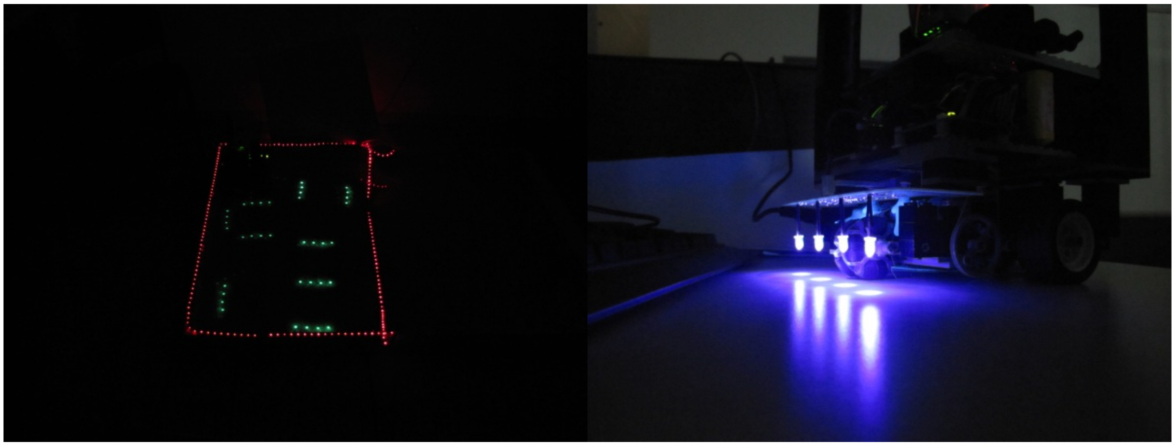
Optimally Covering an Unknown Environment with Ant-like STC algorithm



Project by:

Majd Srour

Anis Abboud

Under the supervision of:

Yotam Elor and Prof. Alfred M. Bruckstein

Technical support by:

Sergey Danielian

Table of Contents

- Abstract
- Introduction
- The Environment
- Task Description
- The Robot
- Image Transformation and Filtering
- Spanning-Tree based Coverage - Algorithm
  - The Ant-like STC algorithm
  - Marking Sub-cells
  - Finding the Parent cell
  - Identifying visited sub-cells
  - Finding the green dots
  - Angle Detection
  - Rotation and Calibration
- Obstacles Detection
- Precision
- Initial Calibration
  - Transformation Calibration
  - Neighbors Calibration
- Noise handling
- Limitations
- Video
- Results
- Conclusions and Suggestions
- Acknowledgment and References

Abstract

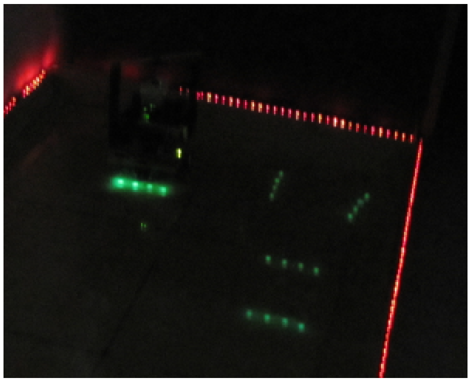
Robots plays key function in daily life tasks. Recently, it was shown that a single ant robot (modeled as finite state machine) can simulate the execution of any arbitrary Turing machine. This proved that a single ant robot, using pheromones, can execute arbitrarily complex single-robot algorithms. In this project, we used an algorithm that leaves Ant-like pheromosne on a surface in order to solve the covering problem.

