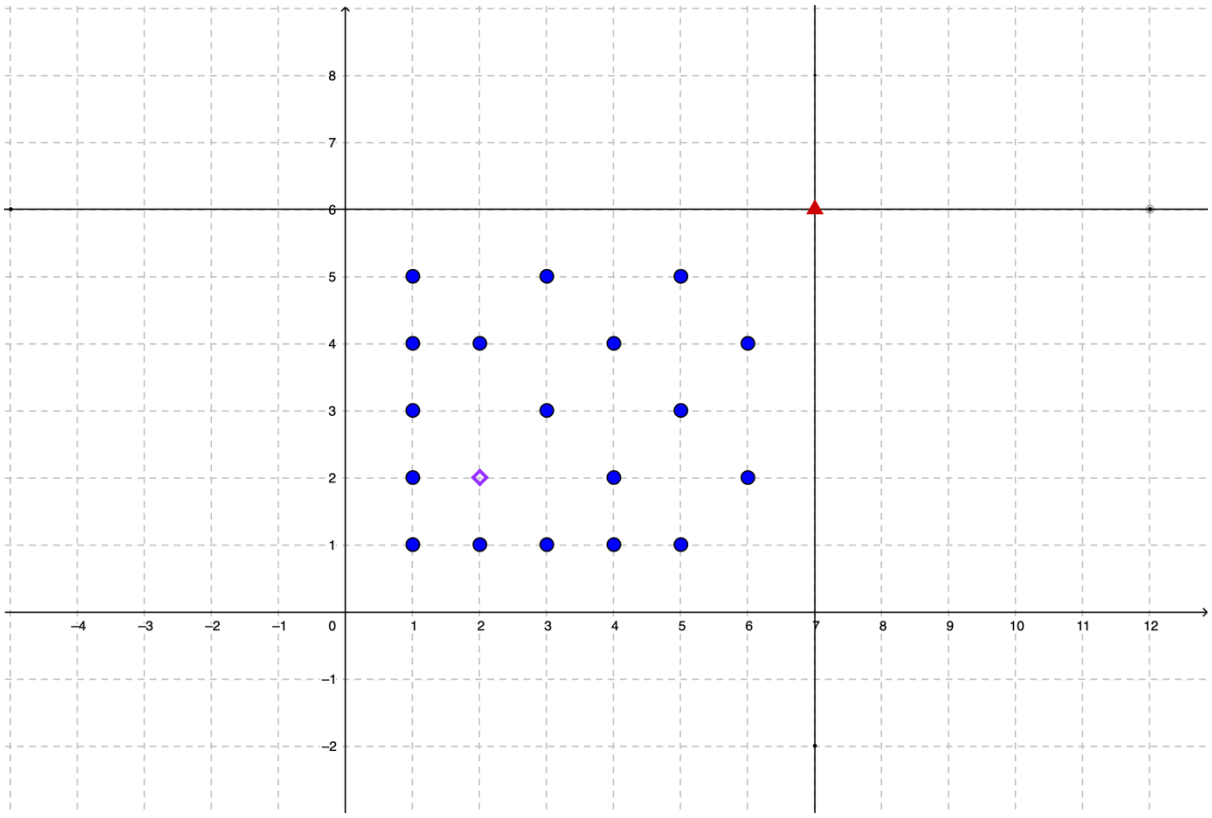

Figure 3.44: Initial placement of the dusts, obstacles, robot and exit

The test checks on the program when there is only one robot in the grid. As shown in Figure 3.44, the setting of the grid adheres to the setting given in the question. However, robot B was removed from the grid and the initial position of robot A was moved to (2, 2). This tests and ensure that the system is still functional even with only one robot operating and the robot does not have to be placed at a fixed location, which was the setting of the test case.

