YSC2229 Introductory Data Structures and Algorithms

Final Project Report

Vroomba Programming

Author: Zhu Tianrui

Team Members: Li Wandan, Guo Ziting

Acknowledgements
This is a group project done in collaboration with Linda and Ziting. The tasks were split as recommended. My contributions are: RoomGenerator.ml, RoomChecker.ml, and RoomUtil.ml.

Important Concepts

In our implementation of the room, we define the following terms:

Board
The board is a n x n square within which the room is drawn.

Room
The room is the space where Vroomba can move and clean. The boundaries of the room form a polygon.

Tile
A tile is a 1x1 square in the room.

pos
A type we designed to represent the nature of a point on the board. An "Edge" *pos* means that the point is sitting on the edge of the room polygon. "Inner" means that the point is within the room polygon but not sitting on the edge. "Outer" means that the point is outside of the room polygon.

Coordinates
The x, y location of a point of (int * int) type.

Relative Coordinates/ Map indices
In the Room datatype, we need to convert the coordinate to its corresponding array indices in order to retrieve its *pos* in the room.map array. The conversion is necessitated by the fact that coordinates can be negative but array indices are not.

Room Design

The room data type we designed has three attributes. First, room.edges contains a list of integer pairs that correspond to the corners of the room polygon. room.map is an array of array that stores the *pos* of relative coordinates. To help with the conversion from coordinates to relative coordinates, we have an attribute room.shift which stores the values to be subtracted from the coordinates to get the map indices.

The advantage of this design is that we are able to access instantaneously both the room polygon and the nature of each point on the board. We can benefit from the flexibility of geometry and the simplicity of a discrete data type.

Below is an example of how coordinates on a board are converted to *pos* in room.map.

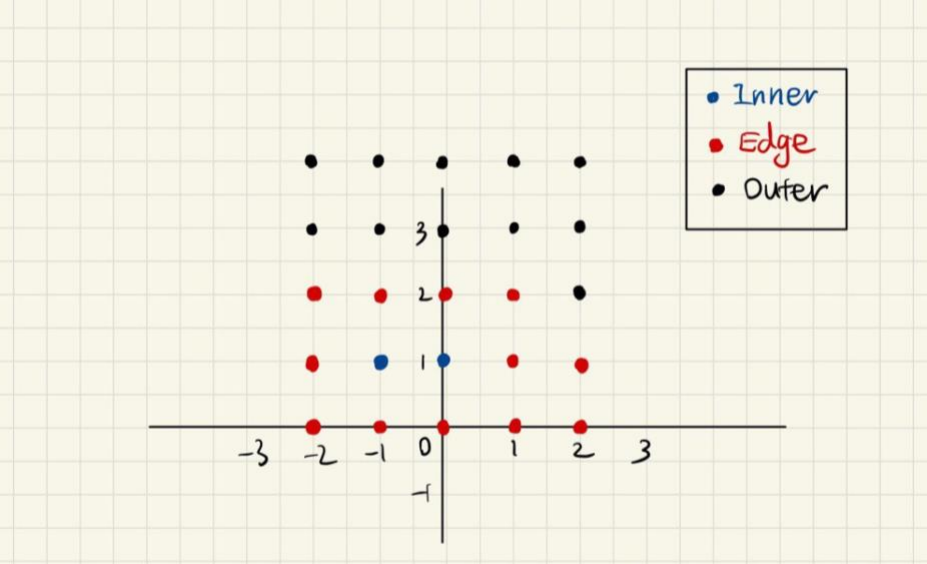

Figure 1. A board containing room polygon (-2, 0) (-2, 2) (1, 2) (1, 1) (2, 1) (2, 0).

The dots are the coordinates' corresponding pos in room.map

The major difficulty we faced was the conversion from polygon to room. We designed two functions for this. First, fill_edges fills room.map with "Edge" from the corner points on the room polygon and the points on the polygon's edges. This function rejects any diagonal edges. Next, with the help of point_within_polygon function from the lecture notes, fill_room fills in "Inner" where the points are within the polygon, and "Outer" where they are outside of the polygon. On the other hand, the room_to_polygon function is very simple given our implementation of the room. We only have to convert room.edges to a list of Point pairs.

To test the polygon_to_room and polygon_to_room functions, we convert a list of polygons into rooms and convert them back to polygons and check for equality.

Room Validity

A valid room is defined as a polygon with no lacunas, no obstacles and no diagonal edges. There must be a 1x1 space at (0, 0), which is the initial Vroomba position. Additionally, non-adjacent edges cannot intersect or be collinear. The figure below shows examples of invalid rooms.