Introduction

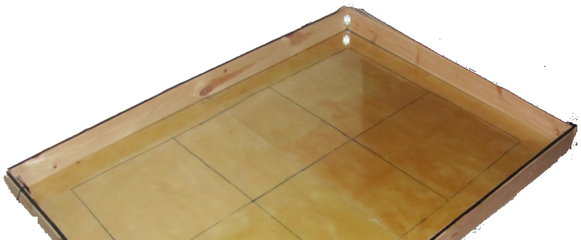
The mobile robot covering problem can be formulated as follows. Let a tool of a specific planar shape be attached to a mobile robot, and let A be a continuous planar work-area bounded by obstacles. Then the mobile robot has to move the tool along a path such that every point of A is covered by the tool along the path. The covering problem is currently receiving considerable attention for several reasons. First, sensor-based coverage by mobile robots seems amenable to the geometric planning techniques developed for the sensor-based robot navigation problem. A second reason is current interest in competitive algorithms for autonomous systems that operate with incomplete information of the area. Finally, recent work on ant-like coverage has spurred interest in the possibility of achieving complex coverage behavior by bounded-resource agents who leave pheromone-like markers in the environment. In the previous project, we've implemented the online version of the Spanning Tree Covering algorithm. In this project, we've implemented the Ant-Like version - which leaves pheromone-like markers in the environment in order to solve the problem.

The Environment

The Robot leaves pheromone-like markers in the environment in order to know the places it has already covered. In order to leave those pheromone-like markers, we used a phosphorescent glowing floor that being charged by the robot using an UV LED that leaves dots on the phosphorescent floor.

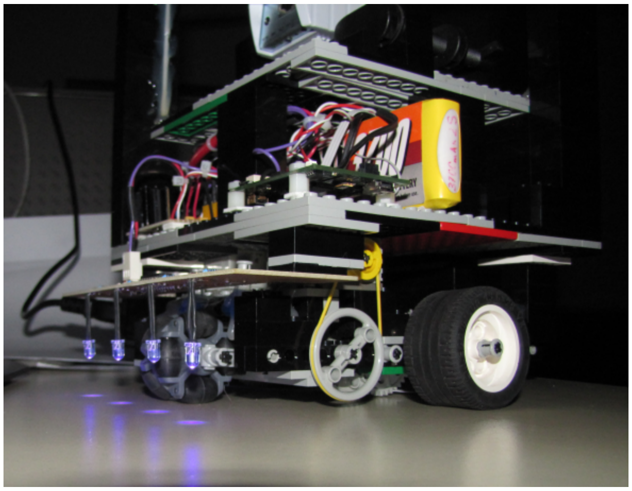


Task Description



Given a phosphorescent glowing floor, the robot divides the floor virtually into sub-cells of its size, and each 4 adjacent sub-cells are considered one cell.
The Robot's task is to cover the floor, passing exactly once over each *sub-cell* and to return to the same *cell* where it started.
A red light in a *cell* indicates that the *cell* is an obstacle.

The Robot



The Robot is built from *Mindstorms Lego*. Its major parts are:

- **Board:** *PIC-18* board.
- **Wireless Module:** *Xbee* wireless communication module.
- **Wheels and Motors:** 2 main wheels, motorized by two gear motors, and two auxiliary wheels that enable the Robot to perform an on-place 90 degree rotation.
- **Camera:** Wireless camera directed up.
- **Mirror:** A curved spherical mirror, directed down.
- **4 UV (Ultraviolet) LEDs:** Enable the robot to leave green tracks (dots) on the glowing floor.

The Robot is wireless-controlled by a computer program which runs the *STC Algorithm.*
The camera with the spherical mirror, enable the Robot to take a 360 degree panorama view of the adjacent area in a single shot, as shown below.

