Optimally Coverage of Unknown Indoor Environment for a Mobile Robot Using Android Application

Submitted by: Nadine Toledano, Ori Ziskind
Advisor: Majd Srour
Intelligent System Laboratory, CS, Technion
January 19, 2015

Abstract

This project objective is to develop an Android application which implements the STC on-line algorithm as developed and shown in [1]. Our project's goal was to implement an optimal navigation application of unknown bounded planar, on an Android device along with solving the problem of controlling a mobile robot in real-time. We achieved those challenges by utilization of the device's camera, for getting input on the neighborhood environment, and avoiding obstacles. As well, we used a IOIO platform, which is attached to the robot, to manage and control the robot movement. We produced an Android application thats implements the on-line STC algorithm, and communicate with the IOIO board, sending it movement commands accordingly.

1 Background

1.1 STC- Spanning Tree Covering

A paper by Yoav Gabriely and Elon Rimon, from the Department of Mechanical Engineering, Technion, Israel Institute of Technology, Israel [1]. This paper offers an algorithm for solving the problem of covering a continuous planar area by a square-shaped mobile robot. The algorithm, subdivides the work-area into disjoint cells corresponding to the square-shaped robot, then follows a spanning tree of the graph induced by the cells, while covering every point precisely once.

The paper suggests three versions of the STC algorithm:

- Off-line STC- Gets a geometrical description of a bounded planar environment as input.
- On-line STC- No prior knowledge about the environment, except that obstacles are stationary. The robot uses it's on-board sensors to detect obstacles.
- Ant-like STC- No apriori knowledge of the environment, but it may leave pheromone-like markers during the coverage process.

1.1.1 The square-shaped robot

A square of size D. It is allowed to move only in directions orthogonal to it's four sides, without rotating during it's motion.

1.1.2 Cell

The work-area is divided into square cells of size 2D (while D is as described in paragraph [1.1.1]). A cell can be free or contain obstacles. It is considered, that if a cell contain an obstacle, the robot cannot reach it.

1.1.3 Sub-Cell

Each cell is divided into 4 square sub-cells, of size D. This means that each sub-cell is precisely the size of the robot.

1.1.4 Spanning tree

As the work-area is divided to cells, each reachable cell will finally become a vertex of a spanning tree. The spanning tree represents the path for optimal coverage of the work-area. The edge between 2 vertices is passing from the center of the cells, while the robot is moving across this edges, covering the sub-cells.

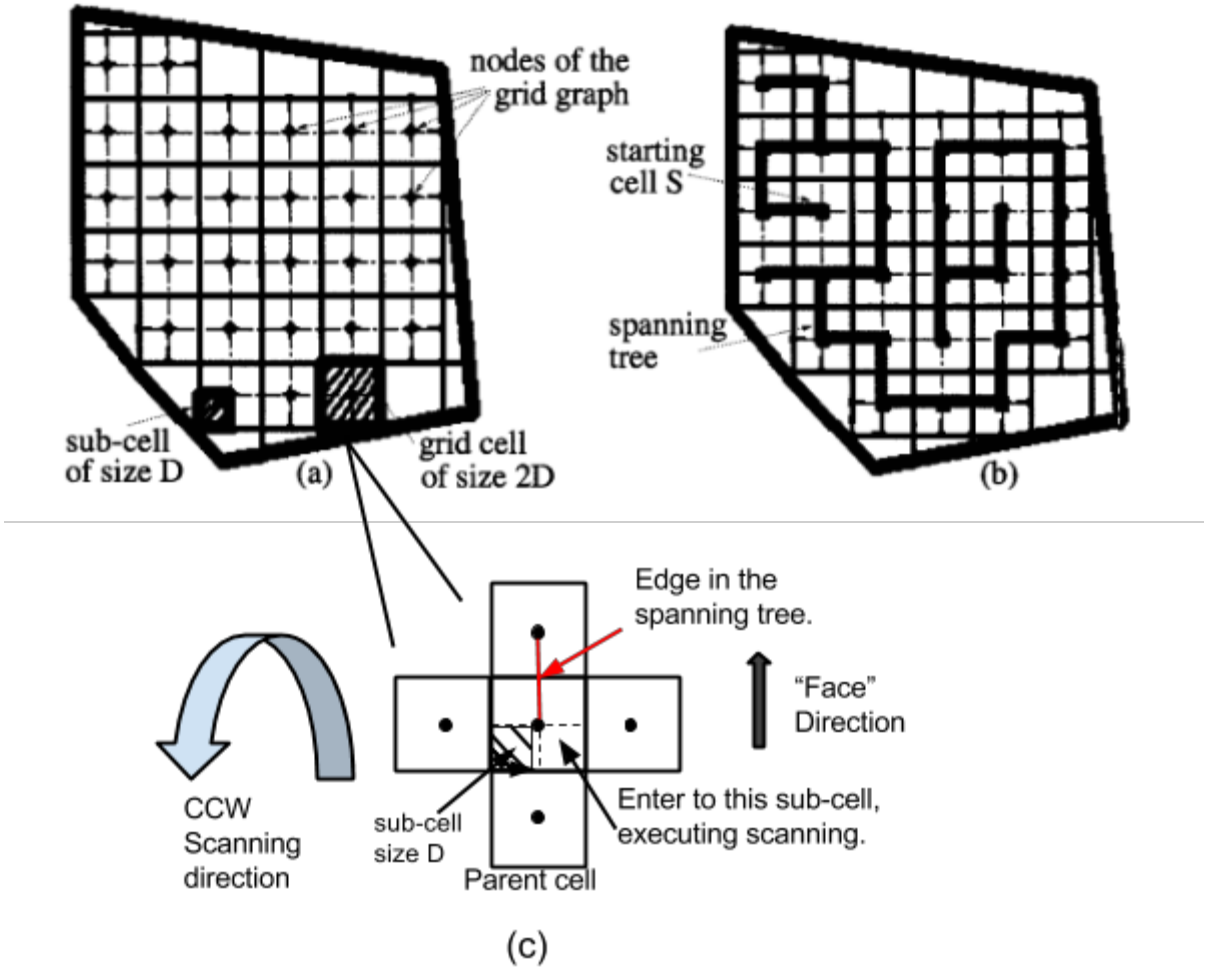


Figure 1: a) Grid approximation of a given work-area. b) A spanning tree of the grid. c) Zoom in on current cell and it's neighbours.

