A Data Driven Algorithm for the Crowdsourcing of Behaviors for Autonomous Agents

S. D'Alessio (supervised by E. Garrabè, G. Russo)

Abstract

This work concerns the realization of a data driven algorithm able to provide an agent with the path to follow in order to reach a goal. The agent basically has its own default behavior and there are a number of contributors representing the possible directions in which the agent can go. Our work starts from a predefined situation and gradually generalizes.
Chapter 1 will be dedicated to expose the problem of crowdsourcing, consequently the theoretical notions and the algorithm on which the rest of the work is based will be treated.
In **chapter 2** we will see the agent moving on a fixed grid.
In **chapter 3** we will see the agent move on a square grid of any size with a series of optimizations on the path taken.
In **chapter 4** the algorithm is further optimized to improve performance which will in turn be analyzed in **chapter 5**.
Finally, in **chapter 6** a description of the graphical interface will be provided, which shows the movement of the agent on the field on screen.
It should be noted that in every version of the algorithm the agent's movements are always along the axes and never diagonally.
The algorithm is implemented in the Python programming language.

Contents

Chapter 1

Introduction to the problem of Crowdsourcing

Let us analyze a typical problem of an agent moving on a square grid of a certain size. The agent given an initial position must reach the target position trying to make as few moves as possible. However, some positions on the grid may not be able to be crossed by the agent, given the presence of obstacles, so the algorithm must be able to calculate a path physically feasible by the agent that may differ from the default behavior. In Figure 1.1 we can see the difference between the ideal agent behavior, without using the algorithm, and the current behavior using the algorithm.

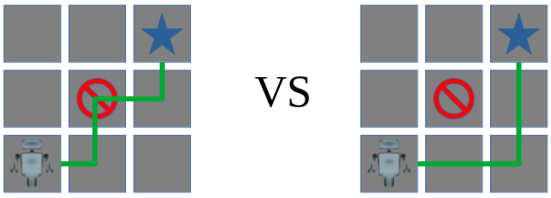


Figure 1.1: Difference between ideal behavior and real behavior

1.1 Theoretical bases

The algorithm used is extracted from [1] in which three factors are considered

Target Behavior Given the current position of the agent, the target behavior indicates which is the next cell to reach having a specific target as its destination. We let $\mathbf{x}_k \in X$ be the state of the agent at time k, $\mathbf{d}_{0:N} := \{\mathbf{x}_0, ..., \mathbf{x}_N\}$ be a dataset illustrating a target behavior for the agent over the time horizon $T := 0 : N$ and $p(\mathbf{d}_{0:N}) = p(\mathbf{x}_0, ..., \mathbf{x}_N)$ be the joint pdf (or mdf) obtained from the observed data. We make the standard assumption that the Markov property holds. Then, following the chain rule for pdfs/mdfs, we have

$$p_{0:N} = p_0(\mathbf{x}_0) \prod_{k=1}^{N} p_k(\mathbf{x}_k|\mathbf{x}_{k-1}) = p_{0:0} \prod_{k=1}^{N} p_{k|k-1} \tag{1.1}$$

which is simply termed as target behavior in what follows.

Agent-Specific Reward The goal of the agent is that of tracking the target behavior while, atthe same time, optimizing an agent-specific reward function. We let $r_k : X \to R$ be the reward obtained by an agent for being in state $\mathbf{x}_k$ at time k. Hence, the expected reward obtained by the agent that follows the behavior is given by

$$E_{\pi_{k-1:k-1}}[\tilde{r}_k(\mathbf{X}_{k-1})] := E_{\pi_{k-1:k-1}}[E_{\pi_{k|k-1}}[r_k(\mathbf{X}_k)]] \tag{1.2}$$

Contributors We let $\mathrm{S} := 1 : \mathrm{S}$ be the set of contributors from which the agent seeks to crowdsource its behavior. In what follows, $\{\Pi^{(i)}_{k|k-1}\}_{1:N}$ is the sequence of pdfs/mdfs (i.e., the behavior) provided by the i-th contributor.

Algorithm 1 is what our solution is based on.