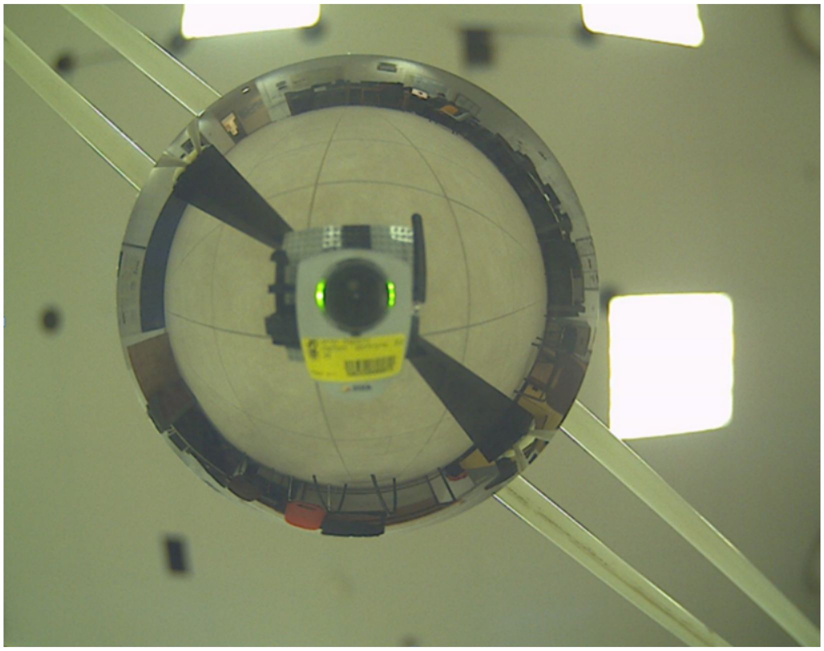
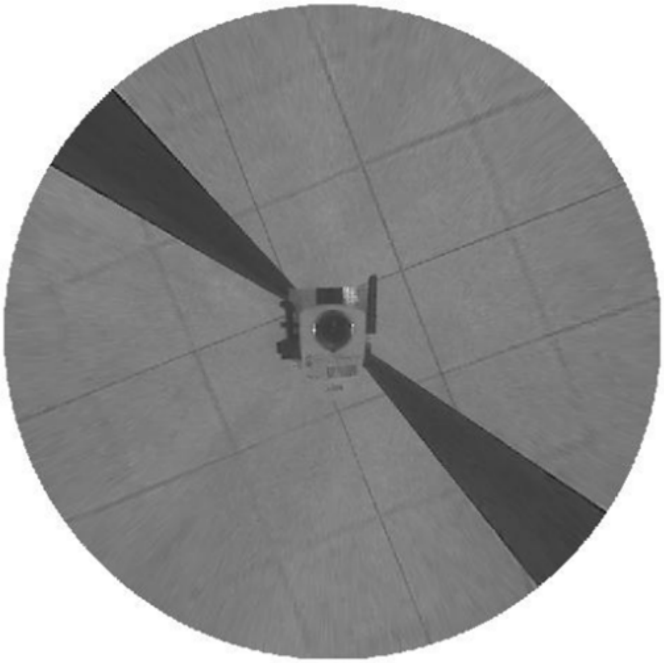
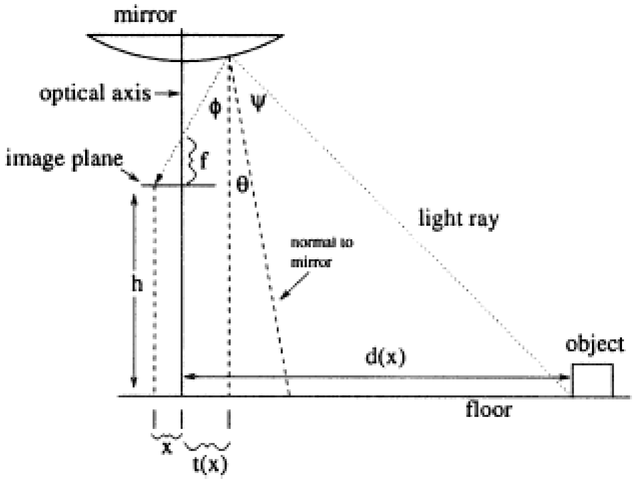


Image Transformation and Filtering

In order to work with the 360 degree panorama view, we apply a transformation, which produces an image like the one below.



