Fault-tolerant algorithm for a mobile robot solving a maze

Article · January 2013

CITATIONS
3

READS
5,020

1 author:

Adam Srebro
Warsaw University of Technology
3 PUBLICATIONS **14** CITATIONS

SEE PROFILE

Fault-tolerant algorithm for a mobile robot solving a maze

Adam Srebro

Warsaw University of Technology, Warsaw, Poland

Maze solving problem has been considered since the ancient times. In robotics community this problem is widely known from the "All Japan Micromouse Competition". In this paper the author has extended the maze solving problem of cases when some robot distance sensors have been damaged. A new fault-tolerant algorithm for a mobile robot solving a maze is proposed. A dedicated simulator has been designed and written in Java to compare several maze solving algorithms for different combination of sensors failures.

Key words: fault-tolerant, flood fill, maze solving, mobile robot, simulator.

Introduction

The maze is a symbol linked to the old Greek myth that spoke of the Athenian hero Theseus, who descended into a dangerous labirynth at Crete to fight with Minotaur and save his people from a horrific end [16]. Maze symbol has been presented in the architecture, religion and music since the ancient times [6,16].

The concept of the Micromouse as a maze solving robot was firstly introduced by the IEEE Spectrum Magazine in 1977.

The first World Micromouse Competition was held in Japan (Tsukuba) in 1985. Nowadays the biggest micromouse competitions so called "All Japan Micromouse Competition", are held in Japan in two categories. The first is "Classic Micromouse" where robots solve a 256 (16x16) cells maze. A single cell consists of 18x18 cm base and 5cm high walls (1,2 cm thick). The second category is "Half-size Micromouse" introduced in 2009 where the maze has been expanded to 1024 (32x32) cells [8].

A micromouse is a autonomous mobile robot designed to solve the maze. A robot has to be able to explore the maze with goal searching to solve the maze. Some basic exploration methods and their modifications are known, however, none of them considers the possibility of robot distance sensors failures [1,10,15,17]. The correct information from the process of exploration is necessary for robot path planing. Typically, the shortest path is determined in terms of distance or time. For this purpose the most commonly used algorithm is Djikstra's algorithm or slightly faster Flood fill algorithm [10].

Fault-tolerant control systems have been developed, especially in aeroplanes [7] and industry [12]. In mobile robotics this problem is becoming an important factor because of the rapidly increasing number of sensors in the environment perception system. Although the algorithms and methods for determining the position or the path planning are still evolving there is a great need to ensure the safe operation of robots. Reflections on the failures of actuators can be found in article [9].

The case of partial degradation of absolute positioning system has been considered in the previous paper [14].

This article focuses on the problem of obstacle detection sensors failures and proposes an algorithm that increases the safety of the robot operation.

In this paper the author makes the following contributions:

- proposing fault-tolerant algorithm for a mobile robot solving a maze. This algorithm includes any variances of indexes of the damaged sensors and it can be applied in obstacles detection system equipped with any number of sensors,
- designing dedicated simulator in Java for micromouse robot operating in the presence of distance sensors faults.

The paper is organized as follows. Section 1 gives a brief overview of a few widely used maze mapping algorithms with goal searching. Section 2 describes an al-

gorithm for finding the shortest path from a mapped area of the maze. A new faulttolerant algorithm for a mobile robot solving a maze is presented in Sect. 3. Section 4 contains the simulations results for the failures of selected sensors. Finally, I conclude the paper in Sect. 5.

1. Maze mapping algorithms with goal searching

Mapping the maze is to memorize the location of the existing walls for each visited cell. To shorten the simulation time, I have assumed that the robot maps the maze until it reaches the goal. Time of the first run from the starting position to the goal is often called a search time. After reaching the goal by the robot, the shortest path to the starting point is determined. In the second run the robot moves along the shortest path from mapped maze cells.

In the examples the author consider the shortest distance, but it may not be the shortest travel time path. To determine it we must also enter a preferred direction of movement as a movement straight ahead and enter the weight coecients for the cells in which the robot can only turn. These values are usually determined experimentally for a given robot.

Figure 1: Indexing of the maze cells.

Algorithm 1: Right hand algorithm

```
Input : Measured values from sensors
Output : Control the direction of motion of the robot
1 while (goal == false ) do
2     if (turnright == true) then
3         rotation (- 1, 90) ;
4     else if (go forward == true) then
          // do nothing;
5     else if (turnleft == true) then
6         rotation (1, 90) ;
      else
7         rotation (- 1, 180) ;
      end
8     goForwardOneCell();
  end
```

1.1. The wall follower algorithms

The goal searching can be performed based on the naive algorithms that use the right or left hand rule, as shown in Figure 2.

However, these simple algorithms do not allow to solve mazes, where the central square (target) is not connected with the walls of the banks of the maze.

The Algorithm 1 concerns the case when we use the right hand rule. If the robot cannot turn right then in the next step the ability to drive straight ahead is checked. In the third step the possibility to turn left is checked and if none of these movements can be executed, the robot is returned. Similarly, works the left hand algorithm, but this time the sequence of checking the possibility of turning on the right and left has been reversed.