Algorithm 1 Algorithm of crowdsourcing

1: **Inputs:** time-horizon τ, target behavior $p_{0:N}$, reward $r_k(\cdot)$, contributors behaviors $\pi^{(i)}_{k|k-1}$
2: **Output:** $\{\tilde{\pi}_{k|k-1}\}_{1:N}$
3: **Initialize:** $\mathbf{a}^T_{N+1}(x_N) \leftarrow 0$ and $\alpha^*_{N+1} \leftarrow 0$
4: **Main loop:**
5: **for** $k = N$ to 1 **do**
6: $\quad \hat{r}_k(\mathbf{x}_k) \leftarrow -\mathbf{a}^T_{k+1}(\mathbf{x}_k)\ \boldsymbol{\alpha}^*_{k+1}$
7: $\quad \bar{r}_k(\mathbf{x}_k) \leftarrow r_k(\mathbf{x}_k)\ \hat{r}_k(\mathbf{x}_k)$
8: $\quad$ **for** $i = 1$ to S **do**
9: $\quad\quad \alpha^{(i)}_k(\mathbf{x}_{k-1}) \leftarrow D_{KL}(\pi^{(i)}_{k|k-1}||p_{k|k-1}) - E_{\pi^{(i)}_{k|k-1}}[\bar{r}_k(\mathbf{X}_k)]$
10: $\quad$ **end for**
11: $\quad j \leftarrow$ index correspondings to smallest $\alpha^{(i)}_k(\mathbf{x}_{k-1})$
12: $\quad \boldsymbol{\alpha}^*_k \leftarrow [\alpha^{(1)}_k, \ \ldots, \alpha^{(S)}_k]^T,\ \alpha^{(i)}_k = 0,\ \ \forall i \neq j,\ \alpha^{(j)}_k = 1$
13: $\quad \tilde{\pi}_{k|k-1} \leftarrow \pi^{(j)}_{k|k-1}$
14: **end for**

As we can see, in line 9 of the algorithm, for every source we need to compute the Kullback-Leibler divergence.
Consider two probability distributions p and q. Usually, p represents the data, the observations, or a probability distribution precisely measured. Distribution q represents instead a theory, a model, a description or an approximation of p. For distributions p and q of a continuous random variable, the KL divergence is defined to be the integral:

$$D_{KL}(p \,||\, q) = \int_{-\infty}^{+\infty} p(x)\ \log\left(\frac{p(x)}{q(x)}\right)\ d(x) \tag{1.3}$$

That in the case of a discrete distribution can be implemented as follows:

$$D_{KL}(p \,||\, q) = \sum_{x \in X} p(x)\ \log\left(\frac{p(x)}{q(x)}\right) \tag{1.4}$$

Chapter 2

First version of the algorithm

In the first version of the algorithm the agent was able to move only on a following 3x3 square grid shown in Figure 2.1

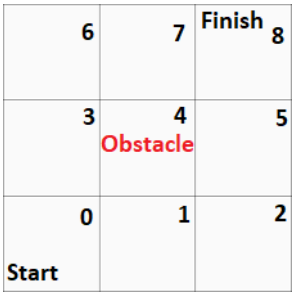


Figure 2.1: Board compatible with this version of the algorithm

2.1 Implementation of the algorithm

In python we define two object:

- An **agent** that represent the entity capable to move and to reach the goal. It is characterized by a name, a position and a board. It also contains the list of contributors in the implementation named sources.
- A **board** that is implemented as a matrix (a grid). It is characterized by a dimension, which corresponds to the number of rows or columns, and the value of each cell

There are two utility methods: ***_to_coordinates*** converts the cell index into two-dimensional coordinates on which the algorithm works while ***_convert*** allows you to report the coordinates in the index format. They are both private methods used by the algorithm and therefore are transparent to the user.

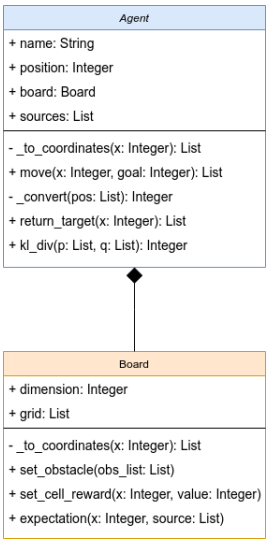


Figure 2.2: UML of classes

2.1.1 Implementation of contributors

Based on a square grid, the contributors (or sources), represented as a discrete probability mass function, are 5 and represent the possible movements of the agent on the grid: stay in the current cell, go left, go up, go right or go down. Each contributor is therefore a list of 5 elements containing the probability of going in the direction defined by the index of the list. The contributors are defined as follows:

- [0.95, 0.1, 0.1, 0.1, 0.1] where the maximum probability value is at index $i = 0$ and corresponds to **stay** in the same position
- [0.1, 0.95, 0.1, 0.1, 0.1] where the maximum probability value is at index $i = 1$ and corresponds to going **left**
- [0.1, 0.1, 0.95, 0.1, 0.1] where the maximum probability value is at index $i = 2$ and corresponds to going **up**
- [0.1, 0.1, 0.1, 0.95, 0.1] where the maximum probability value is at index $i = 3$ and corresponds to going **right**
- [0.1, 0.1, 0.1, 0.1, 0.95] where the maximum probability value is at index $i = 4$ and corresponds to going **down**

2.1.2 The Agent

The agent features the following methods:

- ***return_ target*** is the target behavior
- ***kl_ div*** compute the Kullback-Leibler divergence
- ***move*** is the main method that take in input the position of the agent and the final position, that in our case are fixed respectively to 0 and 8 due to the target behaviour. This method move the agent of only one cell and returns the pmf corresponding to the movement

Considering as x the index of the cell where the agent is located then the agent behavior target is defined in the Algorithm 2. The value returned by the target function is always a probability mass function which in our specific case coincides with the contributors.