1.1.5 On-line STC Algorithm

Our project focused on the on-line version of the algorithm, where the robot uses its on-board sensors to detect obstacles and construct a spanning tree of the environment while covering the work-area. Given N- the number of cells in the work area, the on-line STC algorithm completes an optimal covering path in time O(N). However, it requires O(N) memory, and this requirement limits the size of work-areas that can be covered by the on-line algorithm.

1.2 IOIO Framework

The IOIO is a board specially designed to work with Android devices. The IOIO provides robust connectivity to Android devices via USB or Bluetooth connection and is fully controllable from within an Android application. We used the IOIO library to manage the Robot movement. The board is attached to the robot, and communicates with the application via Bluetooth [2].

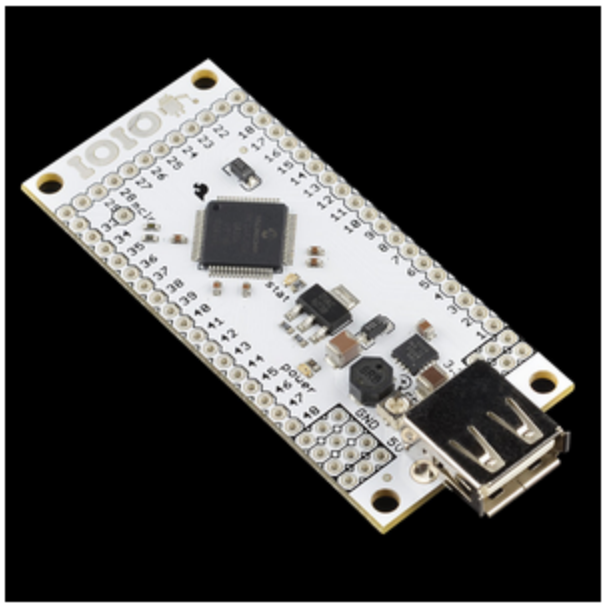


Figure 2: IOIO board

1.3 Obstacle Detection

Our application is using the Android device's camera, and a small convex mirror to take pictures of the surface around the robot.

2 The STC on-line algorithm

Goal: Given bounded surface, the task is to cover the entire surface, while moving in each reachable cell one time exactly, and avoiding obstacles.
Input: A starting free cell S.
Assumptions: Stationary obstacles, bounded surface.
Recursive function: A function STC(w, x), where x is the current cell and w the previous cell along the spanning tree.
Initialization: Call STC(Null, S), where S is the starting cell.
STC(w, x):

1. Mark the current cell x as an old cell.

2. While x has a new obstacle-free neighbor:
    2.1. Scan for first new neighbor of x in counterclockwise order, starting with parent cell w. Call this neighbor y.
    2.2. Construct a spanning-tree edge from x to y.
    2.3. Move to a sub-cell of y as described below.
    2.4. Execute STC(x, y).
3. If x = S, move back from x to a sub-cell of w as described below.
4. Return. (End of STC(w, x)).

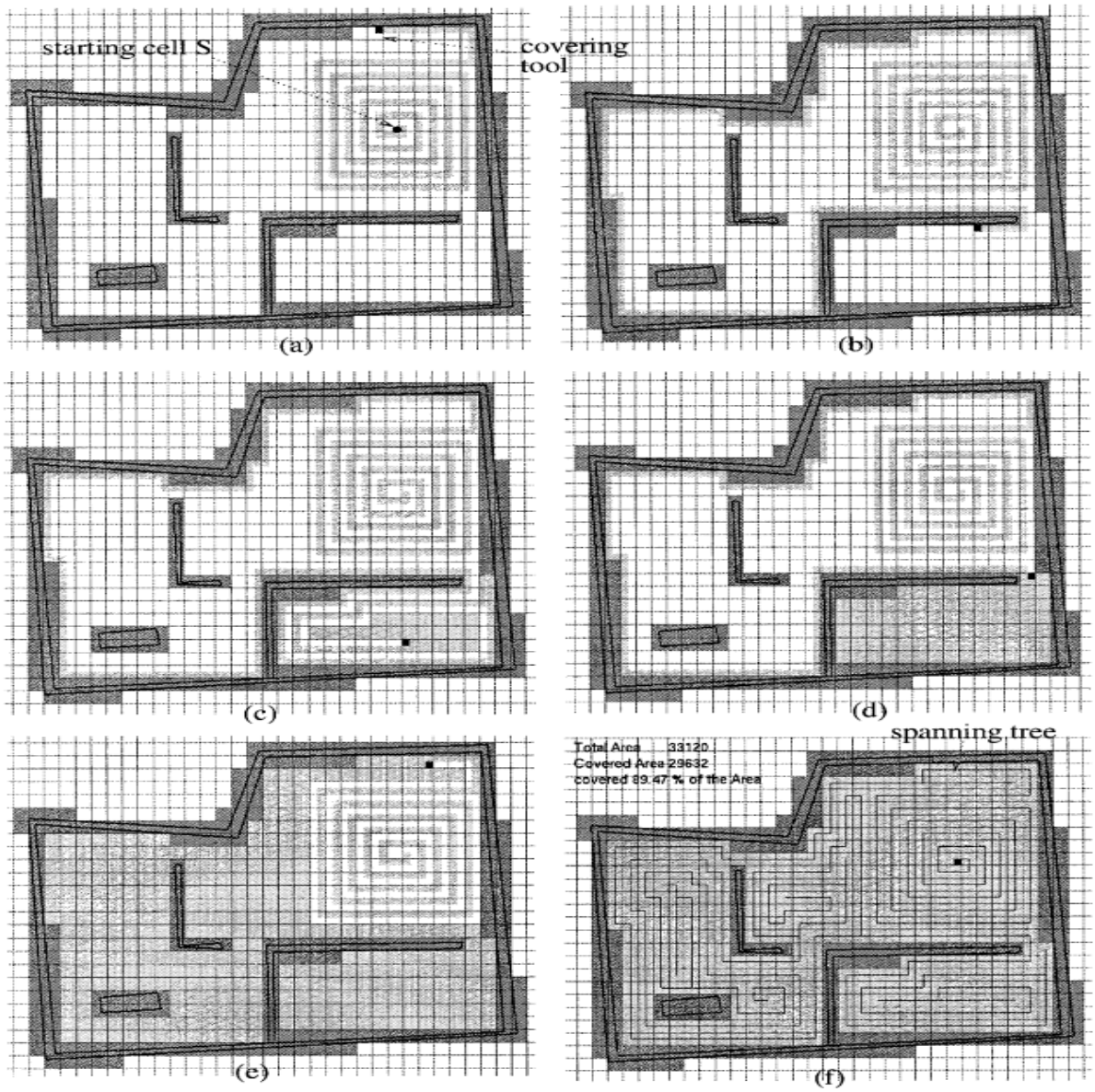


Figure 3: Six covering stages of the on-line STC algorithm [1]

3 Implementation

3.1 Assumptions

The following assumptions are taken:

- The surface is bounded.
- Starting cell is free.
- Obstacles are stationary.
- The robot contains IOIO board.
- The environment is well illuminated.
- The Robot can be attached with android mobile phone.

