World-Learning and Simultaneous Path Planning with a Mobile Robot

Rui Araújo†, A. T. de Almeida†

†Institute of Systems and Robotics (ISR); and
Electrical Engineering Department; University of Coimbra;
Pólo II; Pinhal de Marrocos; 3030 Coimbra - PORTUGAL;
email: rui@ISR.uc.pt

Abstract: In this paper we treat the problem of learning a model of a mobile robot world. We describe an algorithm that also enables learning to navigate, from a start position to a goal region on a world where the location and shape of obstacles is unknown. Additionally, original algorithm modifications will be proposed that are able to improve the algorithm performance. The robot uses its own infrared distance-sensors to detect obstacles present on its path. Besides detecting obstacles, the only other initial robot's ability is to perform straight-line motion. Simulation results will be presented that validate both, the effectiveness of the approach to navigate the mobile robot, and the benefits of the proposed algorithm modifications.

Keywords: World Learning, Path Learning, Mobile Robot.

1. Introduction

In this paper we are concerned with the robot path finding problem with a mobile robot, where a path that avoids collisions with obstacles must be generated between an initial and a goal robot 2D configurations. The mobile robot control system is assumed to have no initial knowledge about the map of the world. A self-learning approach overcomes this lack of a model, thus increasing robot autonomy.

In mobile robotics problems, it is usual to assume world knowledge, generally appearing in the form of a global map of the world over which path planning algorithms operate. However it is difficult to provide the robot with a global map model of its world. The difficulties may arise from various reasons. The map building operation is itself a difficult separate problem. It may become even more complicated if the robot environment changes. Also, if the robot control system requires the introduction of the global map, this may become a tedious and time consuming programming task.

In this paper we demonstrate the effectiveness of the parti-game Algorithm [5] to the specific case of learning an internal representation of the world environment of a mobile robot. The algorithm does not have any initial internal representation, map, or model, of the world, in particular it has no information regarding the location and the shape of obstacles. In addition to learning this representation, the algorithm is also able to learn a mobile robot path from an initial position, to a known goal region in the world. We further propose modifications aimed at improving algorithm performance. We demonstrate that the mobile robot can learn to navigate to the goal having only the predefined abilities of doing straight-line motion, and obstacle detection (not avoidance) using its own distance sensors. We also demonstrate that the novel proposed modifications, clearly improve the learning algorithm operation.

The organisation of the paper is as follows. Section 2 presents the learning controller architecture. In section 3 we present an original modification to the learning algorithm that enables a clear improvement on its performance. In section 4 we present the experimental environment around which we performed the experiments concerning the application of the exploration algorithm. Section 5 presents results of simulations using the controller for learning both a world model, and to navigate the robot to a goal region on the unknown world. Finally in section 6 we make some concluding remarks.

2. Learning Controller Architecture

The problem we wish to solve in this work may be stated as follows: The mobile robot is initially on some position on the environment, or world, and then it must learn a path to a known goal region in the world. Also the algorithm does not have any initial internal representation, map, or model, of the world. In applying the algorithm, we assume that the initial abilities of the mobile robot are only two. First, it is able to perform obstacle detection operations, i.e. to detect obstacles that may obstruct its normal path. Second, the mobile robot is able to perform straight line movements between its current position and some other specified position in the world. Two comments to this second ability are in order. First, this ability requires the knowledge of the robot current position, and second the movements may fail because of the presence of an obstacle that is detected.

The mobile robot controller architecture used in this work, is based on the application of the parti-game learning controller algorithm [5] to the specific case of learning a mobile robot path. In this paper we demonstrate the effectiveness of this approach to the above stated problem.

The parti-game algorithm can be applied to learning control problems in which: (1) We have continuous and multidimensional state and action spaces; (2) “Greedy” and hill-climbing techniques can become stuck, never reaching the goal; (3) random exploration
can be intractably time-consuming; (4) We have unknown, and possibly discontinuous, system dynamics and control laws.

Additionally the algorithm has some restrictions that however, do not prevent its application to our problem: (1) The dynamics are deterministic. (2) The task is specified by a known goal region. (3) A feasible solution is found, not necessarily an optimal path according to a particular criterion. (4) A local greedy controller is available, which we can ask to move greedily towards a desired state. However, there is no guarantee that a request to the greedy controller will succeed. On our case for example, it is possible for the greedy controller (the "straight-line mover") to become stuck because of an obstacle found by the robot.