1.2. Flood fill algorithm

Flooding algorithm is based on the phenomenon of pouring the liquid on the plane which spreads evenly in all directions if there are no obstacles.We start flooding from the goal point (from the center of the maze). At the beginning we do not have any knowledge of the maze structure and we pour the maze as if it was an empty square. Figure 3a shows the second flooding step and Figure 3b step in which all maze was flooded. In the first step blocks neighbouring with four goal blocks are flooded in the horizontal and vertical directions. The values of blocks flooded in step 1 are set to 1 and it is the first level of flooding as in figure 3a. In other words, the current step of flooding inundated neighbouring blocks are blocks from the previous step.

Iterative process of increasing values of blocks flooded in the following steps is continued until all the maze is flooded.

Block diagram of the flooding algorithm for the entire maze is shown in Figure 4. For a single block with a given level of flooding (current step), flooding is reduced to check if we can pour four neighbouring blocks in a horizontal and vertical direction as shown in Figure 5.

Assuming that the cells of the maze are indexed from 0 to 255 as shown in Figure 1 we can easily find neighbouring blocks for all maze blocks. We flooding only those blocks that have not yet been flooded (with values 0) and without the walls that block the flow of liquid as shown in block diagram in Figure 5. If a neighbouring block meets these requirements then it is flooded.

It means that block receives value greater by one than the value of the block, from which it was flooded and its index is stored in the array and checked in the next step

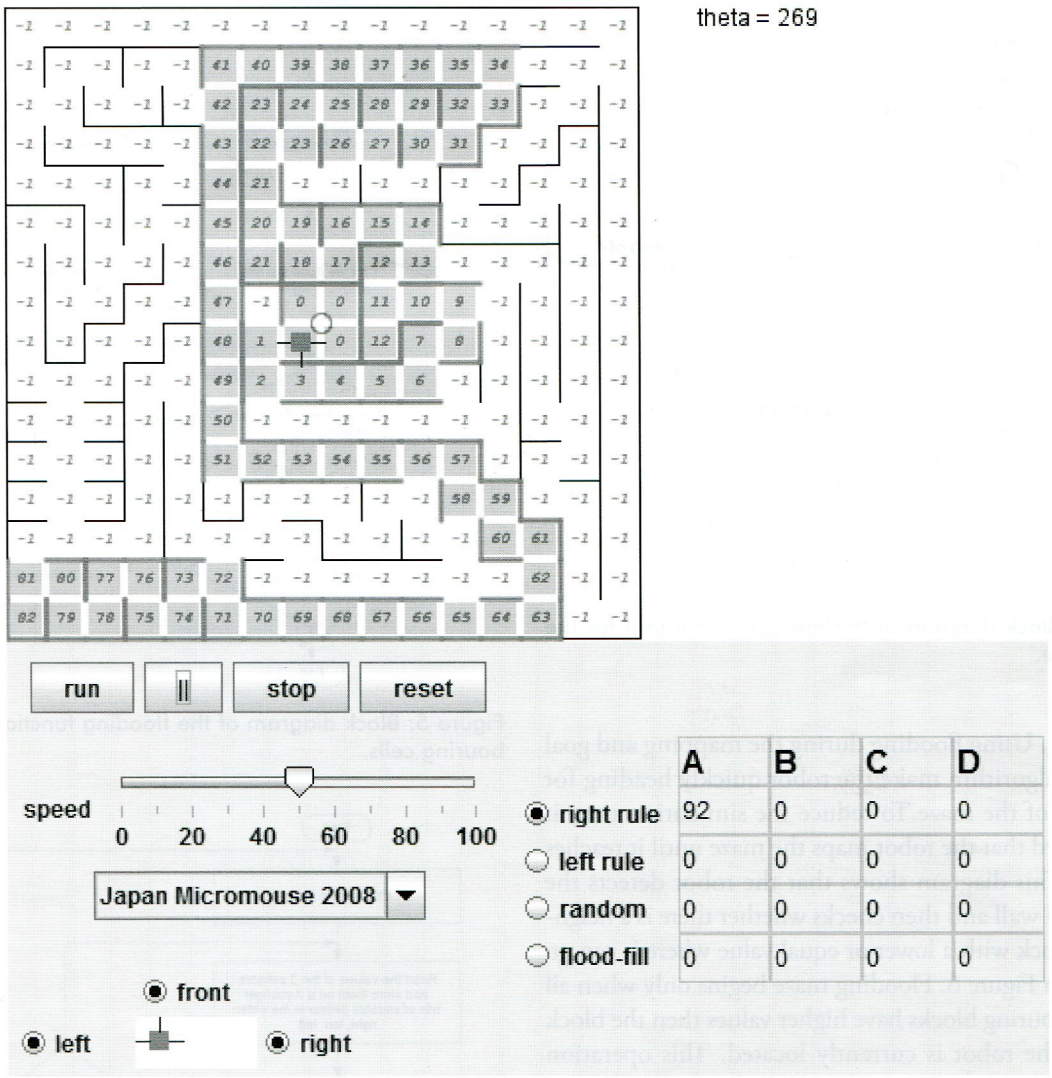

Figure 2: Maze solving using the right hand algorithm.