3.2 Design

3.2.1 Spanning Tree

The spanning tree is one of the basic structures of the algorithm. As we explained before, the spanning tree represent the path in which the robot is covering the surface.
Since the algorithm starts with no a priori knowledge about the surface, and need to store all of the covered surface information on the go.
As the application runs on a mobile phone, a smart selection of data structure was needed, in order to save runtime memory consumption. Consider all of the above, we decided to implement the tree using hash-table. That way, we can save only the amount of data needed. This enables the possibility to run the application on large areas. The cells are the vertices of the tree. The edges are represented by father coordinate information. This way, the robot can proceed with the surface detection and at the point it discovered all of a cell neighbours, he can go back to the father cell.
The hash table entry key is cell coordinate [3.2.2]. The value of each entry contains:

- Coordinates of the cell's father, i.e the cell who discovered it.
- Is the cell an obstacle.

Time Complexity: Each action of the table is O(1) on average as we used Java's HashMap.
Space Complexity: O(N) in memory, as N is the number of cells in the surface.

```
class SpanningTree {
        private HashMap<Integer, HashMap<Integer,Cell>> Table;
        private static SpanningTree _instance=null;
        protected SpanningTree() { /*c'tor , Initialization */ }
        public static SpanningTree getInstance(); /* Since it's singleton */
        public Cell putValue(Coord coord ,Coord fcoord,boolean isObst);
        public Cell getValue(Coord coord);
        public Coord getFatherCoord(Coord coord);
        public Coord getFatherCoord(Coord coord)
        }
```

3.2.2 Cell Coordination, Robot Orientation

Each cell is identified by its row and column numbers. The starting cell is selected to be at the coordinates of (0,0), the robot's front face direction is considered to be "Global North".
Each additional cell that is discovered, gets coordinates due to its location according to the starting cell and the global north.

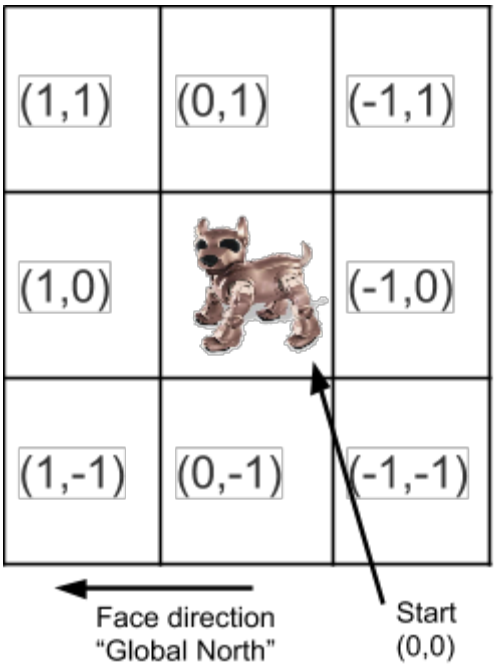


Figure 4: Cells coordination according to starting cell and "Global North".

3.3 Obstacle Detection

For achieving our Goal, to navigate with no apriori knowledge of the environment we use the Android device's front camera as a vision Sensor. The camera's frames are used for Obstacle Detection and avoidance. There are two main steps in Our obstacle detection algorithm steps, first step is the "Learning step" and the second step is "Thresholding and Detection".

3.3.1 "Learning" step

Performed Once, at the Starting cell. The camera's frame taken in the starting cell is used to learn what is considered "free". Under the assumptions that the starting cell is a free cell and we are navigating on a flat and uniform surface (in terms of texture and color).
Flow:

1. Get colored frame of the starting cell
2. Convert to grayscale
3. Sample the cell free floor with 7 different predetermined region and calculate the median color of each ROI.

These 7 color samples are the threshold values and remain constant throughout the run.