3.6.1.2 Program Output (Excerpt)

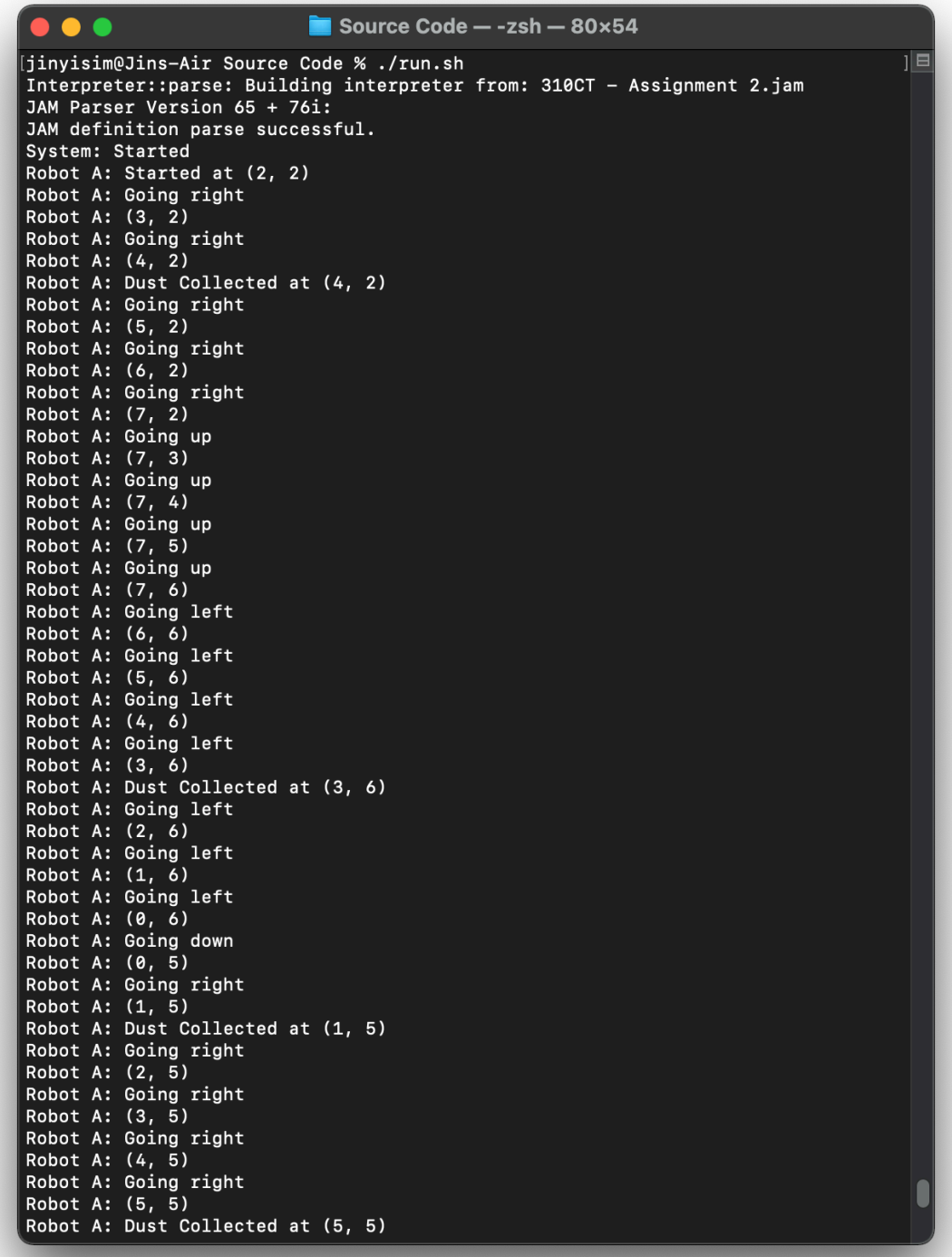

Figure 3.45: Test case 1 output (1), robot A starts at (2, 2), robot B is absent

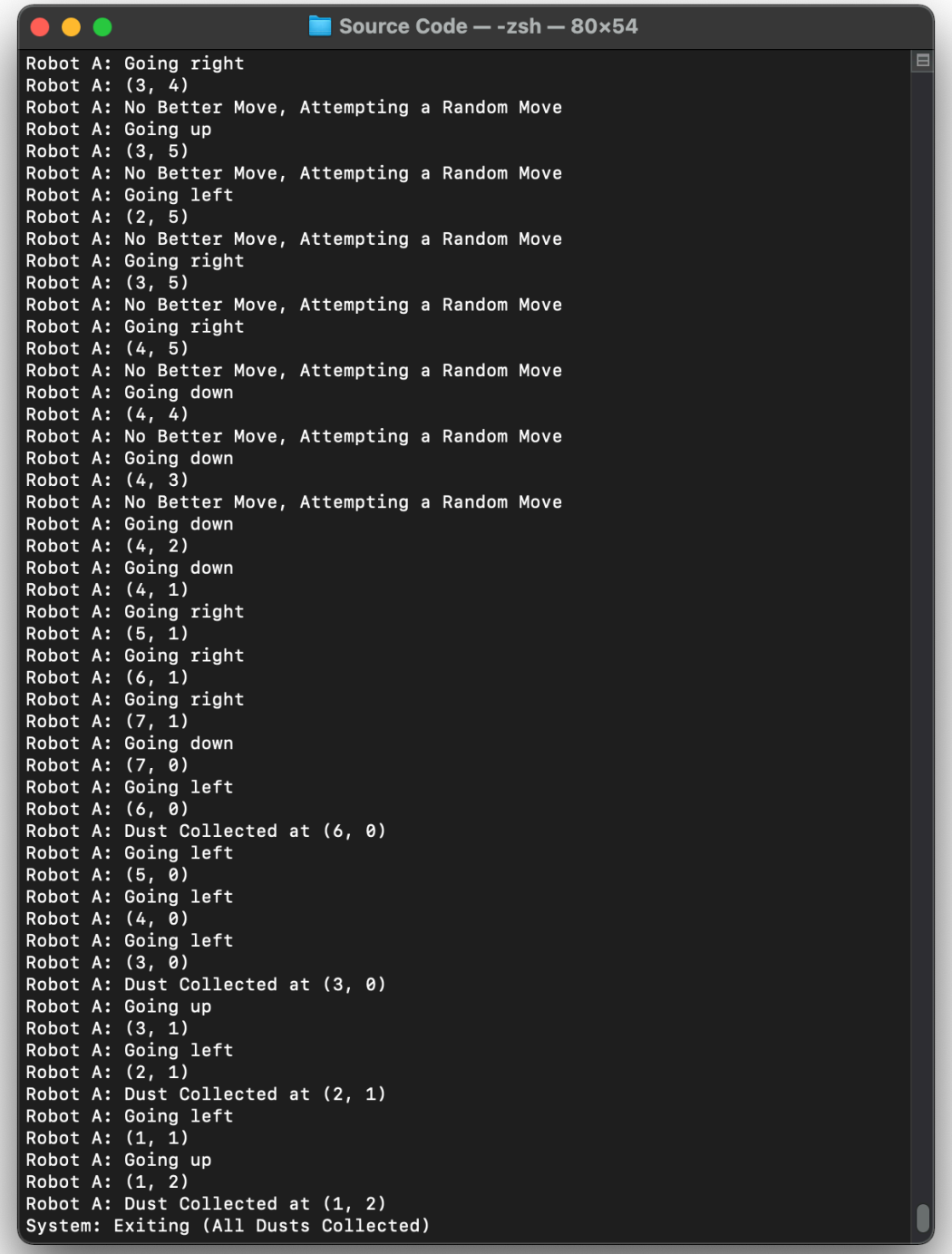

Figure 3.46: Test case 1 output (2), robot A collected all the dusts

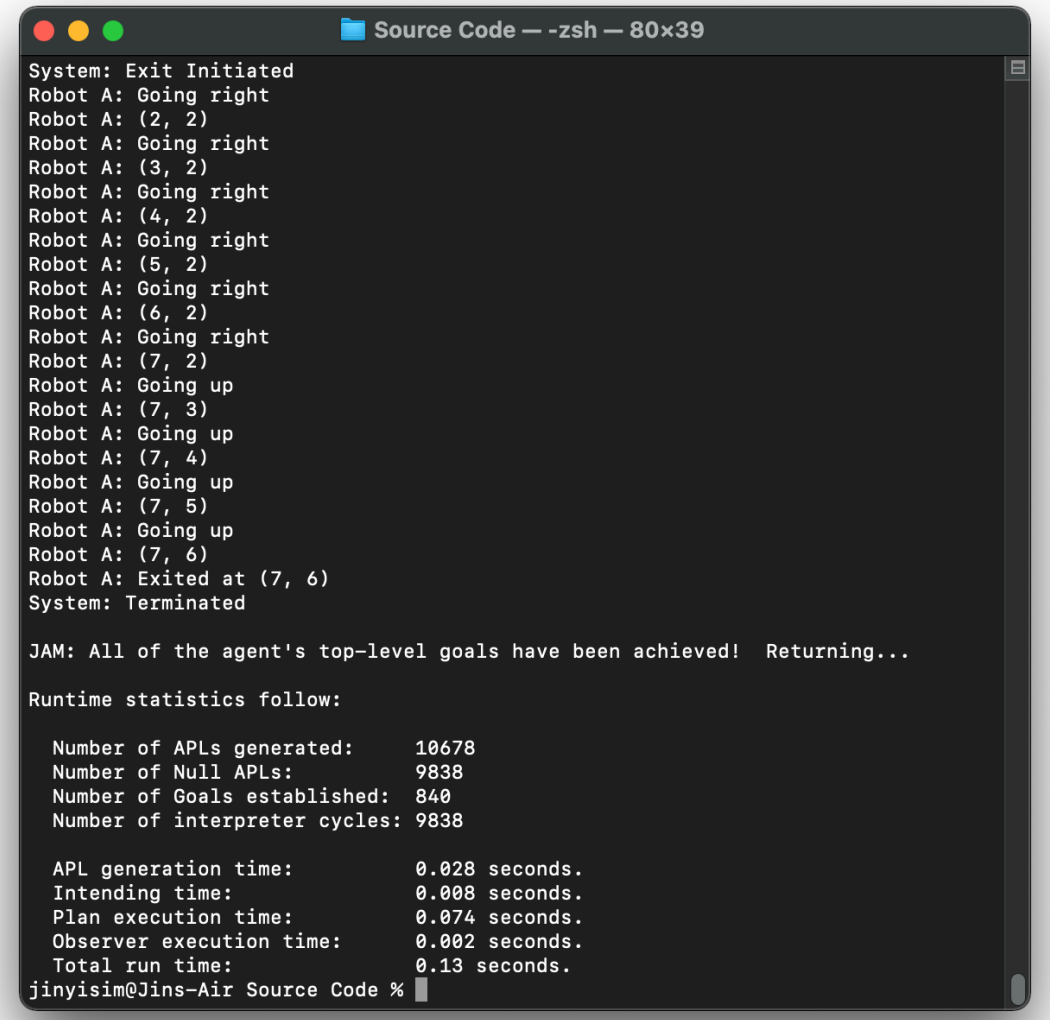

Figure 3.47: Test case 1 output (3), robot A exited from the grid at (7, 6)

3.7 Conclusion

In conclusion, the testing was performed on various scenarios such as when the dusts, obstacles, robots were placed differently on the grid, size of the grid, location of the exit and the number of robots in the grid. The results from the test cases showed that the program was implemented correctly as the outcome was entirely expected, and under no circumstance had the program fails with an error message or gets into an infinite loop which is a common manifestation of errors in the. Therefore, it was clear that the program passed all the tests. On the other hand, the results to the different scenarios were also shown in the screenshots excerpted from the program output with an explanation towards the behaviour of the robots.

4.0 Discussion

4.1 Achievements

The main achievement in the project was the introduction of heuristics that have navigated the robots effectively through the grid cells to collect all the dusts even when the robots and the obstacles were placed randomly in the grid, so long the obstacles do not block the dusts or exit completely. The heuristics were inspired by the Tabu search optimization algorithm with modifications made, which had a positive impact on the effectiveness of the navigation of the robots in prioritizing the unvisited cells. This was particularly important with the introduction of obstacles in the grid, as the fixed navigation algorithms are very limited which does not adapt well to the different placements of the obstacles. Therefore, with the navigation algorithm implemented, the cells are visited effectively by the robots by marking a cell as visited to prevent the robots from revisiting. Moreover, the heuristic on allowing the robots to move randomly when all the adjacent cells were visited was added to the robots to further enhance the algorithm, allowing the robot to continue moving in hope of encountering clusters of unvisited cells to prevent the robots from being stuck in the local optima.

Besides that, the flexibility of the program to cater to the changes in the placement of the dusts, obstacles, exit, robots and size of the grid were also the achievement for the project. This is because the implementation had taken the possible required changes into account, allowing the robots to adapt according to the different situations posed. For example, with the additional dusts and obstacles placed, the robots still manage to find a path to clean all the dusts, then exit from the grid. Also, when designing the program, different possibilities on the placement of the obstacles which can have a dramatic effect over whether if the robots can complete the *clean* top-level goal were considered. As emphasized previously, there exists scenarios which can block the dusts or exit entirely, which prevents the robot from completing the top-level goal to clean up all the dusts. With that, instead of allowing the program to go into an infinite loop, a maximum time limit was imposed to ensure that the robots give up on searching for the dusts and proceed to the exit. Should the exit also blocked, the robots will also keep searching for a path that can lead to the exit until the time limit expires which was calculated based on the grid size to mimic the vacuum robots in real-world scenarios.

4.2 Limitations and Future Enhancements with Agent Technology