2.1. The Algorithm

The parti-game algorithm is based on partitioning the state-space. It begins with a large partition. Then it increases the resolution by subdividing the state-space where the learner predicts that a higher resolution is needed. As usual a partitioning of the state-space is a finite set of disjoint regions, the union of which covers the entire state-space. Those regions will be called cells, and will be labelled with integers 1, 2, \ldots, N. In this paper we will assume that the cells are all axis aligned hyperrectangles (rectangles in our 2D case). Although this assumption is not strictly necessary, it simplifies the computational implementation of the algorithm. A real-valued state, s, is a vector of real numbers in a multidimensional space (2D space in our case). Real-valued states and cells are distinct entities. Each real-valued state is in one cell and each cell contains a continuous set of real-valued states. Let us define \text{NEIGHS}(i) as the set of neighbours, or cells which are adjacent to i. Specifically, two cells are said to be neighbours if their intersection is a set pointed with one constant coordinate, and all the other coordinates freely take values from an interval of real numbers. When we are at a cell i, applying an action consists of actuating the local greedy controller "aiming at cell j". A cell i has an associated set of possible actions that is defined as \text{NEIGHS}(i). Each action can thus be labelled by a neighbouring cell.

The algorithm uses an environmental model, which can be any model (for example, dynamic or geometric) that we can use to tell us for any real-valued state, control action, and time interval, what will be the subsequent real-valued state. In our case the "model" is implemented by the mobile robot (which can be real or simulated), and takes the current position, and position command, to generate the next robot position.

Let us define \text{NEXT-PARTITION}(s, j) function that tells us in which cell we end up, if we start at a given real-valued state, s, and using the local greedy controller, keep moving toward the centre of a given cell, j, until either we exit the initial cell or get stuck. Let i be the cell containing the real-valued state s. If we apply the local greedy controller "aim at cell j" until either cell i is exited or we become permanently stuck in i, then

\[
\text{NEXT-PARTITION}(s, j) = \begin{cases} 
  i & \text{if we became stuck} \\
  \text{the cell containing the exit state otherwise}
\end{cases}
\]

In our work, the test for sticking performs an obstacle detection with the distance sensors of the mobile robot (see section 4). In other systems, sticking could be tested by seeing if the system has not exited the cell after a predefined time interval.

Let \text{CENTER}(i) be the real valued state at the centre of cell i. Define \text{NEXT}(i, k) as the cell where we arrive if, starting at \text{CENTER}(i), we apply the local greedy controller "aiming at cell k":

\[
\text{NEXT}(i, k) = \text{NEXT-PARTITION}(\text{CENTER}(i), k)
\]

In general \text{NEXT}(i, k) \neq k because the local greedy controller is not guaranteed to succeed.

Since there is no guarantee that a request to the local greedy controller will succeed, each action has a set of possible outcomes. The particular outcome of an action depends on the real-valued state s, from which we start "aiming". The outcomes set, of an action j in cell i, is defined as the set of possible next cells:

\[
\text{OUTCOMES}(i, j) = \begin{cases} 
  \{ k \mid \text{exists a real-valued state s in cell i for which} \text{NEXT-PARTITION}(s, j) = k \}
\end{cases}
\]

Define the cell-length of a, possibly not continuous, path on the state space, as the number of cell transitions that take place as we go through the path. When the system is on the real-valued state s of a cell i, one of the key decisions that the algorithm has to take, is to choose the cell at which the system should aim using the local greedy controller. The method for making this decision is called a policy.