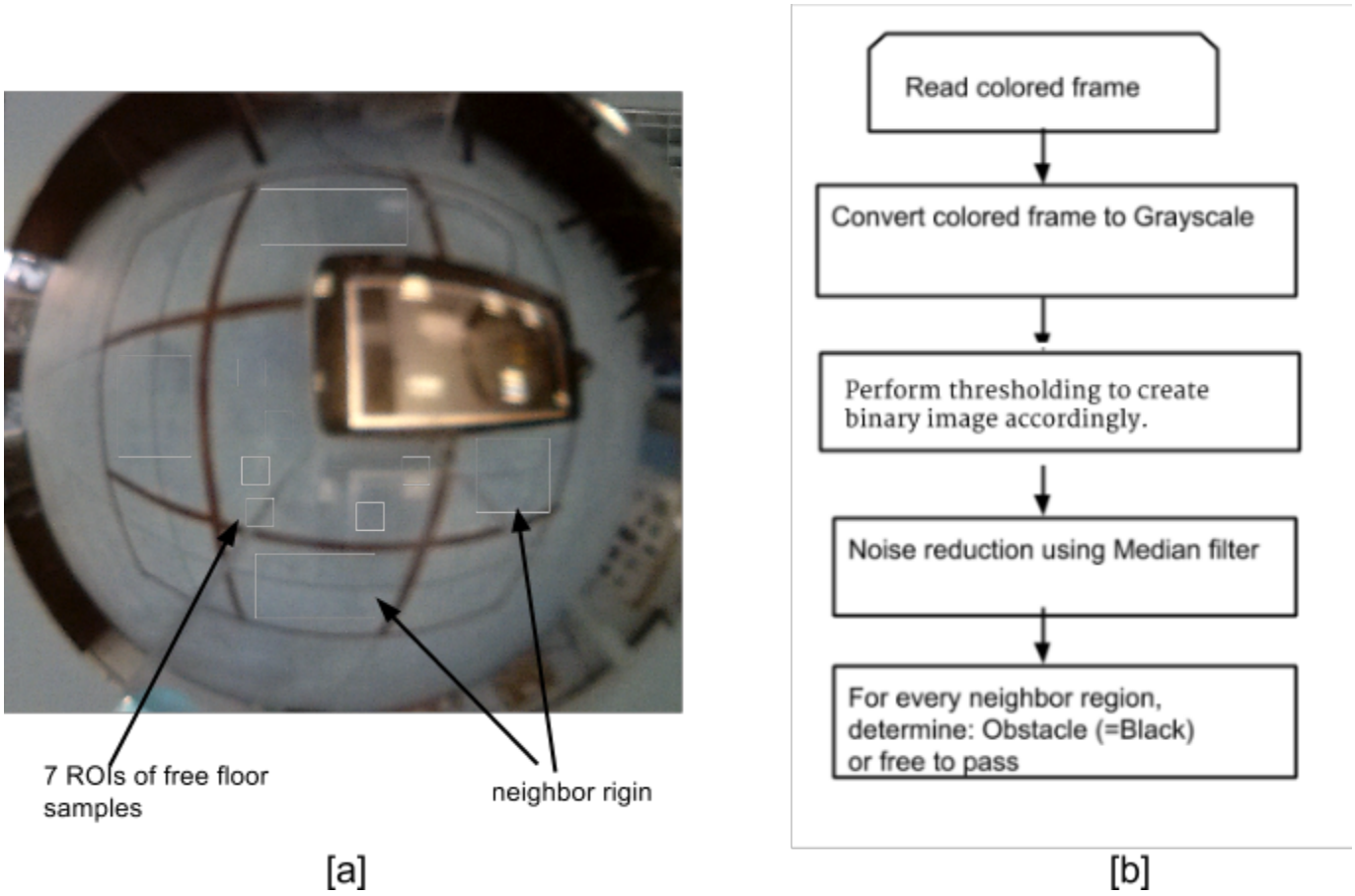


Figure 5: a) Frame capture in the starting cell. b) Flowchart for obstacle detection, when first arrive to a new cell

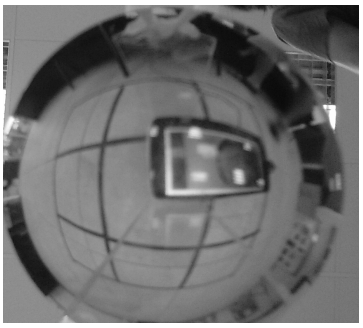
3.3.2 Thresholding and Detection

Performed each time the robot reaches to a new cell, "current" cell. Thresholding algorithm is applied on the current frame using the color samples from the former step. A restriction of this method of detecting obstacles is that an obstacle need to be in a different color from the navigation surface.

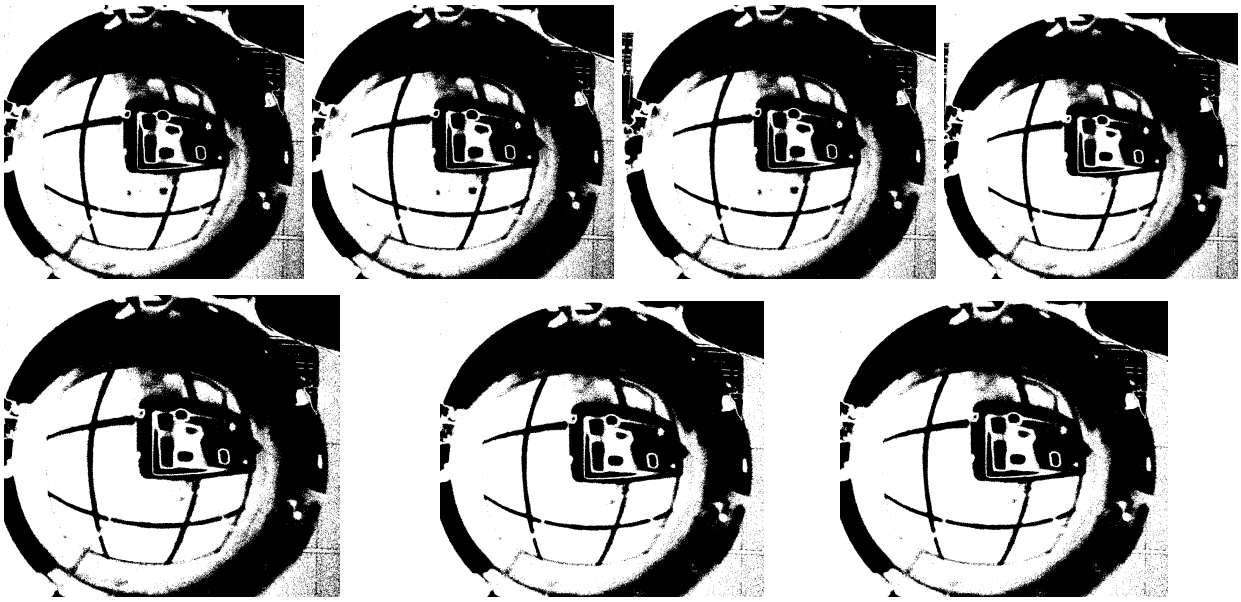
Flow:

1. Get colored frame and convert it to grayscale.
2. For each color sample perform thresholding to create a binary image.
3. Create one binary image which is the sum of the 7 binary images. Note, black (= obstacle) pixel has value 0, so pixel is black in the sum binary image only if it is black according to all the threshold values.
4. Perform median filter to enhance the binary image.
5. For every neighbor predetermined region:
   - 5.1. Calculate median value from the sum image.
   - 5.2. Median value is ~0, then neighbor is occupied (obstacle), otherwise neighbor is free.