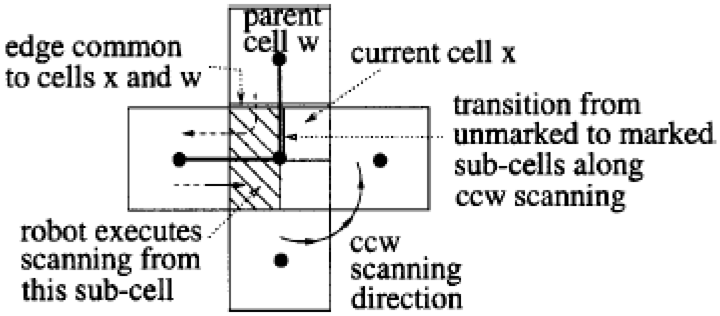
The main idea of the transformation is that if we look at a single pixel in the *curved* image, and compare it to its new coordinates in the *normal transformed* image, we see that the pixel moves on the axis that connects it with the camera lens. In other words, in both the *curved* picture and the *normal* picture, the pixel is in this axis, although in the curved picture the pixel is closer to the camera. So what we need to do is to calculate the ratio between its distances of the camera lens in the two pictures. How much should it move on this axis in order to get the *normal* picture, we did this by calculating the angle *Phi* and *Theta* as shown in the figure below.



after applying the transformation, we filter the image into two images, the obstacles image and the visited cells image. In the obstacle image, we keep all the pixels that the red color is bigger than a threshold. And in the visited cells, we keep the colors that the red colors is less than a threshold. Basically, this gives us two images, one for the green dots the robot left on the board, and other one, is the red dots of the obstacles.

Spanning-Tree based Coverage - Algorithm

The Robot uses the ant-like version of the STC Algorithm described in the paper "Spanning-tree based coverage of continuous areas by a mobile robot" by Yoav Gabriely and Elon Rimon (http://www.springerlink.com/content/n5770672r3852113/fulltext.pdf). A copy of the paper is attached in the appendix.
We assume that the robot occupies a square of size D.
We subdivide the work-area into square cells of size 2D, and assume that obstacles are red lights.
In this version of STC the robot has no prior knowledge about the environment, except that obstacles are stationary. Rather, the robot must use its on-board camera to detect obstacles and plan its covering path accordingly. We assume that the robot is capable of identifying obstacles in the four cells neighboring the robot’s current cell, as described in the following figure:



The Ant-like STC algorithm

The ant-like version of STC is also an on-line algorithm, and is similar to the algorithm we used for the first project. However, now the robot has the ability to leave markers in the *sub-cells* it covers. We utilize these markers to reduce the memory consumption to O(1) (Final-State-Machine) instead of saving the board in the computer, as we did in the previous project.

```
Initialization: Set x = S.
While S has unmarked sub-cells:
1.      Determine the parent cell of x, w = Parent(x).
2.      Determine a new neighbor of x, y = Neighbor(w, x)
3.      If y is not Null:
        3.1.    Move to a sub-cell of y, while marking the sub-cells of x being covered.
        3.2.    Set x = y.
4.      Else (y is Null):
        4.1.    Return from x to a sub-cell of w, while marking the sub-cells of x being covered.
        4.2.    Set x = w.
End of while loop.
```

Marking Sub-cells

Each time the robot decides to drive forward, it turns on the lights for a few seconds before proceeding, leaving a trace in one edge of the visited *sub-cell*.
Because the robot occupies a sub-cell, and his lights are on the back, the lights leave a trace only on the edge of the cell. And because we turn on the light only upon driving forward, exactly one edge of each visited *sub-cell* will be lit.
The reader may wonder here why didn't we keep the leds on during the robot's operation, leaving trace on every covered point. Turning the light on during drive doesn't leave a radiant enough traces. Instead, we have chosen to turn it on only on the edges, for a few seconds, to allow the surface to absorb the light.