We can, using a worst case assumption, define the shortest path from cell i to the goal, JWc(i), as the minimum number of cell transitions to reach the goal assuming that, when we are in a certain cell i and the our intended next cell is j, an adversary is allowed to place us in the worst position within cell i prior to the local controller being activated. The JWc(i) shortest-path is defined as:

\[
JWc(i) = \min_{k \in \text{NEIGHS}(i)} \max_{j \in \text{OUTCOMES}(i, k)} JWc(j) \\
1 + \text{NEIGHS}(i) - \text{OUTCOMES}(i, k)
\]

except, \( JWc(i) = 0 \), if \( i = \text{GOAL} \)

The JWc(i) values can be obtained by the minimax algorithm [2] or by Dynamic Programming. The value of JWc(i) can be \(+\infty\) if, when we are at cell i, our adversary can permanently prevent us from reaching the goal. By definition such a cell is called a losing cell. With this method, the next cell to aim is the neighbour, i, with the lowest JWc(i). Using this approach we are sure that, if JWc(i) = n, then we will get n or fewer transitions to get to the goal starting from cell i. However, the method is too much
Algorithm 1
Repeat forever

1. For each cell $i$ and each neighbour $j \in \text{NEIGHS}(i)$, compute the \text{OUTCOMES}(i, j)$ set in the following way:

1.1 If there exists some $k'$ for which $(i, j, k') \in D$ then:

\[ \text{OUTCOMES}(i, j) = \{ k \mid (i, j, k) \in D \} \]

1.2 Else, use the optimistic assumption in the absence of experience:

\[ \text{OUTCOMES}(i, j) = \{ j \} \]

2. Compute $J_{WC}(i')$ for each cell using minimax.

3. Let $i :=$ the cell containing the current real-valued state $s$.

4. If $i = \text{GOAL}$ then exit, signalling SUCCESS.

5. If $J_{WC}(i) = \infty$ then exit, signalling FAILURE.

6. Else

6.1 Let $j := \text{arg min}_{j' \in \text{NEIGHS}(i)} \text{max}_{k \in \text{OUTCOMES}(i, j')} J_{WC}(k)$

6.2 While (not stuck and $s$ is still in cell $i$)

6.2.1 Actuate local greedy controller aiming at $j$.

6.2.2 $s :=$ new real-valued state.

6.3 Let $i_{\text{new}} :=$ the identifier of the cell containing $s$.

6.4 $D := D \cup \{(i, j, i_{\text{new}})\}$

Loop

Algorithm 2
While ($s$ is not in the goal cell)

1. Run Algorithm 1 on $s$ and $P$. Algorithm 1 returns the updated database $D$, the new real-valued state $s$, and the success/failure signal.

2. If FAILURE was signalled then

2.1 Let $Q :=$ All losing cells in $P$ ($J_{WC} = \infty$).

2.2 Let $Q' :=$ The members of $Q$ who have any non-losing neighbours.

2.3 Let $Q'' :=$ $Q'$ and all non-losing members of $Q'$.

2.4 Split each cell of $Q''$ in half along its longest axis producing a new set $R$, of twice the cardinality.

2.5 $D := D + R - Q''$

2.6 Recompute all new neighbour relations, and delete from the database $D$, those triplets that contain a member of $Q''$ as a start point, an aim-for, or an actual outcome.

Loop

Pessimistic because, regions of a cell that will never be actually visited, are available for the adversary to place us. But those may be precisely the regions that lead to an eventual failure of the process. So although this method guarantees success if it finds a solution, it may often fail on solvable problems.

Next we will describe Algorithm 1, that reduces the severity of this problem by considering only all empirically observed outcomes, instead of all possible outcomes for a given cell. Another argument contributing to this solution, is that as a learning algorithm, it is more important to learn the outcomes set, only from real experience on the behaviour of the system. Besides that, it could be difficult or impossible, to compute all possible outcomes of an action. Whenever an \text{OUTCOMES}(i, j)$ set is altered due to a new experience obtained, equations (1) and (2), are again solved in order to find the path to the goal. Before an action is experienced, we can not leave the \text{OUTCOMES}(i, j)$ set empty. In these situations we use, the default optimistic assumption that we can reach the neighbour that is aimed. Algorithm 1 (see box) keeps apply-
Ing the local greedy controller, aiming at the next
cell, on the “minimax shortest path” to the goal, un-
til either we are caught on a losing cell \( J_{WC} = \infty \),
or reach the goal cell. Whenever a new outcome is
experienced, the system updates the corresponding
OUTCOMES \((i, j)\) set, and equations (1) and (2) are
solved, to obtain the, possibly new, “minimax short-
est path”. Step 6.1 computes the next neighboring

cell on the “minimax shortest path” to the goal. Algo-

rithm 1 has three inputs: (1) The current (on entry)
real-valued state \( s \); (2) A partitioning of the state-
space, \( P \); (3) A database, the set \( D \), of all previously
different cell transitions observed in the lifetime of the
partitioning \( P \). This is a set of triplets of the following
form: (start-cell, aimed-cell, actually-attained-cell).