Source:



Summing binary images, each is result of a different threshold value:



Sum binary image after Median filter (final result):

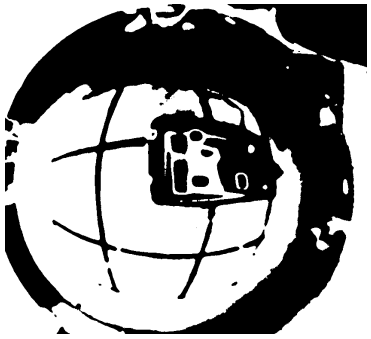


Figure 6: Example of thresholding and detection from source image to final result

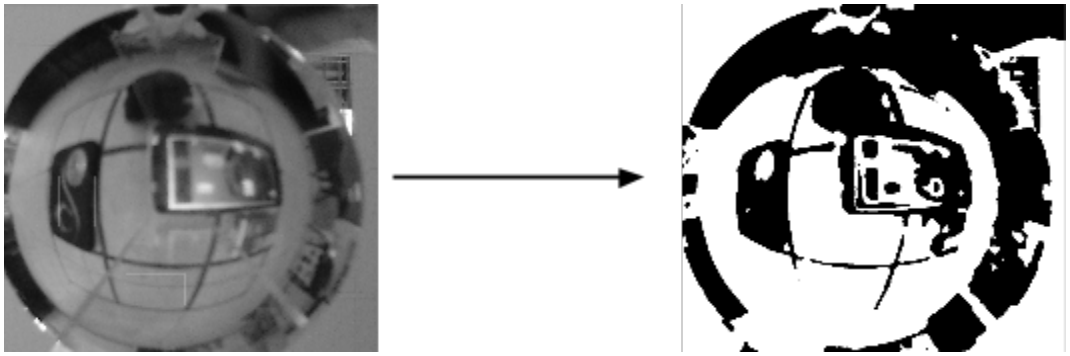


Figure 7: Example of thresholding and detection with obstacles

```
class Detector {
        private SpanningTree treeTable;
        private static Detector _instance = null;
        private Scalar[] _samples;
        private boolean[] _neighStatus;
        ...
        protected Detector(){
                treeTable = SpanningTree.getInstance();
                /* More initialization*/
        } /* c'tor */
        public static Detector getInstance(); /* Since it's singleton */
        /* Returns the direction to the first new neighbor Counterclockwise */
        public Direction getMoveDirection(Cell currCell, Orien faceOrien );
        /* called for every new frame in new cell, update the _neighStatus */
         public Mat process(Mat inputFrame);
        }
```

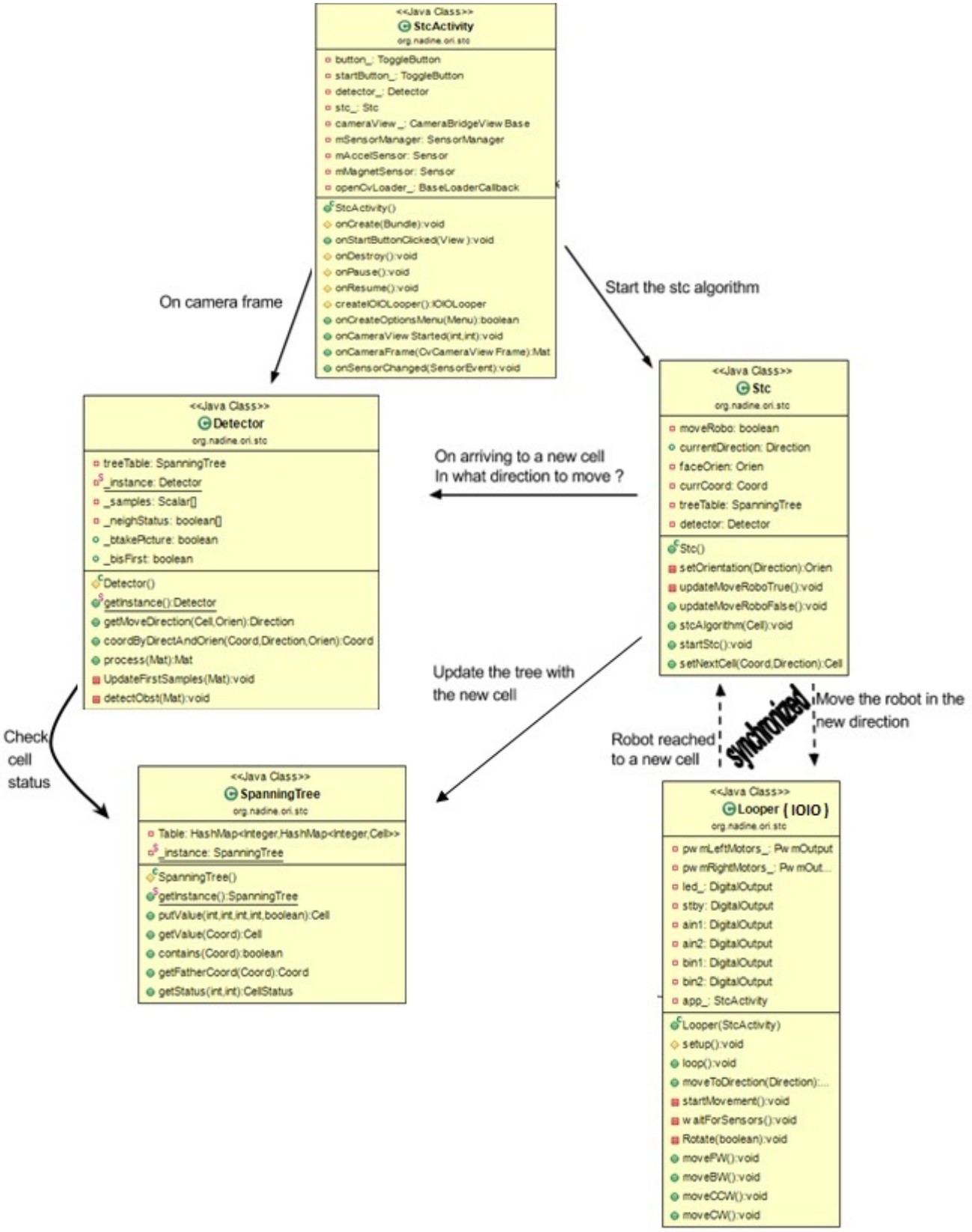
3.4 STC Flow

- Taking first camera frame at the free start cell.
- Calculating thresholds values by floor samples- "learning step" [3.3.1].
- Detecting neighbours status and update [3.3.2].
- While there is neighbor to visit / direction to move to:
    - Find next neighbour counterclockwise direction:
        - Check order: Right-> Up->Left->Down
        - For each direction: If free and not visited or is the father (we came from this cell): choose this direction.
          Else check next direction.