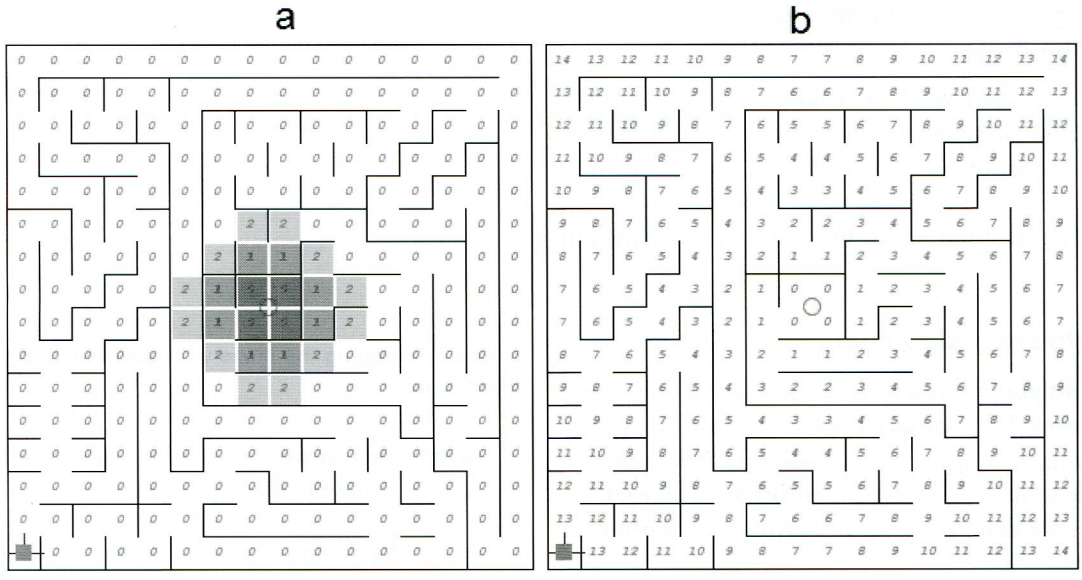

Figure 3: Initial flooding of the maze: a) the second step of flooding, b) maze completely flooded.

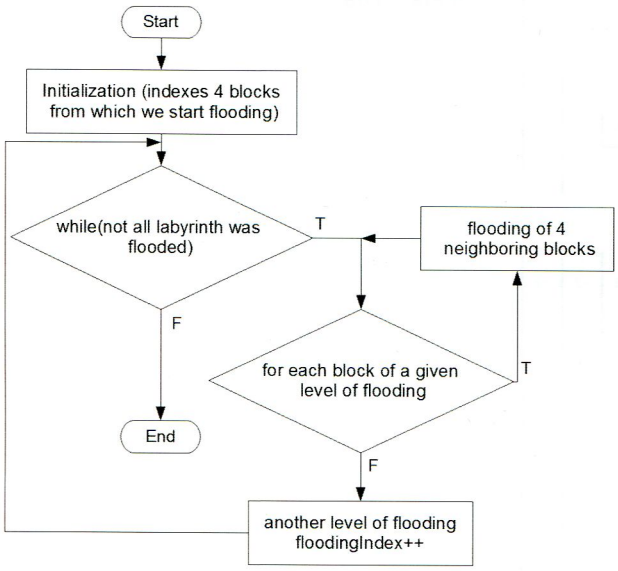

Figure 4: Block diagram of the flooding algorithm for the entire maze.

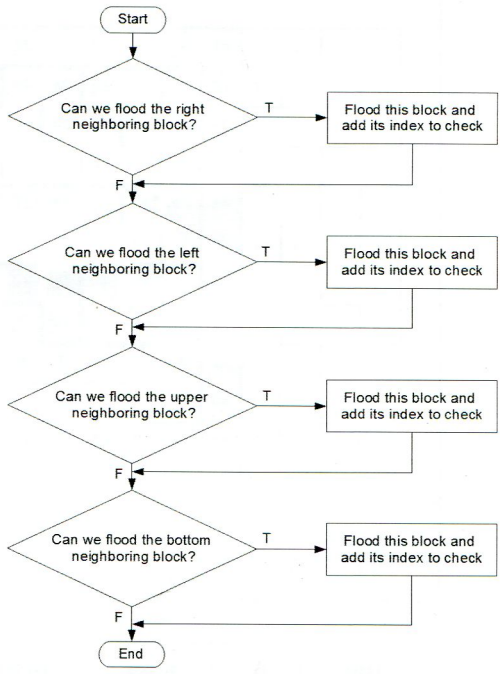

Figure 5: Block diagram of the flooding function of neighbouring cells.

of flooding. Using flooding during the mapping and goal searching algorithm make the robot quickly heading for the center of the maze.To reduce the simulation time it was assumed that the robot maps the maze until it reaches the goal. This diagram shows that the robot detects the current cell wall and then checks whether there is a neighbouring block with a lower or equal value where it can go as shown in Figure 6. Flooding maze begins only when all the neighbouring blocks have higher values then the block in which the robot is currently located. This operation changes the current values of maze blocks and sets a new route to a goal in the direction of decreasing values.