At the end Algorithm 1 returns three outputs: (1)
The updated database of observed outcomes, \( D \), (2)
the final, real-valued system-state \( s \), and (3) a boolean
variable indicating SUCCESS or FAILURE.

We see that Algorithm 1 gives up when it discovers
it is in a losing cell. One of the hypothesis of the algo-

rithm is that all paths through the state space are
continuous. Assuming that a path to the goal actu-
ally exists through the state-space, i.e. the problem is
solvable, then there must be an escaping-hole al-

lowing the transition to a non-losing cell and eventually
opening the way to reach the goal. This hole has been
missed by Algorithm 1 by the lack of resolution of the
partition. A hole for making the required transition to
a non-losing cell, can certainly be found on the cells
at the borders between losing and non-losing cells.

Taking this comments into account, the top level Al-
gorithm 2 (see box), divides in two the cells in the
borders, in order to increase the partition resolution,
and to allow the search for the mentioned escaping-
hole. This partition subdivision takes place between,
each successive calls to Algorithm 1 that keep tak-
ing place while the system not reach the goal region.
Algorithm 2 has three inputs: (1) The current (on entry)
real-valued state \( s \); (2) A partitioning of the state-
space, \( P \); (3) A database, the set \( D \), of all pre-
viously different cell transitions observed in the lif-
time of the partitioning \( P \). This is a set of triplets of
the form: (start-cell, aimed-cell, actually-attained-
cell). At the end, Algorithm 2 returns two outputs:
(1) The new partitioning of the state-space, and (2)
a new database of outcomes \( D \).

3. Algorithm Improvement

In this section we discuss some aspects of the be-

haviour of the algorithm presented in section 2. This
discussion will motivate and enable the presentation of
algorithm modifications aimed at increasing its ef-
effectiveness. Experimental experience has shown that,
with this novel contribution, the algorithm behaviour
has been able to improve on most world environments.
Additionally, in the experiments that were carried
with the new approach, we were able to tune the
new algorithm, such that it never exhibited a strongly
worse, or even slightly worse, behaviour. It is possi-
ble to imagine “worst-case” world environments for
which the new approach is somewhat inferior. How-
ever if the original algorithm was able to solve those
path-planning problems, the new approach will also

be successful.

As a simplified example of a situation that algorithm
of section 2 often faces, suppose that the set of cells
to be split can be decomposed in two subsets (fig. 1).
The cells of each of these subsets form two separated,
closed, and bounded regions on the state-space, Re-
gion A and Region B. Region A is composed of “small-
sized” cells, and Region B is composed of “big” cells.

Here there are various possibilities for defining the size
of a cell:

1. The length of the greatest (or the smallest) side
   of the hyperrectangle.

2. The generalised volume of the hyperrectangle
   that may be defined as the product of the length
   of all its sides.

Assuming that a path to the goal actually exists
through the state-space, i.e. the problem is solvable,
then there must be an escaping-hole, either in Re-
gion A or in Region B, allowing the transition to
a non-losing cell and eventually opening the way to
reach the goal. However, since we don’t know where
this hole really is, there is no information support-
ing a preference for concentrating the search in only
one of the two regions. But this implies that, there
must be at least one system traversal of the possibly
long path between the two regions, for every time a
set of splits occurs (every cycle of Algorithm 2 - sec-
tion 2). This path can become quite long, not only
because of the distance between the two regions, but
also because of the existence of obstacles. When learn-
ing with a physical system, the traversal of this path
typically implies the use of a certain number of sam-
pling intervals. This fact along with an increase of
computational costs that in most of the cases is also
associated to the traversal, leads to the increase of
the system learning time. Thus we need a method for
decreasing those traversals.