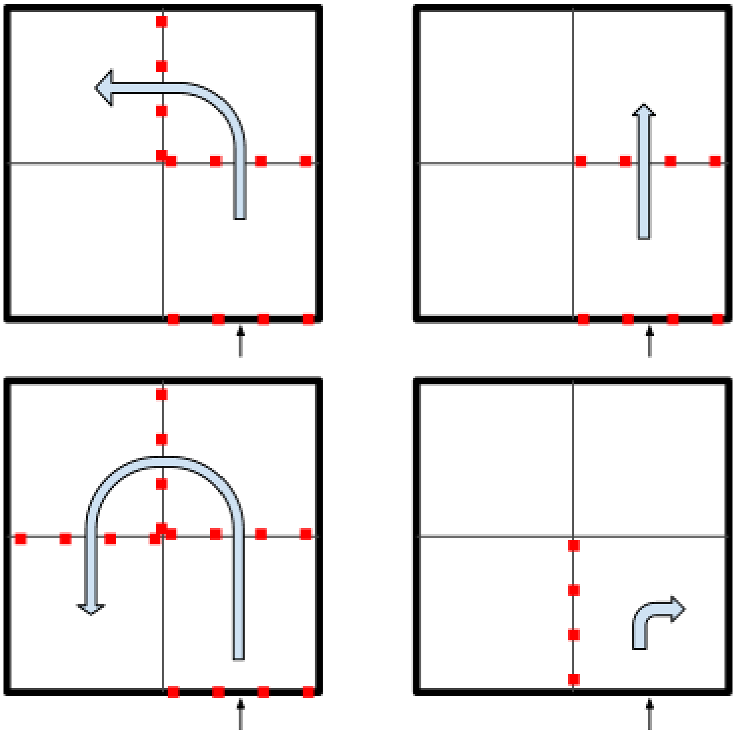
Finding the Parent cell

As long as the four sub-cells of the current cell are not all marked, the robot need only to scan the sub-cells in counterclockwise order and identify the transition between unmarked and marked sub-cells. The two sub-cells bordering this transition have an exterior edge in common. This edge is necessarily the boundary edge between the current cell and its parent cell in the spanning tree.

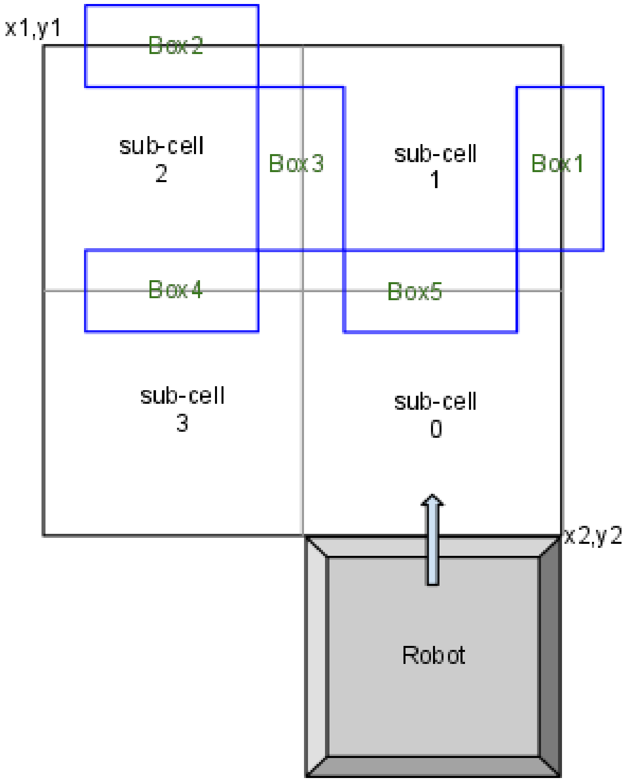
Upon entering a cell, the robot takes a photo of it, and finds its parent cell based on the lights inside it.
However, our robot marks only one edge of each visited sub-cell, and this edge is shared with the neighboring sub-cell, making it trickier to figure out if a sub-cell is visited.