The main limitation is the inability for the program to cater to an infinite number of robots in the grid. The current implementation allows a maximum of two robots in the grid, which can make the solution lacking when the size of the grid becomes large. Hence, this can be enhanced in future by using *RETRIEVEALL* and *NEXTFACT* which were used to cater to an arbitrary number of dusts and obstacles in the grid. The solution was not enhanced is due to the complexity of the code since struct or class is not supported in JAM, which increases the sophistication in manging the data related to the robots in the world model (facts). With agent technology, this can be achieved easily as social ability is one of the characteristics, allowing multiple agents to work with each other effectively. With the robots being able to socialize, more robots can be added easily since the robots can communicate with each other, which facilitate better information sharing, thus allowing the tasks to be completed effectively.

Furthermore, the solution can be enhanced in terms of the proactiveness of the robots. This is because with the current implementation, the robots do not attempt to learn and predict the location of the dusts which improves the overall efficiency. With the proactiveness characteristic under the agent technology, the robots can be equipped with the ability to learn from the past experiences, and attempt to make prediction to search more frequently on the areas with more dusts. For example, the top left corner may contain more dusts compared to other areas for every run, and the proactive robots may pick up the pattern then attempt to search more frequently on the top left corner compared to other locations. Therefore, the robots can be enhanced to become more intelligent and effective in achieving the objectives.

Besides that, the solution is currently tested to work well with the world being a static entity. Despite being able to cater to the changes in a dynamic world by updating the facts, the program was not initially designed with a dynamic world in mind. Hence, the future enhancement should focus on testing and enhancing the algorithm when the world is dynamic. This is especially on the navigation algorithm, which should be changed to allow the robots to revisit the visited cells after a certain period of time (as in the actual implementation of Tabu search) as dusts might be added to the location that was marked as visited dynamically. This was highly related to the reactivity characteristic in the agent technology, whereby the robots should be tested to ensure that they are reactive enough to the changes in the environment, meaning that the robots perform the relevant actions when the location of the dusts change, etc.

4.3 Analysis on the BDI Architecture in the Implementation

```
/* BEGIN: World Model */
FACTS:

    // Robot A (starting (0, 0))
    FACT robotA 0 0;
    FACT robotADirection "right";
    FACT robotASteps 0;

    // Robot B (starting (7, 0))
    FACT robotB 7 0;
    FACT robotBDirection "left";
    FACT robotBSteps 0;

    // List of dusts
    FACT dust 3 0;
    FACT dust 6 0;
    FACT dust 2 1;
    FACT dust 1 2;
    FACT dust 4 2;
    FACT dust 0 3;
    FACT dust 3 3;
    FACT dust 4 4;
    FACT dust 1 5;
    FACT dust 5 5;
    FACT dust 3 6;


    // List of obstacles
    // The dusts that are surrounded by obstacles cannot be cleaned
    // The robot will attempt to search through the grid for an entry
    // But if none, the robot will give up after the time limit (MAX_X * MAX_Y * 10)
    // FACT obstacle 4 1;


    // Exit location
    FACT exit 7 6;

    // Grid boundaries
    FACT MAX_X 7;
    FACT MAX_Y 6;

    // Whether if the all the grids were visited and the dusts were cleaned
    FACT dustCleaned "false";
/* END: World Model */
```