The original cell splitting strategy of the algorithm
(section 2) is based on splitting of all cells of \( Q'' \), re-
sulting on the generation of the set of cells \( R \). For
improving the algorithm in a situation like the one we are
ALGORITHM 3 (this algorithm provides an alternate splitting strategy to be used in ALGORITHM 2)

1. Let $Q_1'' := \text{All of cells of } Q''$, whose size is equal to one of the $N_1$ biggest sizes of cells of $Q''$. Call $R_1$, the set of all the cells resulting from the splitting of cells from $Q''_1$. Note that $R_1 \subset R$.

2. Let $Q_2'' := Q'' - Q_1''$. This implies $Q'' = Q_1'' + Q_2''$. Call $R_2$, the set of all the cells resulting from the splitting of cells from $Q''_2$. Note that $R = R_1 \cup R_2$.

Rule 1. Split all the cells of $Q_1''$. Let $Q'' := Q_1''$.

3. As a temporary situation for preparing Rule 2, split all the cells of $Q''$, and use the minimax algorithm to obtain the shortest-path from the new actual-cell to the GOAL cell.

Rule 2. IF $N_2$ cells among the $N_3$ initial cells of the shortest-path from the new current-cell to the GOAL, belong to $R_2$ THEN:

R2.1 Let $Q'' := Q'' + Q_0''$, where the set $Q_0''$ is composed according to one of the following options:

   Option 1. $Q_0''$ contains all the cells of $Q_2''$ (note that $Q'' = Q''$ in this case).

   Option 2. $Q_0''$ contains all the cells of $Q_2''$ belonging to the shortest-path.

4. Change ALGORITHM 2 to split all the cells of $Q''$ instead of splitting all the cells of $Q''$.

analysing, we argue that there is no reason for insisting to search for an extremely narrow escaping-hole in Region A before we search for a wider escaping-hole on the bigger cells of Region B. Thus if we change the strategy in order to favour the splitting of the bigger cells, we may expect that the system does not to take so much of its time travelling to and from Region A, because it is no longer so much “interested” on analysing the smaller cells. Note that a worse case occurs when the cell sizes are balanced between Regions A and B. In this case more travelling occurs between the two regions, but this is no worse than the original problem that we want to solve. Specifically as a first approach we will only split a subset, $Q_1''$, of cells of $Q''$, whose size is equal to one of the $N_1$ biggest sizes of cells of $Q''$. This splitting produces a new set of cells $R_1 \subset R$. This is the Rule 1 of the new strategy - see box.

However, there are situations when it may be worth splitting smaller cells from Region A. Suppose the system is close to Region A, and all the cells of $Q''$ were split. Then the minimax algorithm was used and a resulting shortest path includes some small cells of Region A. In this case it is better also to split the smallest cells of $Q''$, because in this way we are able to promptly analyse the, possibly feasible, shortest path before we travel all the path to Region B where the biggest cells will be explored.

Algorithm 3 (see box) summarises and precisely specifies the modifications on the cell splitting strategy that were made in Algorithm 2 in order to improve it. Note that, since $Q'' = Q_1'' + Q_2''$, when using Option 1, all the cells are split when the condition of Rule 2 is verified.

4. Experimental Environment

In this section we give details of the experimental environment that we used to prove, the effectiveness of the algorithm presented in section 2 to solve the robot path finding problem that was formulated on the same.

Figure 2. The Khepera miniature mobile robot.
values may give an approximate idea of the distance between the sensor and the external object.

The results reported in this paper were achieved using the "Khepera Simulator Version 2.0" [3]. This simulator allows a Khepera robot to evolve on a square world of 1000 mm of side width, where an environment for the robot may be created by disposing bricks, corks and lamps.

To calculate its output distance value, a simulated robot sensor explores a set of 15 points in a triangle in front of it. An output value is computed as a function of the presence, or absence, of obstacles at these points. A random noise corresponding to ±10% of its amplitude is added to the output light value. The light value output of a sensor is computed as a function of both the distance and the angle between the sensor and the light source. A ±5% random noise is added to the resulting value. In our simulations we do not use corks and lamps, but only bricks. Also we do not use the light value reading from the sensors but only the distance value.

For the sticking condition test primitive, that is required in the algorithm described in section 2, we use \( d(0), \ldots, d(5) \), which are the distance values of robot sensors 0 through 5 respectively (front – see figure 2).