2. Finding the shortest path from a mapped area of the maze

After reaching the goal the last flooding occurs and the shortest path linking the starting point with the goal from the mapped part of the maze is determined as shown in Figure 9b (second drive). The maze flooding is based on the *flood fill* algorithm which block diagram is shown in Figure 4.

3. Fault-tolerant algorithm

A designed algorithm is resistant to sensors failures in any configuration. For a proper operation of the algorithm the following assumptions have to be met: detection system is able to determine which sensors are damaged, assumptions we denote the set of vectors representing all the sensors as S_A.

$$S_A = \{\vec{s}_1, \vec{s}_2, ..., \vec{s}_n\} \quad (1)$$

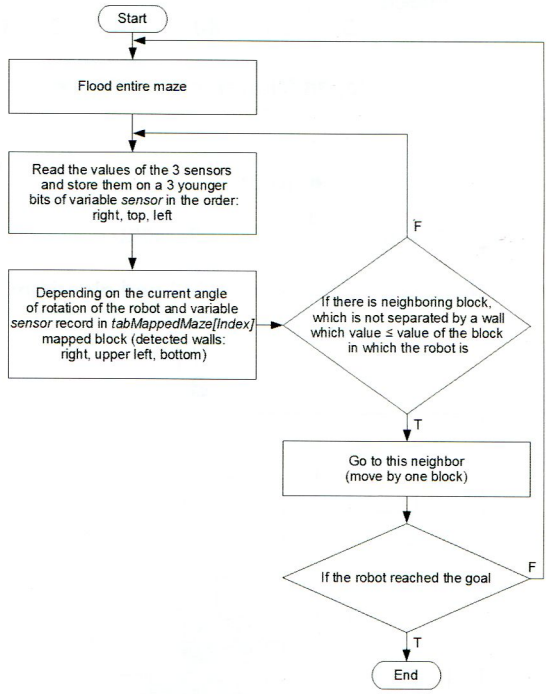

Figure 6: Mapping the maze with goal searching (flood fill algorithm).

Among this set we can identify a subset S_F of the vectors representing the damaged sensors

$$S_F = \{\vec{s}_1*, \vec{s}_2*, ..., \vec{s}_m*\} \quad (2)$$

We assumed earlier that at least one sensor must be working to allow the robot to continue to solve the maze

$$S_F \subseteq S_A \tag{3}$$

The symmetrical arrangement of sensors at a certain angle, as shown in Figure 7, is often used in commercial applications.

This arrangement of the sensors provides an easy way to measure the values at the positions of faulty sensors by making rotation by an angle θ

$$\vec{s_n} = R\vec{s}_{n-1} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \vec{s}_{n-1} \tag{4}$$

The most common failure in practice is a single sensor failure.

For such a case the algorithm can be simplified to perform the following steps:

1. Find the nearest neighbour of faulty sensor (the closest working sensor).
2. Perform the rotation of the selected sensor in the position of the faulty sensor.
3. Make measurement at this position.
4. Turn back to the original position.

After performing the above actions the robot can continue its mapping algorithm.

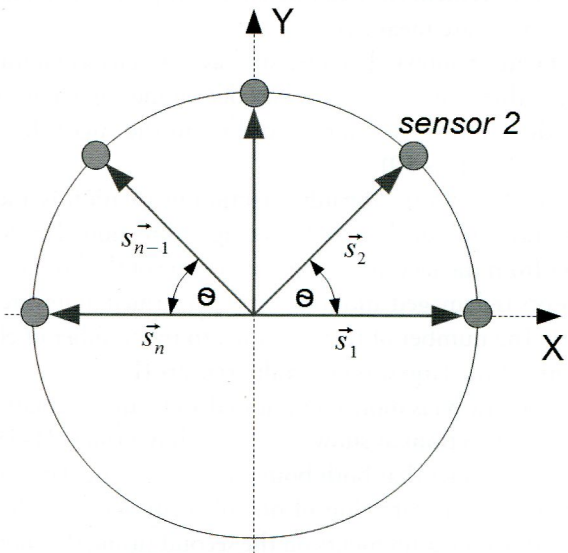

Figure 7: Symmetrical arrangement of the sensors vectors.