        - Return NONE if no direction is valid. (Notice in the end of the algorithm we visited the starting cell in the second time, all neighbours are visited and there is no father)
    - In case direction != NONE
        - Move the robot for the selected direction.
        - Update the spanning tree.
        - Take new frame from the new cell.
        - Detecting neighbours status and update [3.3.2].

```
class Stc {
        private SpanningTree treeTable;
        private Detector detector;
        private boolean moveRobo;
        . . .
        public Direction currentDirection; /* the direction to move the robot */
        public void startStc() {
                treeTable = SpanningTree.getInstance();
                detector = Detector.getInstance();
                /* More initialization */
                stcAlgorithm(p); //start the stc algorithem
                }
         /*run the on-line STC algo, recursive function */
        public void stcAlgorithm(Cell currCell);
        /* use to synchronise between the algorithm and the Robot movement controlled by IOIO*/
        public synchronized void updateMoveRoboFalse();
        private synchronized void updateMoveRoboTrue();
        }
```

3.5 Objects Interface Diagram



4 Test Class

In order to test the our algorithm implementation and data structures usage, we created a test object that gets surface input as a matrix. The matrix is built from int values represent cell status (free or occupied). The information on the neighbour status for the algorithm came from the test object according to the matrix values. Each time the algorithm discovered new cell, the test object updated the matrix with step number. That way, in the end of the algorithm we received the matrix surface with progress of the path.

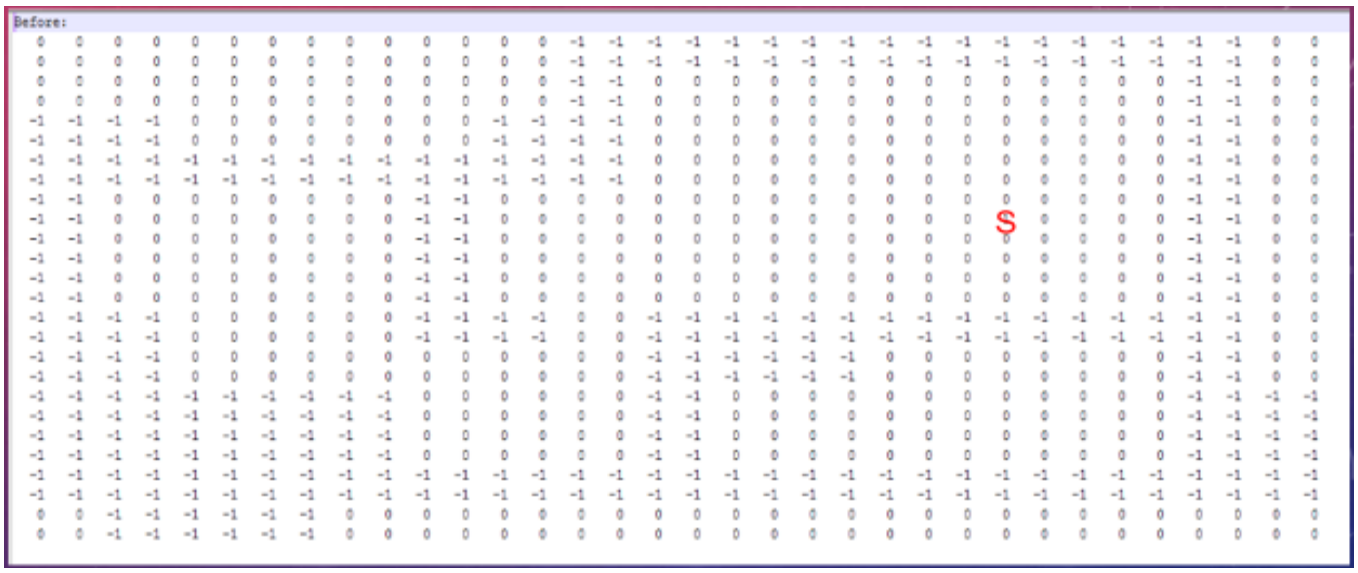


Figure 8.a: Test Class input representing a surface. -1 indicates obstacles, S is starting cell.