After performing tests with the mobile robot we concluded that it is appropriate to consider that the robot was stuck if at least one of the distances \( d(0), \ldots, d(5) \) increases above 700, 400, 900, 900, 400, 700 respectively.

With respect to the motors, the simulated robot simply moves accordingly to the speed set by the algorithm of the user program. A random noise of ±10% is added to the amplitude of the motor speed, and a ±5% random noise is added to the change of direction (angle) resulting from the difference of speeds of the motors.

In our experiments, the learning controller was implemented using the C programming language. The controller was implemented in a process separate from the simulator, and communicates with it via the pipe interprocess communication mechanism.

5. Simulation Results

In this section we present a simulation example, regarding the application of the algorithm presented in sections 2 and including the improvements of section 3. The problem to be solved is to find a path to a predefined goal region. The simulation consisted of a sequence of trials. Concerning Algorithm 3 we used, \( N_1 = 1, N_2 = N_3 = 2 \), and Option 1 of Rule 2. On each trial, the robot started on a given position and tried to reach a goal region. The system has no knowledge about the location or the shape of obstacles. In our example the world has a great number of walls (see figures 3 to 7). This results on the possible paths to the goal being very refined. While learning a world representation, and a path to the goal, the robot uses its sensors to detect obstacles on the way. From trial to trial, the system is always accumulating knowledge about the world. As described in section 2, this knowledge is stored using the appropriate representation of a partition of the state-space and an outcomes database \( D \).

On figures 3, 4, and 5, we can observe the state-space partitions that resulted at the end of trials 1, 2, and 6 respectively. As can be seen, the algorithm is clearly able to incrementally construct a partitioning world-representation, that is suitable for the simultaneous implementation of its own learning capabilities. The robot, in an attempt to find a path to the goal, clearly increases the partition's resolution on areas where it faces greater difficulties to navigate. As obviously expected, these difficulties arise on areas that are close to obstacles. In contrast, on free-space areas, there is no need to increase the resolution that therefore is kept lower.

As can be seen on figure 6, after starting without any knowledge about the world, the robot performs a
considerable amount of exploration on the first trial. However, the robot is already able to reach the goal on this trial. The exploration continues on the succeeding trials, and at the beginning of the sixth trial, the robot has learned all the knowledge required to promptly navigate, through an obstacle-free path to the goal - see figure 7.

For assessing the effectiveness of the algorithm modifications proposed in section 3, we have selected six performance indexes. The first is the Number of cells. In some sense, we can state that, for the same environment, it is a measure of the difficulties the algorithm faced for modelling the world. When comparing two algorithms, it is also a measure of the "over-partitioning" effort of one of them. It is in general also true that with a greater number of cells we have higher computational costs. The second index, the Steps, is the number of time intervals, the robot was advancing in straight-line to a certain point (the centre of an aimed cell). Since we used a constant robot velocity, it is clearly a measure of the robot travelling distance. It does not include the time steps used at the beginning of each aim, in order to point the front of the robot towards the centre of the aimed cell. The third index, the Aims, is the number of times the robot aimed a neighbouring cell. It is a measure of both the exploration effort, and the computational costs. At a level more abstract than the steps, it is also a measure, of travelling effort. The fourth index, Aim Fails, is the percentage of aims that failed, because either the robot became stuck due an obstacle, or it attained a cell other than the aimed one. It is a measure of the exploration difficulties. The fifth index, the Number of minimax, is the number of times the minimax problem of equations (1) and (2) was solved. It is a measure of the important proportion of computational cost, that must be used for solving those problems. Finally, the CPU Time, strictly represents the time used by the learning algorithm presented in sections 2 and 3. It does not include the robot simulation time, or the interprocess communication time.