Excerpt 4.1: Facts in the program

The belief-desire-intention (BDI) is a deliberative architecture that was developed for an agent to reason based on the beliefs, desires and intentions of the agent. Firstly, beliefs are the knowledge of an agent towards the world, something the agent believes to be true on the environment it is situated in, and the beliefs might or might not be actually true *(Perez, 2019)*. In the project, the beliefs of the robots were manifested in the form of the facts at the beginning of the code, and the robots may modify the facts during runtime by using *ASSERT*, *UPDATE* AND *RETRACT*. This forms a communicative mechanism as the facts were shared between the robots, whereby both robots can share information through the facts. Referring to Excerpt 4.1, both robots have the same belief over the location of the dusts in the grid. From the beliefs, such as the presence of dusts in the grid, the robots can form its desire on what to achieve.

```
/* BEGIN: Top-level Goals */
GOALS:
    ACHIEVE clean :UTILITY 10;
    ACHIEVE return :UTILITY 1;
/* END: Top-level Goals */
```

Excerpt 4.2: Top-level goals in the program

This leads to the desire element in the BDI architecture. The desire refers to the ideal state of the environment by the agent, which means that the things that would like to be achieved in the future. Similarly, the desire might or might not be realistic *(Perez, 2019)*. In the project, the desires of the robots were manifested as the top-level goals at the beginning of the code. The robots have the desire to clean up the dusts in the grid or return to the exit. As shown in Figure 4.2, a goal can have priority over the other by specifying with the *UTILITY* keyword. The main desire of the robots should always be to clean up the grid as long as there exists at least one dust in the grid. After cleaning up all the dusts, or the time limit expires, the goal to clean the grid is considered to be achieved, thus the robots will attempt to achieve the other plan which is to return to the exit. The robots will only terminate once all the desires were achieved, and by this point, no intention has been formed and no plan execution were initiated.

The third element in the BDI architecture is the intentions. Intentions are the desires committed to by the agents, which means to be achieved by the agents. In the actual implementation in a program, this is where the agent starts to execute the plan selected based on the intention formed *(Perez, 2019)*. As discussed previously, the intention of the robots will always prioritize the desire to clean the dusts as long as the dusts are present in the grid. By then, the robots select a plan that achieves the intention to clean from the plan library and start executing. Then, the top-level plan to clean up the dusts in the grid includes other plans such as to move around (including to avoid the obstacles), check and collect the dusts, and mark the current location as visited. The same thing applies when the robots commit to the intention to return to the exit once all the dusts were cleaned or the time limit had expired, which will result in the plan achieving *return* to be selected which includes other plans to navigate the robots.

In conclusion, the analysis above shows the incorporating of the BDI architecture in the implementation of the program using JAM. The program was designed adhering to the BDI architecture which is also the nature of JAM. Hence, though the BDI architecture, multiple robots can work together by communicating with each other over a set of common beliefs towards the world, which aids the robots to form an intention from the desires (top-level goals) and commit to it by selecting a plan which achieves the intention and begins the execution.

5.0 Conclusion

In conclusion, the program was developed with JAM, adhering to the BDI architecture to showcase the operation of two communicative robots in the process to clean up the dusts in the grid, then proceed to the exit. The implementation includes eight plans to achieve both the desires to clean up the dusts and return to the exit. Additional features that were added to the project includes the implementation to allow the robots to handle the scenarios when obstacles are present in the grid. Besides that, the navigation algorithm of the robots in the grid was designed with heuristics influenced by the Tabu search algorithm with modifications to ensure that the robots search through the grid cells effectively and clean up all the dusts quickly.

In terms of the navigation, the robots attempt to first scan through all the cells in the grid deterministic to cover most of the cells which were perfectly clear from obstacles. Then, when the robots fail to identify an unvisited cell, the algorithm allows the robots to navigate randomly to find a newly unvisited cell. Other than that, the design also takes the possibility of the unreachable dusts and exit which can be blocked entirely by the obstacles and prevents the program from going into an infinite loop by adding a time constraint, whereby the robots will only search for a fixed amount of time evaluated from the size of the grid before giving up.

As for the testing and results, all the test cases have passed by examining the outputs and the results were as expected. Lastly, the enhancement that can be made on the program includes the ability to add more robots in the grid which have the social ability to ensure that the robots communicate well while performing the tasks to maximize the overall efficiency. Besides that, the robots can also be enhanced to be proactive, whereby the robots will attempt to predict the location of the dusts based on the past information and search more frequently at an area. Then, the reactivity of the robots should also be enhanced to cater to the scenario when the world changes (location of dusts, etc.) dynamically, especially on the navigation algorithm.

In short, the implementation of the robots with JAM adheres to the BDI architecture, whereby the facts in the program were the common beliefs to the robots. Then, the desires of the robots are denoted by the top-level goals to be achieved. Lastly, the intention is the subset of the desires which was selected to be executed. Here, the plan that fulfills the intention selected by the robots will be selected from the plan library and the plan execution begins.

6.0 References

1. Baguley, R., & McDonald, C. (2015). *Appliance Science: How Robotic Vacuums Navigate - CNET*. CNET. https://www.cnet.com/news/appliance-science-how-robotic-vacuums-navigate/
2. Perez, A. (2019). *Machine Learning - Leveraging the Beliefs-Desires-Intentions Agent Architecture*. Microsoft Docs. https://docs.microsoft.com/en-us/archive/msdn-magazine/2019/january/machine-learning-leveraging-the-beliefs-desires-intentions-agent-architecture