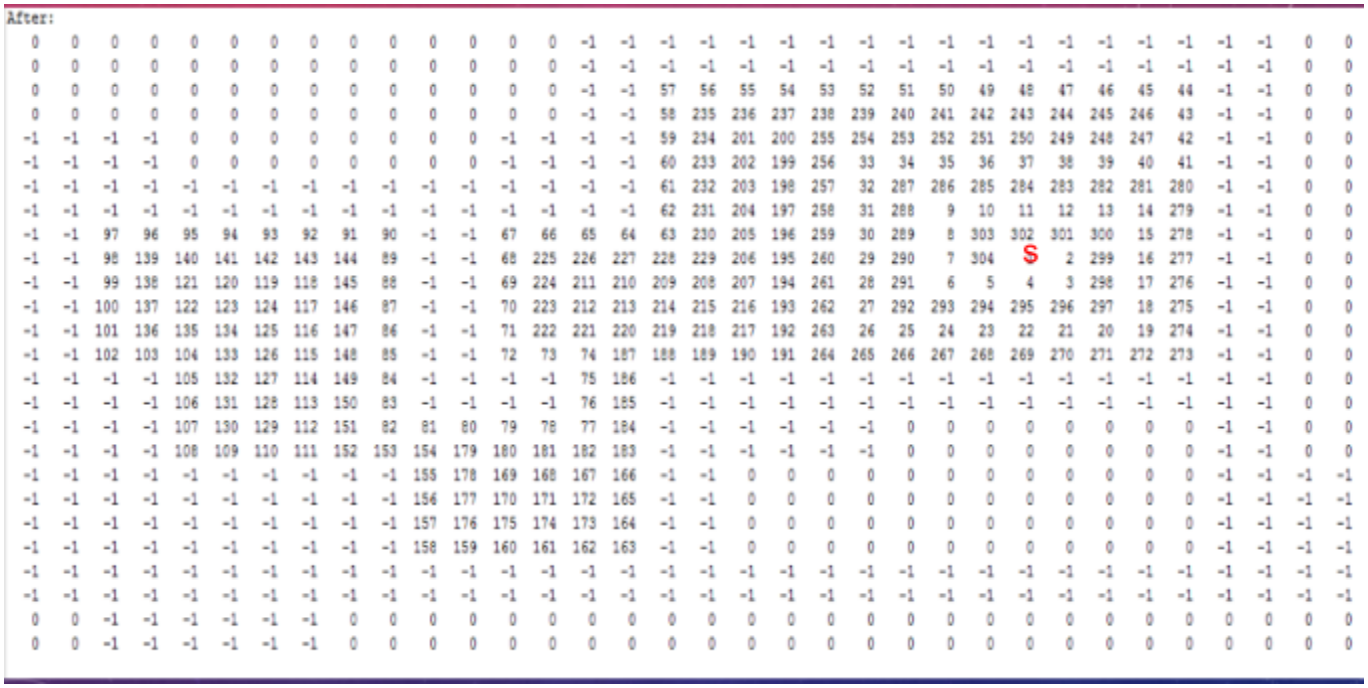


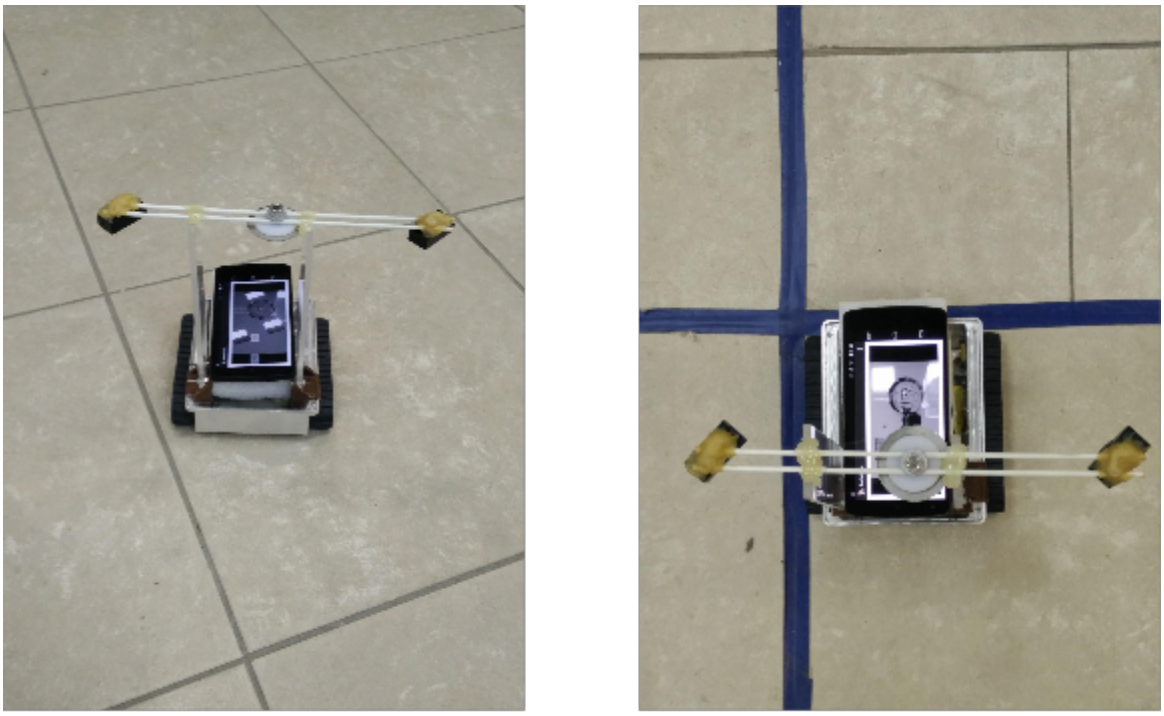
Figure 8.b: Test Class output, showing the path the algorithm created by steps.

5 Platform Demo

5.1 The robot

The robot we worked with contains:

- 2 motors which are connected to caterpillars.
- IOIO board- enable connectivity with Android devices via Bluetooth
- Convex mirror- for getting 360 degrees image of the environment surrounding the robot, used for obstacles detection.
- Encoders- converting movement into digital and analog signals.
- The Robot can be attached with a mobile phone.



a b

Figure 9: a) The robot and Android device we worked with. b) A look on the frame of taking a picture of the convex mirror, to get a vision of the robot's area.

5.2 Mobile Phone

For our work, we used Google Nexus 5.

In order to run the app, a mobile device must have the following prerequisites:

- Camera- taking pictures of the convex mirror on the robot, and getting constant vision on the robot's environment, used for obstacle detection.
- Bluetooth- establishing connection with the IOIO board on the robot.
- Acceleration and Magnetic sensors- used for controlling the robot's movement.

6 Challenges

6.1 Robot Structure

6.1.1 Mirror Distance

As mentioned before, the application uses the phone's camera in order to take pictures of a convex mirror and by that gaining information on the neighbours of the current cell. The mirror on the robot we worked with, is lifted up on plastic holders. The phone is attached to the robot and due to this, the distance between the camera and the mirror became quite large. This affected the region we could identify using this method as well as the possibility for good image processing, as the neighbour cells were only a small portion of the frame. Trying to zoom in reduced the quality of the image since the mobile phone doesn't have optical zoom. Zooming also could not solve the problem of small region. In order to solve it, we built an elevation to the phone, to get the camera physically closer to the mirror.

6.1.2 Identifying Free Cells Through Plastic Holders

The plastic holders mentioned in the former paragraph, got in the frame of the camera, as part of the region of the left and right neighbours. This cause to detect free cells as obstacles. We solved that by adding samples of the plastic holders in the "learning step" of the obstacle detection [3.3.1].

6.2 Controlling The Robot Movement

The main problem of the movement was not accurate enough steps. The algorithm requires the robot for exact steps, as each sub-cell is the size of the robot. Taking uneven steps can get lack of coordination between the robot physical location and where the application thinks the robot is. The atomic steps expected from the robot are 90 degrees rotation to both sides, and forward movement of a sub cell. Here follows, several methods to control the robot movement.

6.2.1 Motor Control Using Time

At first, we tried to control the time the motors are running. We measured the time it takes the robot to make each atomic step, and hard-coded these times as part of the motors control. This was not sufficient due to battery power lose. In addition, the caterpillars tension got different results of rotating the robot, even with the same motors power and running time.

6.2.2 Encoders

Next, we tried to use the encoders of the robot to measure movement. This encoders converts movement into digital signals. Analysis of the signals also could not make a progress, since the encoders are using the motors spins. The caterpillars tension got the spin number to be inconsistent, which made the signals inconsistent and we were unable to rely on it. After consulting with experts we abandoned this approach realizing our platform is not stable enough.