Identifying visited sub-cells

Because the robot turns on the light only upon driving forward, its path in a new cell can be one of the following four:



The figure below shows a cell, divided into 4 sub-cells, and 5 edges of sub-cells are marked with boxes.
By examining all the possibilities where the robot is visiting a cell, we reached to the conclusion that only the 5 boxes below can be lit upon entering a cell.



If the robot hasn't entered this cell before, none of the 5 boxes above will be lit. This stems from the definition of the robot's path.

If the robot entered the cell from sub-cell 1 before:

- It could take right, lighting box 5 and exiting.
- It could move forward, lighting boxes 1 and 3.
- Or it could take left, lighting boxes 1, 3, and 4.

If the robot entered the cell from sub-cell 2 before:

- It could take right, lighting box 3 and exiting.
- Or it could move forward, lighting boxes 2 and 4.

If the robot entered the cell from sub-cell 3 before:

- It could only turn right, lighting box 4 and exiting.

We assume that sub-cell 0 was not visited before, because otherwise, the robot would not have chosen to enter it. And from all of the above, we can conclude that:

1. sub-cell 1 is visited iff one of the boxes 1 and 5 is lit.
2. sub-cell 2 is visited iff one of the boxes 2 and 3 is lit.
3. sub-cell 3 is visited iff box 4 is lit.

Finding the green dots

To find the green dots left by the robot as traces, we crop the relevant section of the image, and look for "dots" in it.

To find a dot, we look for the square of pixels (of a dot's size) that has the highest average value.

Looking for a square, instead of a single pixel allows ignoring noise.

If the maximum found value is under a threshold, then the sub-cell was not visited.
After finding a dot, we black out the square, and look for a new dot.

Angle Detection

Because of the battery decay, and the fact that the robot is built from Lego, a 90 degree turn, wasn't actually 90 degree, so we needed an algorithm to detect the current angle, and to fix the rotation in case it was not truly 90 degree rotation. In order to do that, we take the green transformation image, find the maximum green value (x0,y0), take a small area surrounding the max value we found, cover the dot we found (X0,y0) with black box, then find another max value, we do this 3 times. with this, we find the newest 3 dots the robot left. we pass a line between those 3 lines (as accurate as possible). And the slope of this line basically is the angle we are seeking for.

Rotation and Calibration

For the robot to rotate 90 degrees, we send it a rotate command, specifying the time, T, we want it to spend in rotation. We initially set T to a time that let's the robot do the first rotation accurate enough.
After each rotation (including the first), we take an image of the area, and using the angle detection algorithm described above, we detect the robot's deviation. If the robot has done the rotation accurate enough, no calibration is needed. Otherwise, let d be the angle that the robot actually rotated in (calculated using the angle detection algorithm). Now we know that in the next time, we should give the robot 90/d * T time to do the rotation, because in the time slice T, it rotated d degrees, so for 90, it needs a time slice of 90/d * T. So we reassign T to get the new value and send the robot another command in order to let it complete the 90 degrees rotation it was supposed to perform.

Obstacles Detection

To distinguish between visited cells, that are marked with a green line, and obstacles, the latter were defined as red-lights. If a cell contains a red light, it's regarded as an obstacle.
Because the transformation works with a grayscale image, after taking a photo, we:

- Filtered out the red light, and ran the transformation on the green image, identifying visited cells or sub-cells.
- Filtered out the green light, and ran the transformation on the red image, identifying obstacles.

Precision

One of the challenging changes between the previous project and this one, was the precision issue. In the previous project, we only needed to identify whether or not a *cell* is an obstacle.
In this project, if a cell is not an obstacle cell, we need to identify visited and free *sub-cells*.
A sub-cell's area is only fourth of a cell's area. Moreover, we have markers only on the edges of the sub-cells.
Thus, small inaccuracies stemming from the camera, the environment, a low light from the

outside, or a deviation of the robot, caused the robot to make wrong decisions.
One way we used to debug the robot's decisions was saving a copy of each taken snapshot, with a timestamp on it. Thus, we were later able to track the robot's path and spot the issues.

To improve the accuracy, we added a two-step initial calibration.