Appendix A

Complete JAM Source Code

```
/*
 * Name: Sim Jin Yi (P17008744 / 9658521)
 * Date: October 22, 2020
 * Description: 310CT - Assignment 2
 */

/* BEGIN: Top-level Goals */
GOALS:
    ACHIEVE clean :UTILITY 10;
    ACHIEVE return :UTILITY 1;
/* END: Top-level Goals */


/* BEGIN: World Model */
FACTS:

    // Robot A (starting (0, 0))
    FACT robotA 0 0;
    FACT robotADirection "right";
    FACT robotASteps 0;

    // Robot B (starting (7, 0))
    FACT robotB 7 0;
    FACT robotBDirection "left";
    FACT robotBSteps 0;

    // List of dusts
    FACT dust 3 0;
    FACT dust 6 0;
    FACT dust 2 1;
    FACT dust 1 2;
    FACT dust 4 2;
    FACT dust 0 3;
    FACT dust 3 3;
    FACT dust 4 4;
    FACT dust 1 5;
    FACT dust 5 5;
    FACT dust 3 6;

    // List of obstacles
    // The dusts that are surrounded by obstacles cannot be cleaned
    // The robot will attempt to search through the grid for an entry
    // But if none, the robot will give up after the time limit (MAX_X * MAX_Y * 10)
    // FACT obstacle 4 1;

    // Exit location
```

```
    FACT exit 7 6;

    // Grid boundaries
    FACT MAX_X 7;
    FACT MAX_Y 6;

    // Whether if the all the grids were visited and the dusts were cleaned
    FACT dustCleaned "false";
/* END: World Model */


/* BEGIN: PLAN clean */
PLAN:
{
NAME:
    "Walkthrough the grids and clean the dusts"

DOCUMENTATION:
    "The plan achieves the top-level goal to clean the dusts,
        - The robots will attempt to clean until either,
            - All the dusts were cleaned.
            - The time limit had expired.
        - Both robot will make a move in each iteration.
        - The checking on the status of the grid as follow will be performed every iteration:
            - Whether if all the dusts were cleaned.
            - Whether if the time limit had expired."

GOAL:
    ACHIEVE clean;

BODY:

    EXECUTE println "System: Started";

    // Perform the initial check on the status
    PERFORM check;

    // Obtain the grid boundaries and the time limit to clean
    RETRIEVE dustCleaned $dustCleaned;
    RETRIEVE MAX_X $MAX_X;
    RETRIEVE MAX_Y $MAX_Y;

    ASSIGN $MAX_TRIES (* $MAX_X $MAX_Y 10);
    ASSIGN $iteration 0;

    // Loop to keep the robot roaming around the grid until,
    //  - All the dusts were cleaned.
    //  - The time limit had expired.
    WHILE: TEST(&& (== $dustCleaned "false") (< $iteration $MAX_TRIES)) {

        // Advances one step for both robot
```

```
        PERFORM run "robotA";
        PERFORM run "robotB";

        // Check and retrieve the status of the grid
        PERFORM check;
        RETRIEVE dustCleaned $dustCleaned;
        ASSIGN $iteration (+ $iteration 1);
    };

    EXECUTE print "System: Exiting ";
    WHEN: TEST(== $dustCleaned "true") {
        EXECUTE print "(All Dusts Collected)";
    };

    EXECUTE println "";

FAILURE:
    EXECUTE println "System: Failed to Clean";
}
/* END: PLAN clean */


/* BEGIN: PLAN return */
PLAN:
{
NAME:
    "Navigate the robots to the exit"

DOCUMENTATION:
    "The plan navigates the robots to the exit after,
        - All the dusts were cleaned.
        - The time limit to clean had expired.
    If there exists obstacles blocking the pathway to the exit,
        - The plan continues to try to navigate the robots to the exit.
        - The plan stops after the time limit had expired, whereby the exit was not
achievable in time."

GOAL:
    ACHIEVE return;

BODY:

    // Assign the maximum iterations (time limit) to try to navigate the robots to the exit
    EXECUTE println "System: Exit Initiated";
    RETRIEVE MAX_X $MAX_X;
    RETRIEVE MAX_Y $MAX_Y;
    ASSIGN $MAX_TRIES (* $MAX_X $MAX_Y 10);
    ASSIGN $iteration 0;

    // Retrieve the location of the robots and exit
    RETRIEVE robotA $aX $aY;
```

```
    RETRIEVE robotB $bX $bY;
    RETRIEVE exit $exitX $exitY;

    // Loop to keep the robot walking towards the exit until,
    //  - Both robots have exited.
    //  - The time limit had expired.
    WHILE: TEST(&& (|| (!= $aX $exitX) (!= $aY $exitY) (!= $bX $exitX) (!= $bY $exitY)) (<
$iteration $MAX_TRIES)) {

        // Advances one step for both robots towards the exit
        PERFORM quit "robotA";
        PERFORM quit "robotB";

        // Retrieve the location of the robots
        RETRIEVE robotA $aX $aY;
        RETRIEVE robotB $bX $bY;
        ASSIGN $iteration (+ $iteration 1);
    };

    EXECUTE println "System: Terminated";

FAILURE:
    EXECUTE println "System: Failed to Exit";
}
/* END: PLAN return */


/* BEGIN: PLAN run $robot */
PLAN:
{
NAME:
    "Decide and advances the selected robot for one step"

DOCUMENTATION:
    "The plan achieves the sub-goal to advance the selected robot for one step,
        - The plan selects the best cell for the selected robot to advance next based on
heuristics.
            - The best cell is selected based on,
                - Whether if the cell was visited.
                - Whether if the next cell is still within the boundaries of the grid.
                - Whether if the next cell is an obstacle.
        - Once there is no better cell, the plan advances the robot randomly within the grid.
        - This hopes that the robot will eventually find an uncollected dust.
        - In the process, the robot will:
            - Update the fact on the visited cells.
            - Attempt to check and collect the dusts."

GOAL:
    ACHIEVE run $robot;

BODY:
```

```
    RETRIEVE MAX_X $MAX_X;
    RETRIEVE MAX_Y $MAX_Y;

    // Robot A selected
    WHEN: TEST(== $robot "robotA") {
        RETRIEVE robotA $x $y;
        RETRIEVE robotASteps $iteration;

        WHEN: TEST(<= $iteration 0) {
            EXECUTE println "Robot A: Started at (" $x ", " $y ")";
            PERFORM collect "robotA";
            PERFORM store "robotA";
        };


        /* BEGIN: Robot Movement Logic */
        // Variables to store the valid moves
        ASSIGN $up "true";
        ASSIGN $down "true";
        ASSIGN $left "true";
        ASSIGN $right "true";
        ASSIGN $moved "false";

        // Check which direction is valid and can be moved to base on,
        // - Whether if the robot is at the boundary.
        // - Whether if the cell to be advanced to was visited.
        // - Whether if the cell to be advanced to was blocked by an obstacle.
        WHEN: TEST(|| (<= $x 0) (FACT visited (- $x 1) $y) (FACT obstacle (- $x 1) $y)) {
            ASSIGN $left "false";
        };

        WHEN: TEST(|| (>= $x $MAX_X) (FACT visited (+ $x 1) $y) (FACT obstacle (+ $x 1) $y))
{
            ASSIGN $right "false";
        };

        WHEN: TEST(|| (<= $y 0) (FACT visited $x (- $y 1)) (FACT obstacle $x (- $y 1))) {
            ASSIGN $down "false";
        };

        WHEN: TEST(|| (>= $y $MAX_Y) (FACT visited $x (+ $y 1)) (FACT obstacle $x (+ $y 1)))
{
            ASSIGN $up "false";
        };

        // Check if there is no better move