Algorithm 2: Fault-tolerant algorithm for single sensor failure

```
Input : Faulty sensor index (FsIndex)
Output : Measurement at the position off aulty sensor
         (mArray[FsIndex])
1 if (FsIndex == s_n) then
2 |   rotation (1, θ) ;
3 |   mArray[FsIndex] = getSensorValue(FsIndex-1)
4 |   rotation (- 1, θ) ;
  else
5 |   rotation (- 1, θ) ;
6 |   mArray[FsIndex] = getSensorValue(FsIndex + 1)
7 |   rotation (1, θ) ;
  end
```

Algorithm 3: Fault-tolerant algorithm for multiple sensors failures

```
Input: Set of indexes of faulty sensors
Output: Measurements at the positions of faulty sensors
        (mArray[ ])
1  findMaxMinIndexes(); // find (FsIndex_max, FsIndex_min) ;
2  NumberOfFaultySets=findAllSetsOfFaultySensors();
3  foreach (i < NumberOfFaultySets; i + +) do
4  |   FaultySet[0][i]=findSmallestSetIndex(i);
5  |   FaultySet[1][i]=findNumberOfElements(i);
   end
6  if (FsIndex_max == s_n) ∩ (FsIndex_min == s_1) then
7  |   direction = 2 ;
8  else if (FsIndex_max == s_n) then
9  |   direction = 1 ;
   else
10 |   direction = −1 ;
   end
11 foreach (i < NumberOfFaultySets; i + +) do
12 |   if (direction==1) then
13 |   |   nearestActiveNeighbourIndex[0][i]=
14 |   |   FaultySet[0][i]-1;
15 |   else if (direction==-1) then
16 |   |   nearestActiveNeighbourIndex[0][i]=
17 |   |   FaultySet[0][i]+FaultySet[1][i] ;
   |   else
18 |   |   if (i==0) then
19 |   |   |   nearestActiveNeighbourIndex[0][i]=
20 |   |   |   FaultySet[0][i]+FaultySet[1][i];
21 |   |   |   nearestActiveNeighbourIndex[1][i]=-1;
   |   |   else
22 |   |   |   nearestActiveNeighbourIndex[0][i]=
23 |   |   |   FaultySet[0][i]-1;
24 |   |   |   nearestActiveNeighbourIndex[1][i]=1;
   |   |   end
   |   end
   end
25 Ls=findLargestSetOfFaultySensors();
26 sLs=findSecondLargestSetOfFaultySensors();
27 if (direction ≠ 2) then
28 |   foreach (i < Ls; i + +) do
29 |   |   rotation(direction, θ);
30 |   |   getSensorsValues();
   |   end
31 |   rotation(−direction, θ * Ls);
   else
32 |   rightSetLimit=FaultySet[1][0];
33 |   foreach (i < rightSetLimit; i + +) do
34 |   |   rotation(−1, θ);
35 |   |   getSensorsValues();
   |   end
36 |   rotation(1, θ * rightSetLimit);
37 |   if (rightSetLimit==Ls) then
38 |   |   numberOfTurns=sLs;
   |   else
39 |   |   numberOfTurns=Ls;
   |   end
40 |   foreach (i < numberOfTurns; i + +) do
41 |   |   rotation(1, θ);
42 |   |   getSensorsValues();
   |   end
43 |   rotation(−1, θ * numberOfTurns);
   end
```

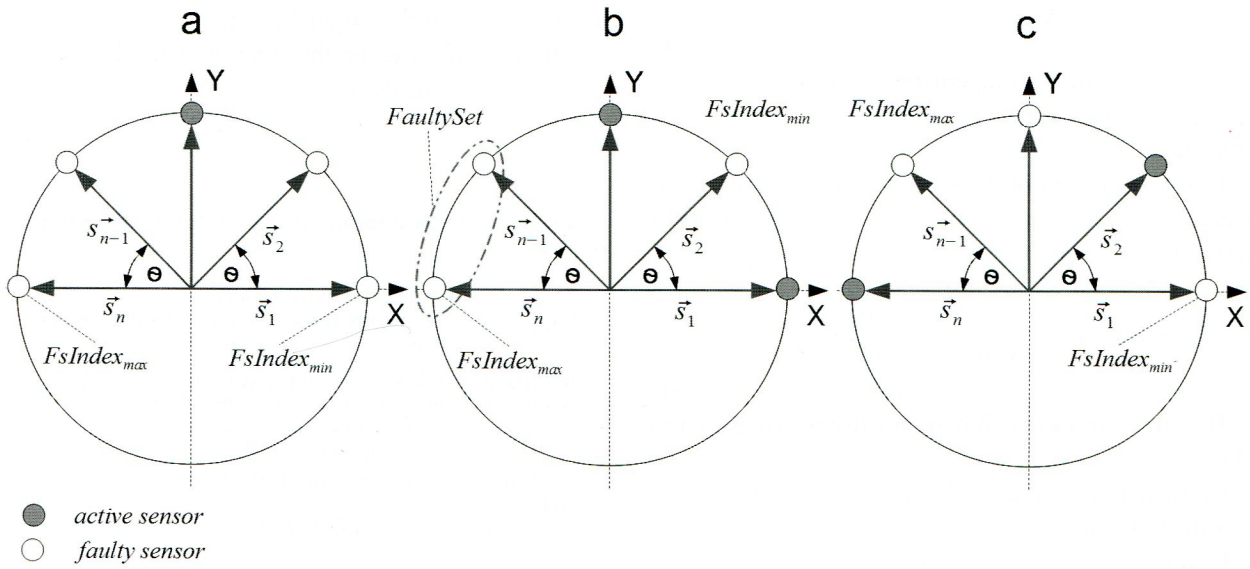

Figure 8: Characteristic cases of sensors failures: a) boundary sensors failures, b) sensor failure at index n, c) sensor failure at index 1.