Initial Calibration

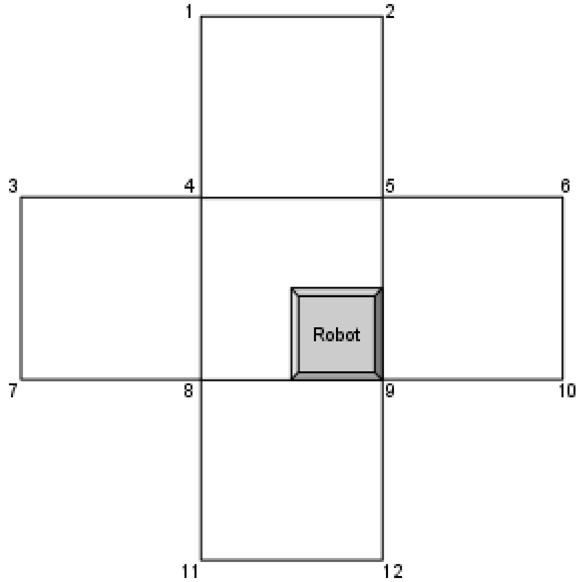
Because the robot is built from LEGO, the parts can easily move and the transformation can be easily distorted. That's why we added a calibration logic, which runs optionally in the beginning.

Transformation Calibration

Initially, the application asks the user to keep the lights on, and helps him calibrate the robot on the lab floor, where we can spot whether the transformation parameters are set correctly.
The application shows the user an image of the floor, and asks him to select the center of the camera lens. The user is shown a transformed image, and is asked if it looks good. If the answer was yes, the coordinates are serialized to a file, allowing the transformation to use them, and saving them for future runs where we might want to use the old configuration.

Neighbors Calibration

The camera, and the transformation, are not perfect, and the image is not perfectly symmetric. Thus, after the transformation is calibrated, the user is shown a transformed image, and is asked to click on the 12 corners of the 5 relevant cells in the following order:



This step increases precision and makes sure that the robot is sampling the correct part of the image when checking the neighboring cells.
After that, the points are also serialized to a file, for future runs.

Noise handling

Because the robot works in the dark, and the image is black except for some green and red dots, the static noise of the camera could have a significant effect.
To reduce this noise, we took a photo in complete darkness, with no light, and deducted this photo from every taken snapshot, thus reducing the camera static noise.

Limitations

The web-camera

After lighting a dot on the floor for a few seconds, the light could be still seen by humans until after 10 minutes. However, it could be seen by the camera we used only in the first minute or so.
The camera we used, was not intended to work with low-light conditions, and it couldn't identify the green light dots after less than a minute.

The system can be improved if we use different technique for identifying the green lights. For example, using a better camera, or using other types of sensors. To our opinion, a camera with a fully controlled integration time will be the best solution. Improving the floor itself or using more powerful LEDs to charge it won't help much since the light is visible to the human eye for tens of minutes as is.

Video

www.youtu.be/ZkVvyI5Qmww

Note: Unfortunately, due to the limitations described above, the web camera we used could not identify the light traces after less than a minute. Thus, it was not possible to run the robot on a large board.

Results

Using a simple and cheap robot with limited sensing and computational capabilities, we succeeded in solving the covering problem with the Ant-like approach, where the robot leaves pheromones-like markers on the surface.

Conclusions and Suggestions

This project comes as a proof that simple robots with limited computational capabilities, can solve complex problems by leaving pheromones like markers on the surface.
Because of the camera limitation, we couldn't try the algorithm on a bigger surface, we

suggest for a future work, to try using a better camera, preferably camera with a fully controlled integration time and trying the algorithm on a bugger surface. Also, because the algorithm is modeled as a Finite-State-Machine, all the algorithm can be implemented and ran on the robot itself, without using an external computer.

Acknowledgment and References

1. http://en.wikipedia.org/wiki/Ant_robotics
2. http://www.springerlink.com/content/n5770672r3852113/fulltext.pdf