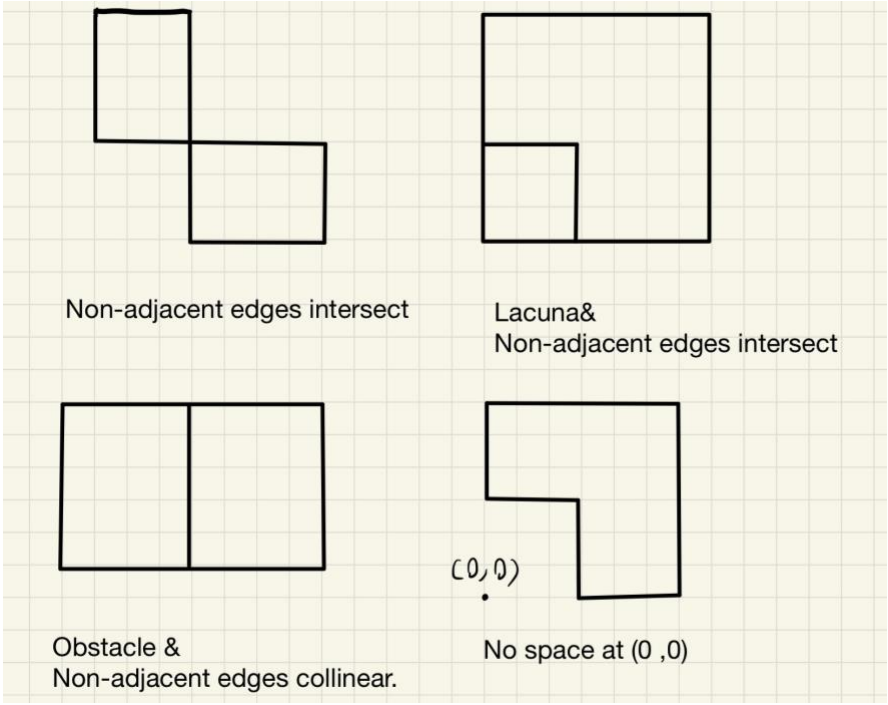

Figure 2 Invalid Rooms

In our implementation of the room data type, the room.edges attribute is a list of (int * int) coordinates which correspond to the room polygon's corner points. Due to this design, it is impossible to draw a lacuna or obstacle without making some non-adjacent edges of the room polygon intersect or overlap. Hence, the check for room validity boils down to two steps:

1. Check that non-adjacent edges do not intersect or overlap.
2. Check that (0, 0) is the lower-left corner of a tile in the room.

Room Generator

To generate a valid room, we first draw a n x n square cantered at (0, 0) to represent a board of size n x n. Then we divide the board into four square quadrants. The sides of each quadrant are of length n / 2, or (n /2) + 1 in the case of odd n. The idea is to walk a point randomly across the four quadrants till it returns to the starting position. We will keep track of all turnings along the path and produce a polygon from these turning points.

To ensure that the generated room is valid, we impose some rules on the movement of the point within the quadrants.

General Rules:

- The point is given a random direction and random number of steps each time.

- The maximum number of steps allowed in a direction is constrained by the point's distance to the boundaries specific to each quadrant.
- If the previous move is Up, the next can't be Down, etc. Basically, it cannot "cancel" the previous move.

Quadrant-specific rules:

- Each quadrant enclosed by the x-axis, y-axis and the boundaries of the board. One of the axis will be the starting axis where the point enters into the quadrant. The other axis is the end axis where the point finishes its movements in the quadrant.
- To avoid collinear edges and invalid edge intersections, the point's first move in a quadrant has to be away from the starting axis, and it is not allowed to touch the starting axis again.
- A set of boundaries and allowed directions are specified for each quadrant. See the figure below.