Algorithm 2 shows the fault-tolerant algorithm for a single sensor failure. An input argument of this algorithm is faulty sensor index. The output result of this algorithm is a cell of array containing the measurement at the position of a faulty sensor. The *getS ensorValue* function performs the measurement using a sensor with the index contained in the argument.

If more than one sensor is faulty then for each of them measurements are made at the correct position by working sensors.

In this case, the algorithm includes the following steps:

1. Find the maximum and minimum index ($FsIndex_{max}$; $FsIndex_{min}$) from the set of all damaged sensors.
2. Find the smallest index for each set of damaged sensors and the number of its elements.
3. Compare indexes ($FsIndex_{max}$; $FsIndex_{min}$) with indexes limit s_1 and s_n. Based on this comparison set the initial direction of rotation.
4. For each set find the nearest working (active) neighbour based on the direction of rotation.
5. Assign each active nearest neighbour connected with the given set a number of measurements to do and the direction.
6. Perform the number of turns in a given direction equal to the number of elements of the largest set of damaged and neighbouring sensors. After each rotation the values of selected sensors are measured if they are in positions where the damage occurred.
7. For the special case ($FsIndex_{max} = s_n$; $FsIndex_{min} = s_1$) after the rotation and measurements in a given direction a robot turns back to the original position. Then it continues to rotate in this direction performing measurements until all signals at the position of faulty sensors are measured.

Figure 8 shows characteristic cases of sensors failures. These three characteristic cases determine the choice of the direction of rotation according to the Algorithm 3 (lines of code 6-10).

For case 8b the default direction of rotation is a left direction and for the case 8c is a right direction. This follows from the fact that in both cases one of the boundary sensors is damaged and rotation is performed in its direction. The number of turns is equal to the number of elements of the largest set of faulty sensors (Ls).

The case 8a is more complicated and requires rotation in both directions as shown in Algorithm 3 (lines 31-41). Due to the fact that both boundary sensors are damaged rotation in the direction of one of them does not allow to perform measurements on the second (using the shortest angular distance). To perform measurements at all positions of damaged sensors we should divide this task into two stages. In the first stage, we rotate a robot to the right until an active sensor reach the position at index 1. Then we turn a robot back to its original position. In the second step we rotate a robot in the opposite direction than in the first stage.

In the case of 8a two of the largest sets of faulty sensors are calculated (*Ls* and *sLs*). If the right boundary sensor belongs to the largest faulty sensors set then number of turns in the second phase is equal to the number of elements of the second largest set. A choice of such a condition allows us to perform measurements at all positions of damaged sensors.

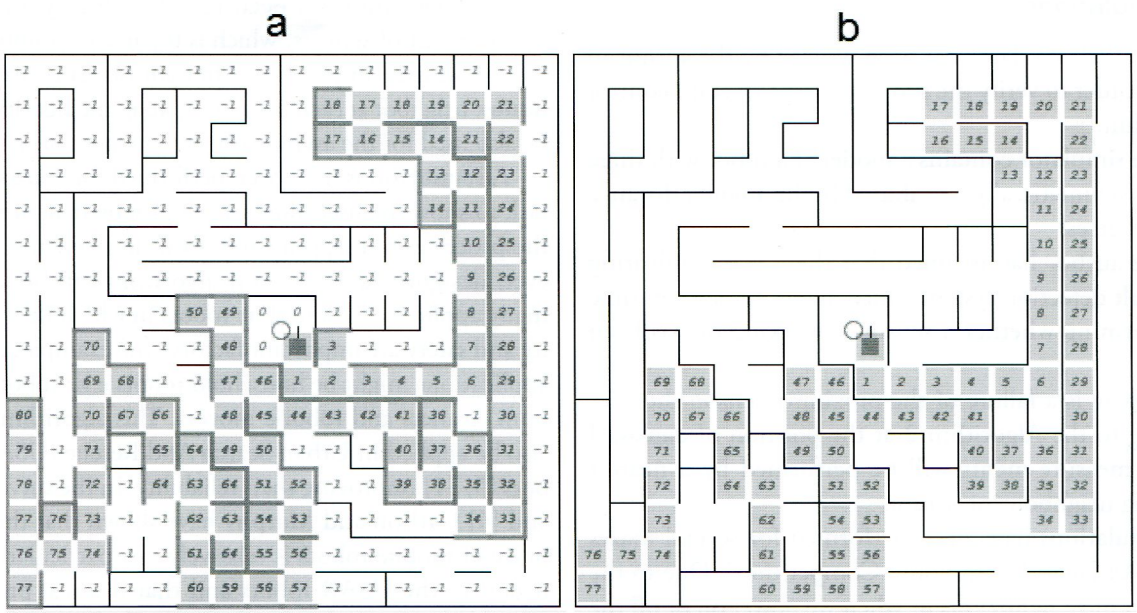

Figure 9: All Japan Micromouse Competition 2010 for the case of failure of the right and left sensor: a) mapping with goal searching b) finding the shortest path of the mapped part of the maze.