        WHEN: TEST(&& (== $up "false") (== $down "false") (== $left "false") (== $right
"false")) {

            EXECUTE println "Robot A: No Better Move, Attempting a Random Move";
```

```
    // If the robot is at the boundary, move in the opposite direction
    WHEN: TEST(<= $x 0) {
        UPDATE (robotADirection) (robotADirection "right");
        ASSIGN $moved "true";
    };

    WHEN: TEST(<= $y 0) {
        UPDATE (robotADirection) (robotADirection "up");
        ASSIGN $moved "true";
    };

    WHEN: TEST(>= $x $MAX_X) {
        UPDATE (robotADirection) (robotADirection "left");
        ASSIGN $moved "true";
    };

    WHEN: TEST(>= $y $MAX_Y) {
        UPDATE (robotADirection) (robotADirection "down");
        ASSIGN $moved "true";
    };

    // If the robot is not at the boundary, attempt to make a move randomly
    WHEN: TEST(== $moved "false") {
        DO_ANY {
            UPDATE (robotADirection) (robotADirection "right");
            ASSIGN $moved "true";
        } {
            UPDATE (robotADirection) (robotADirection "up");
            ASSIGN $moved "true";
        } {
            UPDATE (robotADirection) (robotADirection "left");
            ASSIGN $moved "true";
        } {
            UPDATE (robotADirection) (robotADirection "down");
            ASSIGN $moved "true";
        };
    };
};

// Attempt to move rightward whenever possible
WHEN: TEST(&& (== $right "true") (== $moved "false")) {
    UPDATE (robotADirection) (robotADirection "right");
    ASSIGN $moved "true";
};

// Otherwise, attempt to move upward
WHEN: TEST(&& (== $up "true") (== $moved "false")) {
    UPDATE (robotADirection) (robotADirection "up");
    ASSIGN $moved "true";
};
```

```
        // Otherwise, attempt to move leftward
        WHEN: TEST(&& (== $left "true") (== $moved "false")) {
            UPDATE (robotADirection) (robotADirection "left");
            ASSIGN $moved "true";
        };

        // If all else fails, attempt to move downward
        WHEN: TEST(&& (== $down "true") (== $moved "false")) {
            UPDATE (robotADirection) (robotADirection "down");
            ASSIGN $moved "true";
        };
        /* END: Robot Movement Logic */


        // Move the robot in the decided direction for one step
        // Store the location as visited in the world model (fact)
        // Check and collect the dust if available at the location
        RETRIEVE robotADirection $direction;
        PERFORM move "robotA" $direction;
        PERFORM store "robotA";
        PERFORM collect "robotA";

        UPDATE (robotASteps) (robotASteps (+ $iteration 1));
    };


    // Please refer to the comments in the section above
    // The code is identical but applicable for Robot B
    WHEN: TEST(== $robot "robotB") {
        RETRIEVE robotB $x $y;
        RETRIEVE robotBSteps $iteration;

        // First iteration
        WHEN: TEST(<= $iteration 0) {
            EXECUTE println "Robot B: Started at (" $x ", " $y ")";
            PERFORM collect "robotB";
            PERFORM store "robotB";
        };

        ASSIGN $up "true";
        ASSIGN $down "true";
        ASSIGN $left "true";
        ASSIGN $right "true";
        ASSIGN $moved "false";

        WHEN: TEST(|| (<= $x 0) (FACT visited (- $x 1) $y) (FACT obstacle (- $x 1) $y)) {
            ASSIGN $left "false";
        };

        WHEN: TEST(|| (>= $x $MAX_X) (FACT visited (+ $x 1) $y) (FACT obstacle (+ $x 1) $y))
{
```

```
            ASSIGN $right "false";
        };

        WHEN: TEST(|| (<= $y 0) (FACT visited $x (- $y 1)) (FACT obstacle $x (- $y 1))) {
            ASSIGN $down "false";
        };

        WHEN: TEST(|| (>= $y $MAX_Y) (FACT visited $x (+ $y 1)) (FACT obstacle $x (+ $y 1)))
{
            ASSIGN $up "false";
        };

        WHEN: TEST(&& (== $up "false") (== $down "false") (== $left "false") (== $right
"false")) {

            EXECUTE println "Robot B: No Better Move, Attempting a Random Move";

            WHEN: TEST(<= $x 0) {
                UPDATE (robotBDirection) (robotBDirection "right");
                ASSIGN $moved "true";
            };

            WHEN: TEST(<= $y 0) {
                UPDATE (robotBDirection) (robotBDirection "up");
                ASSIGN $moved "true";
            };

            WHEN: TEST(>= $x $MAX_X) {
                UPDATE (robotBDirection) (robotBDirection "left");
                ASSIGN $moved "true";
            };

            WHEN: TEST(>= $y $MAX_Y) {
                UPDATE (robotBDirection) (robotBDirection "down");
                ASSIGN $moved "true";
            };

            WHEN: TEST(== $moved "false") {
                DO_ANY {
                    UPDATE (robotBDirection) (robotBDirection "right");
                    ASSIGN $moved "true";
                } {
                    UPDATE (robotBDirection) (robotBDirection "up");
                    ASSIGN $moved "true";
                } {
                    UPDATE (robotBDirection) (robotBDirection "left");
                    ASSIGN $moved "true";
                } {
                    UPDATE (robotBDirection) (robotBDirection "down");
                    ASSIGN $moved "true";
                };
```

```
            };
        };

        WHEN: TEST(&& (== $right "true") (== $moved "false")) {
            UPDATE (robotBDirection) (robotBDirection "right");
            ASSIGN $moved "true";
        };

        WHEN: TEST(&& (== $up "true") (== $moved "false")) {
            UPDATE (robotBDirection) (robotBDirection "up");
            ASSIGN $moved "true";
        };

        WHEN: TEST(&& (== $left "true") (== $moved "false")) {
            UPDATE (robotBDirection) (robotBDirection "left");
            ASSIGN $moved "true";
        };

        WHEN: TEST(&& (== $down "true") (== $moved "false")) {
            UPDATE (robotBDirection) (robotBDirection "down");
            ASSIGN $moved "true";
        };

        RETRIEVE robotBDirection $direction;
        PERFORM move "robotB" $direction;
        PERFORM store "robotB";
        PERFORM collect "robotB";

        UPDATE (robotBSteps) (robotBSteps (+ $iteration 1));
    };

FAILURE:
    EXECUTE println "System: Failed to Run - " $robot;
}
/* END: PLAN run $robot */


/* BEGIN: PLAN quit $robot */
PLAN:
{
NAME:
    "Decide and advances the selected robot for one step towards the exit"

DOCUMENTATION:
    "The plan attempts to navigate the robot towards the exit using heuristics,
        - The heuristic tries to move the robot along the obstacles or boundaries.
        - The movement prioritize the horizontal motion towards the right.
        - This helps in ensuring that the robot gets closer to the exit even when blocked by
obstacles."

GOAL:
```

```
    ACHIEVE quit $robot;

BODY:
    RETRIEVE MAX_X $MAX_X;
    RETRIEVE MAX_Y $MAX_Y;
    RETRIEVE exit $exitX $exitY;

    // Robot A selected
    WHEN: TEST(== $robot "robotA") {
        RETRIEVE robotA $x $y;

        // Check if the robot is already at the exit