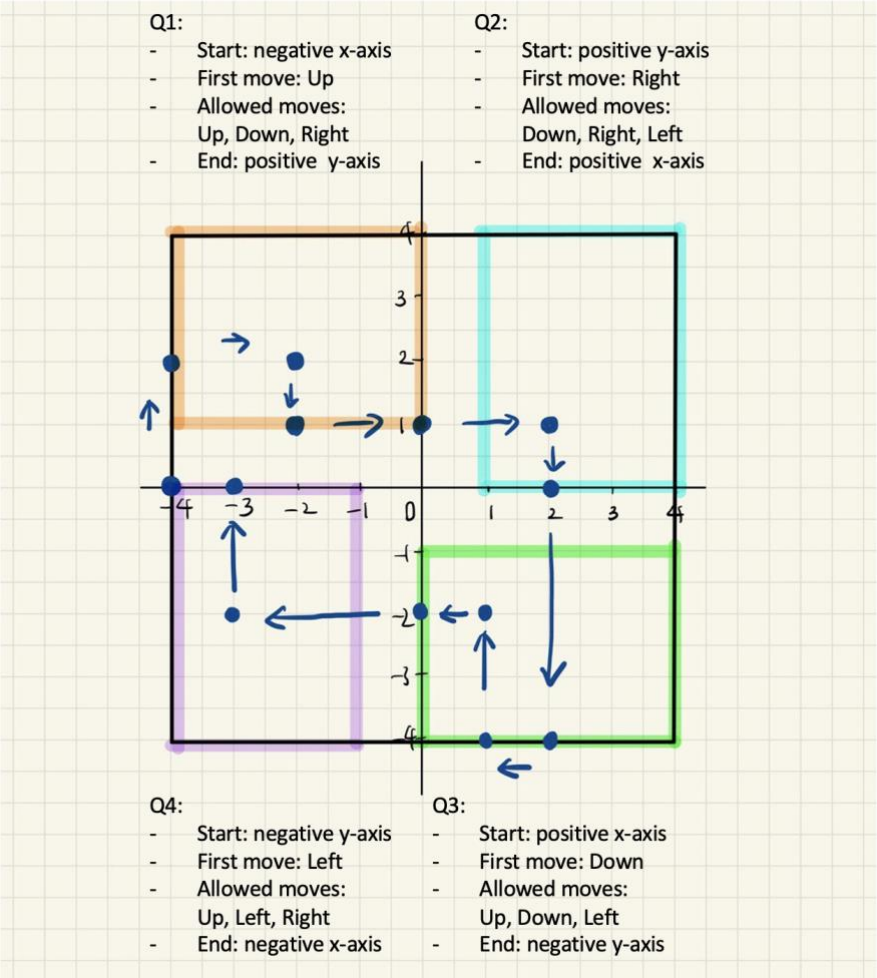

Figure 3 Generating a room

At the end of the random walks, we obtain a polygon from all the turning points. We then shift (0, 0) to a randomly chosen tile in the polygon. Finally, the shifted polygon is processed by polygon_to_room and returns a room.

Solution Checker

The solution checker checks if Vroomba follows a given string of moves, starting from (0, 0), all tiles of the room will be cleaned. At the heart of the checker is the "state" data type. It stores three things: Vroomba's current position, a hash table where the keys are tile coordinates and values are the hygiene status of the tiles, and a dirty tile counter to keep track of tiles yet to be cleaned.

A valid solution should clean all room tiles without stepping out of the room boundary. At each move, Vroomba will clean the tile it is on, and those among the eight neighboring tiles that are reachable. When cleaning a tile, the hash table `state.table` will be updated with the tile's latest hygiene status. If the counter `room.dirty_tiles` is 0 after going through all the moves, the solution is proven valid.

There are two important checks in this procedure: checking whether a coordinate denotes a tile and checking whether a neighbor tile is reachable from Vroomba's current position.

A coordinate certainly denotes a tile if its corresponding *pos* in `room.map` is "Inner". It cannot denote a tile if the pos is "Outer". If the pos is "Edge", we have to check the three other corners of the tile. Presence of one or more "Outer" corners means the current point does not denote a tile. Presence of one ore more "Inner" corners means otherwise. When all three corners are "Edge", we check if the center of the square is within the larger room polygon. Below is an illustration of the tile check.