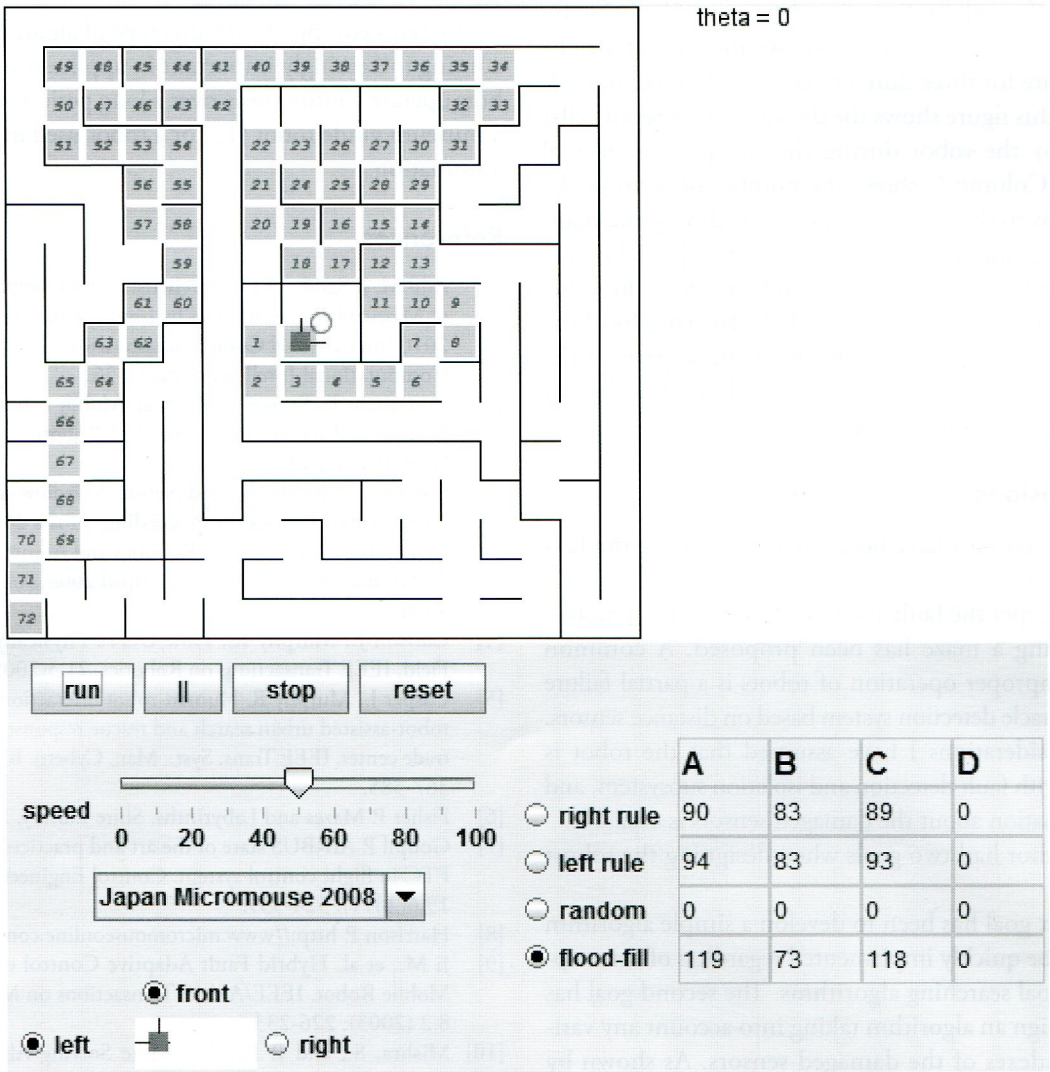

Figure 10: Finding the shortest path of the mapped part of the maze for the case of right sensor failure.

4. Simulations

A dedicated simulator has been written by the author in Java in order to verify the proposed fault-tolerant control algorithm.

The simulator contains a model of a robot with three sensors symmetrically distributed in the angular distance of 90 degrees.

The author has assumed that there is a monitoring and fault detection system which allows for a given time to determine whether a sensor is working or has been damaged.

If there is a failure of the proximity sensor, then according to the Algorithm 2 at the position of damaged sensor measure-ments will be performed by a nearest working neighbouring sensor.

Simulations have been performed for several mazes from "All Japan Micromouse Competition". Due to the fact that most failures occur when we power on or off the device, we have assumed that the fault occurs from the first moment of the simulation.