        // Perform the movement only if the robot is not already at the exit
        WHEN: TEST(|| (!= $x $exitX) (!= $y $exitY)) {


            /* BEGIN: Robot Movement Logic */
            // Variables to store the valid moves
            ASSIGN $up "true";
            ASSIGN $down "true";
            ASSIGN $left "true";
            ASSIGN $right "true";
            ASSIGN $moved "false";

            // Check which are the valid moves
            WHEN: TEST(|| (<= $x 0) (FACT robotAVisited (- $x 1) $y) (FACT obstacle (- $x 1)
$y)) {
                ASSIGN $left "false";
            };

            WHEN: TEST(|| (>= $x $MAX_X) (FACT robotAVisited (+ $x 1) $y) (FACT obstacle (+
$x 1) $y)) {
                ASSIGN $right "false";
            };

            WHEN: TEST(|| (<= $y 0) (FACT robotAVisited $x (- $y 1)) (FACT obstacle $x (- $y
1))) {
                ASSIGN $down "false";
            };

            WHEN: TEST(|| (>= $y $MAX_Y) (FACT robotAVisited $x (+ $y 1)) (FACT obstacle $x
(+ $y 1))) {
                ASSIGN $up "false";
            };

            // If there is no other move that leads better to the exit,
            // Perform a random move
            WHEN: TEST(&& (== $up "false") (== $down "false") (== $left "false") (== $right
"false")) {

                EXECUTE println "Robot A: No Better Move to Exit, Attempting a Random Move";
```

```
            // If the robot is at the boundaries, move in the opposite direction
            WHEN: TEST(<= $x 0) {
                UPDATE (robotADirection) (robotADirection "right");
                ASSIGN $moved "true";
            };

            WHEN: TEST(<= $y 0) {
                UPDATE (robotADirection) (robotADirection "up");
                ASSIGN $moved "true";
            };

            WHEN: TEST(>= $x $MAX_X) {
                UPDATE (robotADirection) (robotADirection "left");
                ASSIGN $moved "true";
            };

            WHEN: TEST(>= $y $MAX_Y) {
                UPDATE (robotADirection) (robotADirection "down");
                ASSIGN $moved "true";
            };

            // Attempt to make a random move if the robot is not at the boundaries
            WHEN: TEST(== $moved "false") {
                DO_ANY {
                    UPDATE (robotADirection) (robotADirection "right");
                    ASSIGN $moved "true";
                } {
                    UPDATE (robotADirection) (robotADirection "up");
                    ASSIGN $moved "true";
                } {
                    UPDATE (robotADirection) (robotADirection "left");
                    ASSIGN $moved "true";
                } {
                    UPDATE (robotADirection) (robotADirection "down");
                    ASSIGN $moved "true";
                };
            };
        };

        // The following logic helps to navigate the robot in moving along the obstacles
or boundaries
        // Attempt to move towards the right if possible (prioritizing the movement to
the right)
        WHEN: TEST(&& (== $right "true") (== $moved "false")) {
            UPDATE (robotADirection) (robotADirection "right");
            ASSIGN $moved "true";
        };

        // Otherwise, attempt to move upwards
        WHEN: TEST(&& (== $up "true") (== $moved "false")) {
```

```
                UPDATE (robotADirection) (robotADirection "up");
                ASSIGN $moved "true";
            };

            // Otherwise, try to move towards the left
            WHEN: TEST(&& (== $left "true") (== $moved "false")) {
                UPDATE (robotADirection) (robotADirection "left");
                ASSIGN $moved "true";
            };

            // If all else fail, move towards the bottom
            // This is a heuristic in hope to find a path to the exit
            WHEN: TEST(&& (== $down "true") (== $moved "false")) {
                UPDATE (robotADirection) (robotADirection "down");
                ASSIGN $moved "true";
            };
            /* END: Robot Movement Logic */


            // Move the robot in the decided direction for one step
            // Mark the current location as visited for the robot
            RETRIEVE robotADirection $direction;
            ASSERT robotAVisited $x $y;
            PERFORM move "robotA" $direction;

            RETRIEVE robotA $x $y;
            WHEN: TEST(&& (== $x $exitX) (== $y $exitY)) {
                EXECUTE println "Robot A: Exited at (" $exitX ", " $exitY ")";
            };
        };
    };

    // Please refer to the comments in the section above
    // The code is identical but applicable for Robot B
    WHEN: TEST(== $robot "robotB") {
        RETRIEVE robotB $x $y;

        WHEN: TEST(|| (!= $x $exitX) (!= $y $exitY)) {
            ASSIGN $up "true";
            ASSIGN $down "true";
            ASSIGN $left "true";
            ASSIGN $right "true";
            ASSIGN $moved "false";

            WHEN: TEST(|| (<= $x 0) (FACT robotBVisited (- $x 1) $y) (FACT obstacle (- $x 1)
$y)) {
                ASSIGN $left "false";
            };

            WHEN: TEST(|| (>= $x $MAX_X) (FACT robotBVisited (+ $x 1) $y) (FACT obstacle (+
$x 1) $y)) {
```

```
                ASSIGN $right "false";
            };

            WHEN: TEST(|| (<= $y 0) (FACT robotBVisited $x (- $y 1)) (FACT obstacle $x (- $y
1))) {
                ASSIGN $down "false";
            };

            WHEN: TEST(|| (>= $y $MAX_Y) (FACT robotBVisited $x (+ $y 1)) (FACT obstacle $x
(+ $y 1))) {
                ASSIGN $up "false";
            };

            WHEN: TEST(&& (== $up "false") (== $down "false") (== $left "false") (== $right
"false")) {

                EXECUTE println "Robot B: No Better Move to Exit, Attempting a Random Move";

                WHEN: TEST(<= $x 0) {
                    UPDATE (robotBDirection) (robotBDirection "right");
                    ASSIGN $moved "true";
                };

                WHEN: TEST(<= $y 0) {
                    UPDATE (robotBDirection) (robotBDirection "up");
                    ASSIGN $moved "true";
                };

                WHEN: TEST(>= $x $MAX_X) {
                    UPDATE (robotBDirection) (robotBDirection "left");
                    ASSIGN $moved "true";
                };

                WHEN: TEST(>= $y $MAX_Y) {
                    UPDATE (robotBDirection) (robotBDirection "down");
                    ASSIGN $moved "true";
                };

                WHEN: TEST(== $moved "false") {
                    DO_ANY {
                        UPDATE (robotBDirection) (robotBDirection "right");
                        ASSIGN $moved "true";
                    } {
                        UPDATE (robotBDirection) (robotBDirection "up");
                        ASSIGN $moved "true";
                    } {
                        UPDATE (robotBDirection) (robotBDirection "left");
                        ASSIGN $moved "true";
                    } {
                        UPDATE (robotBDirection) (robotBDirection "down");