6.2.3 Phone Sensors

Eventually, we tried to use the fact that the phone is located physically on the robot, to use it's sensors for movement. This solved the rotation problem. Using the magnetic and acceleration sensors we could get a change in the phone's orientation angle- and we used it to determine when we got a 90 degrees rotation and need to stop the motors. This left us with the problem of

getting atomic step forward. The robot length is relatively short (10 cm). This took GPS off the table. Trying to use the pedometer wasn't successful as it requires vibrations of the phone. Any other sensor we tried to use also didn't had the ability to determine such a small movement change. This is an open issue we have with the given robot.

7 Summary and Future Work

The relevant background we had for this project was the Object Oriented Programing course. We started with no knowledge about Android development, image processing or robotics. As the paper implies, we gained knowledge and experience in those fields.
As shown in this paper, navigation with no a-priori data about the surface can be done in real time even on modern day mobile phones. Obstacle detection can be done using camera and image processing, allowing the surface to be with no actual limitations. As we showed, there is no real limit even on the size of the surface, other than runtime memory. Our work produced android application, ready to be used as a platform for connecting to a robot and executing the algorithm. The reliance on the IOIO device enables wide possibilities of robot platforms, as the IOIO is easy to get and assemble.

For future work we recommend:

- Working with a reliable robot that produce unchanged atomic steps.

Run instructions:

- After installing the application, choose the AndroidSTC icon:

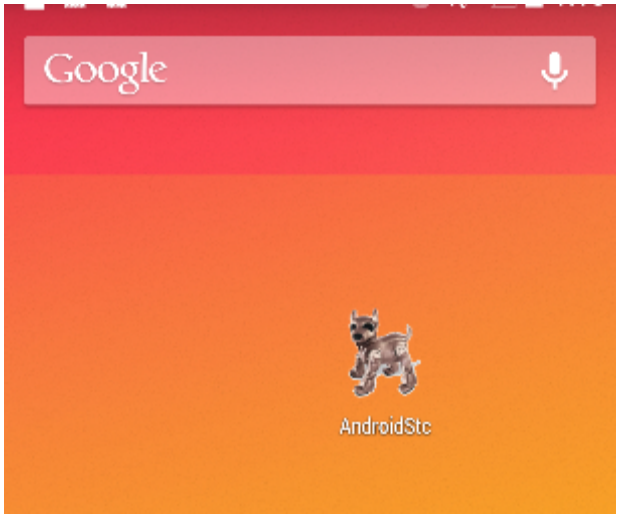


Figure 10: The application icon after installing.

- The main screen shows the image of the frontal camera, on top of it 2 buttons and 4 squares.
- The IOIO button is for checking connectivity with the IOIO board.
- Align the squares with the starting cell neighbours.
- Press the STC button to start the algorithm.

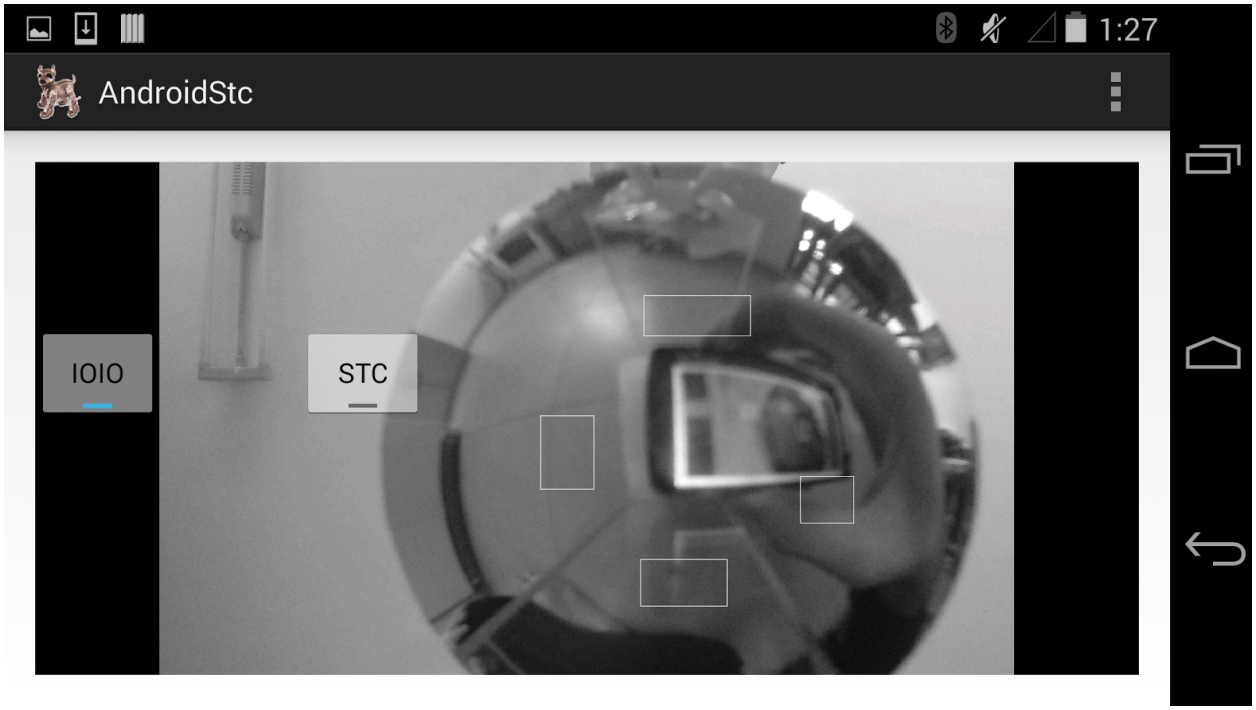


Figure 11: Application's main screen

References:

1. Yoav Gabriely and Elon Rimon, Spanning-tree based coverage of continuous areas by a mobile robot, Department of Mechanical Engineering, Technion, Israel Institute of Technology, Israel
2. Ytai Ben-Tsvi, IOIO for Android documentation, https://github.com/ytai/ioio/wiki
3. Basem M. ElHalawany, Hala M. Abdel-Kader, Adly TagEldeen, Alaa Eldeen, Sayed Ahmed, Vision-Based Obstacles Detection for a Mobile Robot, Shoubra Faculty of Engineering, Benha University, Cairo, Egypt.