The results for the case of left and right sensor failure for mapping and finding the shortest path tasks are shown in Figure 9. Figure 10 shows the case of a right sensor failure for three different control algorithms. Column A in this figure shows the distance (number of cells) travelled by the robot during the mapping with goal searching. Column C shows the number of turns of the robot (turns to the measuring position) during the mapping with a faulty left sensor. In the last cell (goal point) the measurement is not needed and therefore the value in cell C is one less than in the cell A. The lengths of the shortest paths to the goal from the mapped parts of the maze are shown in column B and are always less than or equal to the values in column A.

5. Conclusions

The simulations have been carried out using the Java environment.

In this paper the fault-tolerant algorithm for a mobile robot solving a maze has been proposed. A common cause of improper operation of robots is a partial failure of their obstacle detection system based on distance sensors. In my considerations I have assumed that the robot is equipped with fault detection and isolation subsystem, and that information about the damaged sensors is available.

The author had two goals when designing the robust algorithm.

The first goal has been to develop a simple algorithm that could be quickly implemented regardless of the mapping and goal searching algorithms. The second goal has been to design an algorithm taking into account any variances of indexes of the damaged sensors. As shown by simulation results both goals have been achieved.

The algorithm has been tested with a symmetrical arrangement of sensors, which is the most common configuration used in practice. After adjustment (search for nearest neighbor at each measuring step), the algorithm can be also used for the asymmetric arrangement of sensors.

All simulations have been performed for maze solving task to show the usefulness of the proposed algorithm in the processes of mapping and obstacle avoidance. However, designed fault-tolerant algorithm can be used in many applications of mobile robots in which the reliability and safety are important factors. The examples of such applications might be household robots e.g.: robotic vacuum cleaners and robotic movers. This algorithm could also be applied in Urban Search And Rescue (USAR) robots [5, 11] improving their reliability. In the surveys: [3] and [4] Carlson and Murphy have shown that the reliability of mobile robots is relatively low particularly with regard to the USAR robots and that is a real problem. The proposed solution can be particularly useful for robots with multiple obstacles detection sensors allowing them to continue current task in cases where some of the sensors have been damaged.

Systems equipped with this type of algorithms are becoming increasingly important in applications where robots operate continuously over a long period of time, e.g.: a museum guide robot [13] or a robot used in warehouse automation.

References

[1] Adil M. J. Sadik , A Comprehensive and Comparative Study of Maze-Solving Techniques by Implementing Graph Theory, 2010 International Conference on Artificial Intelligence and Computational Intelligence.pp.52-56.

[2] Borenstein J., Everett H.R., et al. Mobile robot positioning Sensors and techniques. Journal of Robotic Systems, 14.4 (1997): 231-249.

[3] Carlson J., Murphy R., and Nelson A. Follow-up analysis of mobile robot failures. In Proceedings of the 2004 IEEE International Conference on Robotics and Automation (ICRA 2004), Barcelona, Spain, 18-22 April 2004, vol.5, pp. 4987-4994.

[4] Carlson J., Murphy R., How UGVs Physically Fail in the Field, IEEE Transactions on Robotics, 21.3(2005): 423-437.

[5] Casper J., Murphy R. Human-robot interactions during the robot-assisted urban search and rescue response at the world trade center. IEEE Trans. Syst., Man, Cybern. B, 33.3(2003): 367-385.

[6] Fisher P. Mazes and Labyrinths. Shire Library, 2008.

[7] Goupil P. AIRBUS state of the art and practices on FDI and FTC in flight control system. Control Engineering Practice, 19.6(2011): 524-539.

[8] Harrison P. http://www.micromouseonline.com

[9] Ji M., et al. Hybrid Fault Adaptive Control of a Wheeled Mobile Robot. IEEE/ASME Transactions on Mechatronics, 8.2 (2003): 226-233.

[10] Mishra, S., and P. Bande. "Maze Solving Algorithms for Micro Mouse". In Proc. of the IEEE International Confer-

ence on Signal Image Technology and Internet Based Systems (SITIS '08). 30 Nov.-3 Dec. 2008, Bali, Indonesia, 2008: 86-93.

[11] Murphy R. Trial by Fire, IEEE Robotics and Automation Magazine, 11.3(2004): 50-61.

[12] Nou H., Theilliol D., et al. Fault-tolerant Control Systems: Design and Practical Applications. Springer-Verlag London, 2009.

[13] Nourbakhsh I. R., The mobot museum robot installations, in Proc. IEEE/RSJ IROS 2002 Workshop Robots in Exhibitions, 2002, pp. 14-19.

[14] Srebro A. "A self-tuning fuzzy PD controller for a wheeled mobile robot operating in the presence of faults". The Challenges of Modern Technology, 2.4(2011): 11-20.

[15] Vladimir J. L., A Comparative Study on the Path Length Performance of Maze-Searching and Robot Motion Planning Algorithms. IEEE Transactions on Robotics and Automation 1.1(1991): 57-66.

[16] Wright C. The Maze and theWarrior: Symbols in Architecture, Theology, and Music. Shire Library. Harvard University Press, 2001.

[17] Zhang, X., "A Micromouse Maze Solving Algorithm". MCU and Embedded System 5(2007): 84-85.