                        ASSIGN $moved "true";
```

```
                };
            };
        };

        WHEN: TEST(&& (== $right "true") (== $moved "false")) {
            UPDATE (robotBDirection) (robotBDirection "right");
            ASSIGN $moved "true";
        };

        WHEN: TEST(&& (== $up "true") (== $moved "false")) {
            UPDATE (robotBDirection) (robotBDirection "up");
            ASSIGN $moved "true";
        };

        WHEN: TEST(&& (== $left "true") (== $moved "false")) {
            UPDATE (robotBDirection) (robotBDirection "left");
            ASSIGN $moved "true";
        };

        WHEN: TEST(&& (== $down "true") (== $moved "false")) {
            UPDATE (robotBDirection) (robotBDirection "down");
            ASSIGN $moved "true";
        };

        RETRIEVE robotBDirection $direction;
        ASSERT robotBVisited $x $y;
        PERFORM move "robotB" $direction;

        RETRIEVE robotB $x $y;
        WHEN: TEST(&& (== $x $exitX) (== $y $exitY)) {
            EXECUTE println "Robot B: Exited at (" $exitX ", " $exitY ")";
        };
    };
};

FAILURE:
    EXECUTE println "System: Failed to Quit - " $robot;
}
/* END: PLAN quit $robot */


/* BEGIN: PLAN move $robot $direction */
PLAN:
{
NAME:
    "Move the given robot one step on the given direction"

DOCUMENTATION:
    "The plan attempts to move the given robot in the given direction for one step, with the
validation,
        - To ensure that the direction to advance does not contain an obstacle.
```

```
        - To ensure that the robot does not go out of the boundaries of the grid."

GOAL:
    ACHIEVE move $robot $direction;

BODY:
    RETRIEVE MAX_X $MAX_X;
    RETRIEVE MAX_Y $MAX_Y;

    // Move Robot A
    WHEN: TEST(== $robot "robotA") {
        RETRIEVE robotA $x $y;

        // Check and ensure that the right side does not contain an obstacle and is within
the boundaries
        // The same applies for the following validations
        WHEN: TEST(&& (== $direction "right") (!(FACT obstacle (+ $x 1) $y)) (<= (+ $x 1)
$MAX_X)) {
            EXECUTE println "Robot A: Going right";
            UPDATE (robotA) (robotA (+ $x 1) $y);
        };

        WHEN: TEST(&& (== $direction "left") (!(FACT obstacle (- $x 1) $y)) (>= (- $x 1) 0))
{
            EXECUTE println "Robot A: Going left";
            UPDATE (robotA) (robotA (- $x 1) $y);
        };

        WHEN: TEST(&& (== $direction "up") (!(FACT obstacle $x (+ $y 1))) (<= (+ $y 1)
$MAX_Y)) {
            EXECUTE println "Robot A: Going up";
            UPDATE (robotA) (robotA $x (+ $y 1));
        };

        WHEN: TEST(&& (== $direction "down") (!(FACT obstacle $x (- $y 1))) (>= (- $y 1) 0))
{
            EXECUTE println "Robot A: Going down";
            UPDATE (robotA) (robotA $x (- $y 1));
        };

        RETRIEVE robotA $x $y;
        EXECUTE println "Robot A: (" $x ", " $y ")";
    };

    // Move Robot B
    // Please refer to the section above as the logic is identical but applicable for Robot B
    WHEN: TEST(== $robot "robotB") {
        RETRIEVE robotB $x $y;

        WHEN: TEST(&& (== $direction "right") (!(FACT obstacle (+ $x 1) $y)) (<= (+ $x 1)
$MAX_X)) {
```

```
            EXECUTE println "Robot B: Going right";
            UPDATE (robotB) (robotB (+ $x 1) $y);
        };

        WHEN: TEST(&& (== $direction "left") (!(FACT obstacle (- $x 1) $y)) (>= (- $x 1) 0))
{
            EXECUTE println "Robot B: Going left";
            UPDATE (robotB) (robotB (- $x 1) $y);
        };

        WHEN: TEST(&& (== $direction "up") (!(FACT obstacle $x (+ $y 1))) (<= (+ $y 1)
$MAX_Y)) {
            EXECUTE println "Robot B: Going up";
            UPDATE (robotB) (robotB $x (+ $y 1));
        };

        WHEN: TEST(&& (== $direction "down") (!(FACT obstacle $x (- $y 1))) (>= (- $y 1) 0))
{
            EXECUTE println "Robot B: Going down";
            UPDATE (robotB) (robotB $x (- $y 1));
        };

        RETRIEVE robotB $x $y;
        EXECUTE println "Robot B: (" $x ", " $y ")";
    };

FAILURE:
    EXECUTE println "System: Failed to Move - " $robot;
}
/* END: PLAN move $robot $direction */


/* BEGIN: PLAN collect $robot */
PLAN:
{
NAME:
    "Check and collect dust at the location of the given robot"

DOCUMENTATION:
    "The plan checks and collects the dust at the location of the given robot.
        - The fact of the collected dust will be retracted as a method of communication
between the two robots."

GOAL:
    ACHIEVE collect $robot;

BODY:

    // Retrieve all the dusts from the world model (fact)
    RETRIEVEALL $FACTS dust $x $y;
```

```
    // Check the location of Robot A with the location of the dusts
    // If the robot is at the location of a dust, collect the dust and retract the fact
    WHEN: TEST(== $robot "robotA") {
        RETRIEVE robotA $rX $rY;
        WHILE: NEXTFACT $FACTS dust $x $y {
            WHEN: TEST(&& (== $x $rX) (== $y $rY)) {
                RETRACT dust $x $y;
                EXECUTE println "Robot A: Dust Collected at (" $x ", " $y ")";
            };
        };
    };

    WHEN: TEST(== $robot "robotB") {
        RETRIEVE robotB $rX $rY;
        WHILE: NEXTFACT $FACTS dust $x $y {
            WHEN: TEST(&& (== $x $rX) (== $y $rY)) {
                RETRACT dust $x $y;
                EXECUTE println "Robot B: Dust Collected at (" $x ", " $y ")";
            };
        };
    };

FAILURE:
    EXECUTE println "System: Failed to Collect Dust - " $robot;
}
/* END: PLAN collect $robot */


/* BEGIN: PLAN store $robot */
PLAN:
{
NAME:
    "Store the visited location of the given robot"

DOCUMENTATION:
    "The plan stores the location visited by the given robot in the world model (fact)"

GOAL:
    ACHIEVE store $robot;

BODY:

    // Check if the location was previously added to the visited List
    // If it is not, then add it to the visited list
    WHEN: TEST(== $robot "robotA") {
        RETRIEVE robotA $x $y;
        WHEN: TEST(!(FACT visited $x $y)) {
            ASSERT visited $x $y;
        };
    };
```

```
    WHEN: TEST(== $robot "robotB") {
        RETRIEVE robotB $x $y;
        WHEN: TEST(!(FACT visited $x $y)) {
            ASSERT visited $x $y;
        };
    };

FAILURE:
    EXECUTE println "System: Failed to Store Visited Location - " $robot;
}
/* END: PLAN store $robot */


/* BEGIN: PLAN check */
PLAN:
{
NAME:
    "Check whether if all the dusts were cleaned"

DOCUMENTATION:
    "Check and update if all the dusts were cleaned in the world model (fact)."

GOAL:
    ACHIEVE check;

BODY:

    // Go through the list of dusts
    ASSIGN $dustCount 0;
    RETRIEVEALL $FACTS dust $x $y;
    WHILE: NEXTFACT $FACTS dust $x $y {
        ASSIGN $dustCount (+ $dustCount 1);
    };

    // If the dust list is empty, update the dustCleaned fact to true
    WHEN: TEST(<= $dustCount 0) {
        UPDATE (dustCleaned) (dustCleaned "true");
    };

FAILURE:
    EXECUTE println "System: Failed to Check If the Dusts were Cleaned";
}
/* END: PLAN check */
```