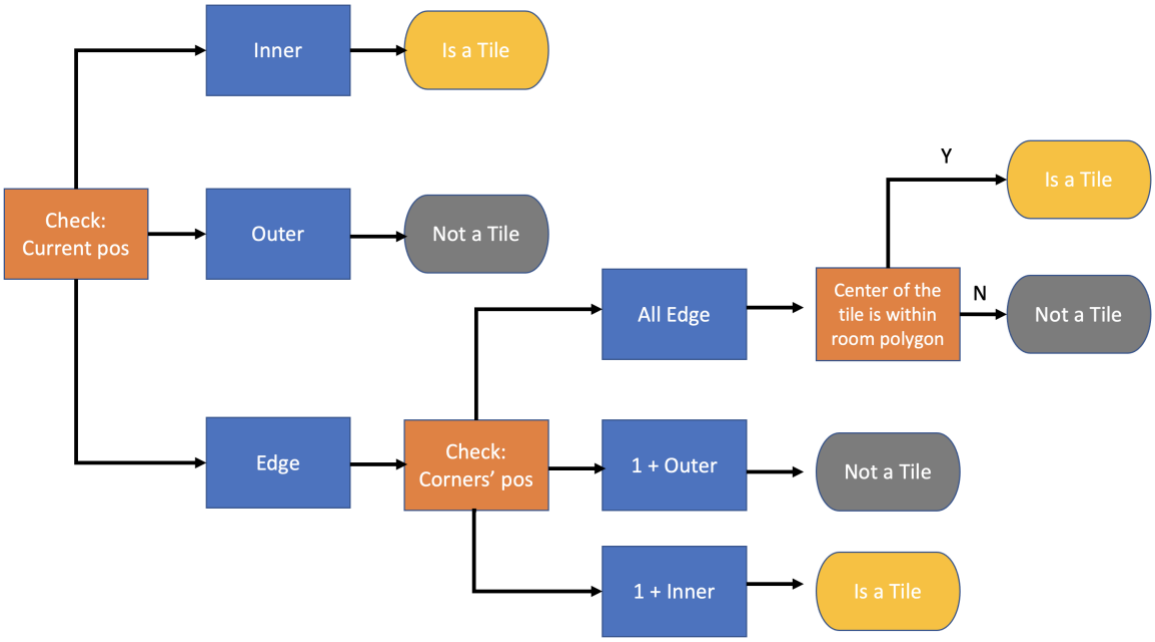

Figure 4 Tile check

Another important check is whether a neighbor tile is reachable from the current tile. First of all, the neighbor has to be a room tile to be reachable. Any of the four next-door neighbours are always reachable. A diagonal tile is reachable only if its two neighbours in the same 2 x 2 square as the current tile are both room tiles.

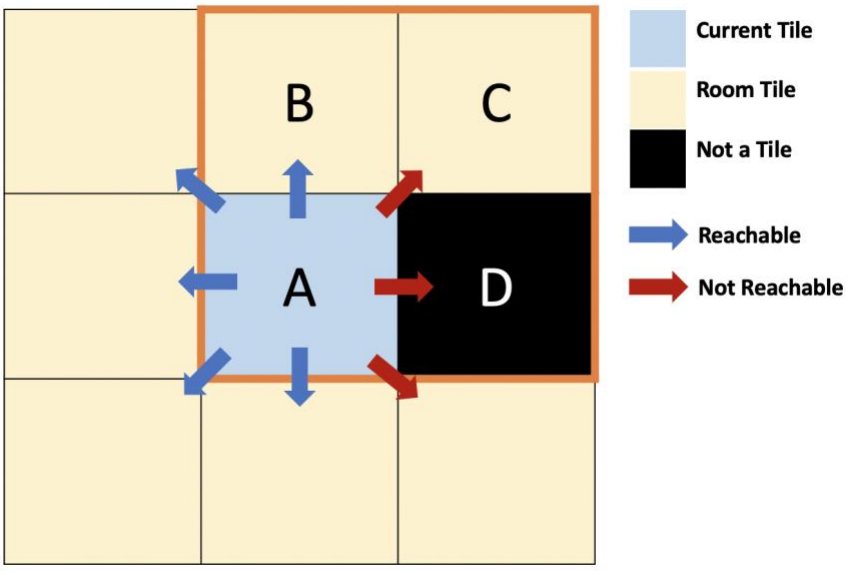

Figure 5. C is not reachable from A because D is not a tile

Room Solver

The solver aims to find a list, as short as possible, of moves that cleans the entire room. It first creates a binary search tree of the coordinates of all dirty room tiles. It keeps track of all the dirty tiles and deletes the newly cleaned ones from the tree. We also create a hashtable of the next node of all tiles and a graph representing the possible paths between tiles.

The solver first performs a depth-first search on the graph, moving Vroomba to the node with dirty neighbors. If we end up at a node with all clean neighbors, we just move to the root of the binary search tree with the Djikstra shortest path algorithm. The time complexity of the solver is therefore O ($|V^2 + E|$) due to the use of Djikstra algorithm.

The solver is tested with RoomChecker on randomly generated rooms.

Room Rendering

In the "play" mode, the user is given one or a series of rooms to solve. At the start of each session, the board, the room, and Vroomba will be drawn. In order to display rooms of different sizes pleasantly on the screen, we convert the coordinates to absolute coordinates by scaling the coordinates with the width of a room tile.

An important function here is `wait_until_q_pressed` which receives inputs form the user and move Vroomba in corresponding directions. Cleaning of the tile where vroomba is at and the neighboring tiles reachable from Vroomba is performed, and the graphics of Vroomba's position and the newly cleaned portion will be updated accordingly. When the user presses "q", the game ends.

The user-inputted solutions are written to an output file whenever a room is solved or when the game is closed.