In order to analyse the benefits introduced on the algorithm, by the modifications of section 3, we collected the performance indexes on simulations with and without the modifications. Table 2 presents the indexes for the simulation with the modifications. This corresponds to the simulation we have presented on figures 3 to 7, and associated discussion text above. Table 1 presents the indexes for the same simulation (same world, start position, goal region) but without modifications of section 3. In both tables, and for all indexes, both the "trial-cumulative" and "trial-differential" figures are presented. As can be seen the modifications clearly improved the behaviour of the algorithm. In fact all the indexes have improved with the modifications. The Number of Cells have decreased to less than a half, reflecting lower world modelling difficulties and less computational costs. The
Table 1. Simulation Results – without Algorithm Modification

<table>
<thead>
<tr>
<th></th>
<th>Trial 1</th>
<th>Trial 2</th>
<th>Trial 3</th>
<th>Trial 4</th>
<th>Trial 5</th>
<th>Trial 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cells</td>
<td>Σ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>324</td>
<td>334</td>
<td>490</td>
<td>490</td>
<td>490</td>
<td>490</td>
</tr>
<tr>
<td></td>
<td>Δ</td>
<td>10</td>
<td>156</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Steps</td>
<td>Σ</td>
<td>103383</td>
<td>117312</td>
<td>129150</td>
<td>149291</td>
<td>158898</td>
</tr>
<tr>
<td></td>
<td></td>
<td>133929</td>
<td>11838</td>
<td>20141</td>
<td>9607</td>
<td>7798</td>
</tr>
<tr>
<td>Aims</td>
<td>Σ</td>
<td>1022</td>
<td>1174</td>
<td>1274</td>
<td>1641</td>
<td>1722</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1022</td>
<td>152</td>
<td>100</td>
<td>367</td>
<td>81</td>
</tr>
<tr>
<td>Aims</td>
<td>Σ</td>
<td>48.34</td>
<td>46.59</td>
<td>45.05</td>
<td>45.03</td>
<td>44.08</td>
</tr>
<tr>
<td></td>
<td></td>
<td>48.34</td>
<td>34.87</td>
<td>27.00</td>
<td>44.96</td>
<td>24.69</td>
</tr>
<tr>
<td>Aim Fails (%)</td>
<td>Σ</td>
<td>1040</td>
<td>1129</td>
<td>1296</td>
<td>1664</td>
<td>1746</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1040</td>
<td>154</td>
<td>102</td>
<td>368</td>
<td>82</td>
</tr>
<tr>
<td>Number of minimaxs</td>
<td>Σ</td>
<td>39.84</td>
<td>49.82</td>
<td>56.97</td>
<td>84.90</td>
<td>93.72</td>
</tr>
<tr>
<td></td>
<td></td>
<td>39.84</td>
<td>9.58</td>
<td>7.15</td>
<td>27.93</td>
<td>8.82</td>
</tr>
<tr>
<td>CPU Time (unit)</td>
<td>Σ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Steps also decreased indicating a lower robot travelling distance. The number of Aims also decreased reflecting lower exploration effort and lower computational costs. Additionally, among the lower number of Aims that were performed, the percentage of Aim Fails also decreased, confirming lesser exploration difficulties. The Number of minimaxs problems solved also decreased, indicating again a decrease of computational effort. Finally, the CPU Time was strongly decreased to less than a half, due to the introduction of the modifications. The “quality” of the final solution obtained on both cases, as measured in terms of a final travelling path of approximately 7500 steps on both cases, was quite similar. However, the modifications allowed decrease of the modelling difficulties, the exploration effort, and the computational costs, to achieve the solution. This same analysis was performed on other different world environments, and the same results were consistently observed, i.e. the algorithm modifications lead to an improvement on the performance of the algorithm.

6. Conclusion

In this article we have described the use of a learning algorithm for obtaining a map of a mobile robot world. There is no initial information regarding the location and shape of obstacles. Additionally, the algorithm is able to learn to navigate the robot from a start position to a goal region. In section 3, original modifications were proposed for improving the algorithm operation. Simulation results were presented, that show the effectiveness of the approach to the modelling of the world, and to finding a path to the goal. Simulation results also shown the benefits made possible by the algorithm modifications that were proposed. While exploring the world, the robot uses its distance sensors to detect obstacles.

References

RecPad'97
9th PORTUGUESE CONFERENCE ON PATTERN RECOGNITION

CONFERENCE PROCEEDINGS

Sponsored by

APRP
Department of Electrical Engineering - University of Coimbra
ISR - Instituto de Sistemas e Robótica
IT - Instituto de Telecomunicações
FCTUC
Fundação Calouste Gulbenkian
JNICT

March 20-21, 1997,
Department of Electrical Engineering,
University of Coimbra,
Coimbra, PORTUGAL