Algorithm 2 Target function

Input: Current position of the agent x
if $x == 0$ **or** $x == 3$ **or** $x == 4$ **or** $x == 6$ **or** $x == 7$ **then**
    return [0.1, 0.1, 0.95, 0.1, 0.1]
else if $x == 1$ **or** $x == 2$ **or** $x == 5$ **then**
    return [0.1, 0.1, 0.1, 0.95, 0.1]
else
    return [0.95, 0.1, 0.1, 0.1, 0.1]
end if

The calculation of the KL is defined in Algorithm 3 where p is the sequence of pdfs/mdfs provided by the i-th contributor and q is the target behavior.

Algorithm 3 Computation of the Kullback-Leibler divergence

Inputs: p, q
$sum \leftarrow 0$
for $i = 0$ in p **do**
    $sum \leftarrow sum + p[i] \cdot \ln \frac{p[i]}{q[i]}$
end for
return sum

2.1.3 The Board

The board features the following methods

- ***set_ obstacle*** takes as input a list of cell and put in this cells a big negative value
- ***set_ cell_ reward*** take as input a cell and a value to put in update based on the time_step, at every call, increments the value of the negative cells
- ***expectation*** compute the expectation designed in the algorithm: takes as input a position and a source and returns a single value

The expectation is a method that iterates through all sources and checks the value present in the cell indicated by the respective source. All found values are added together. if the source leads to an invalid position such as it can be for example outside the grid, then a large negative value is added.
The calculation of the expectation is done by following the Algorithm 4, so at the end ot this method we have a value that will be used by the move.

Algorithm 4 Calculation of the expectation

Inputs: The list of sources S and the current position of the agent $curr$
for s in S **do**
    $pos \leftarrow curr$
    $sum \leftarrow 0$
    if s corresponds to $left$ **then**
        $pos \leftarrow$ index of the cell to the left
    else if s corresponds to up **then**
        $pos \leftarrow$ index of the cell above
    else if s corresponds to $rigth$ **then**
        $pos \leftarrow$ index of the cell to the right
    else if s corresponds to $down$ **then**
        $pos \leftarrow$ index of the cell under
    end if
    if pos is valid **then**
        $sum \leftarrow sum +$ value of the cell in pos
    else
        $sum \leftarrow sum +$ big negative value
    end if
end for
return sum

2.2 Definition of the reward

Values stored in the cells are according to the concept of reward. We have seen that obstacles can be present on the grid and since we do not want the agent to end up in a position with an obstacle it is logical to assign a large negative reward to these cells in order to keep the agent away from them.
In our work the following implementation choices were made

- For an obstacle is set -100 as value of the cell
- The value of the next cell to be reached will be decreased by 10
- The value of the cell just left is incremented by 1
- If the cell is outside the board we assign -100

So, before we move to the next cell we assign at this cell a negative value because we don't want to stay there and we give a little increment at the past cell so in case of doubt to stay or go back we choose to go back.
In Figure 2.3 we can see ax example of reward in a more general case.

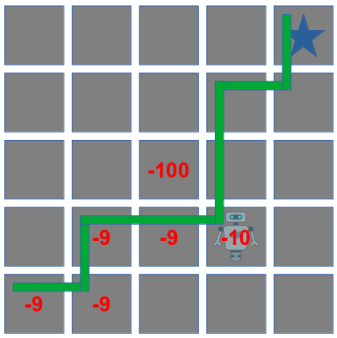


Figure 2.3: Example of implementation of the reward in a more general grid

2.3 The method move

Move is the main method of the algorithm because this is what determines the agent's movement.
The method starts with a cycle

for $i = 0$ in S **do**
$v[i] \leftarrow$ ***kl_ div***$(S[i],\ target\_behavior) -$ ***expectation***$(current\_position,\ S[i])$
end for

After this cycle we have a vector of dimensions equal to the number of sources which contains the calculated differences. The D_{KL} tells us how similar are the probability distributions taken in input so a small value is an index of similarity. From this we subtract the expected value calculated according to the previous procedure: a negative expected value represents a penalty and therefore subtracting a negative value and as if we were going to add the module to the value of the D_{KL}. From this it is clear that the smaller the final value obtained, the better the choice.
We then take the first index relative to the smallest value of the vector

$i \leftarrow$ first index corresponding to the smallest value of v

Based on this index we make the movement with respect to the definition of the reward.

$set\_cell\_reward\,(+1)$
$pos \leftarrow$ new position that depends on i
$set\_cell\_reward\,(-10)$

After that the agent has changed its position, and the values of the cells have been set.

Chapter 3

Second version of the algorithm

With the second version of the algorithm, no changes were made to the Board object. The agent, on the other hand, has been modified in the move method which is now divided into two phases: one of exploration and another of movement. A path optimization phase has also been added.

3.1 The Agent

The agent has the knowledge of the entire space but the algorithm in every step sees only a part of it; it depends on how much the algorithm is able to look forward from a given position. In figure 3.1 we see the updated diagram.

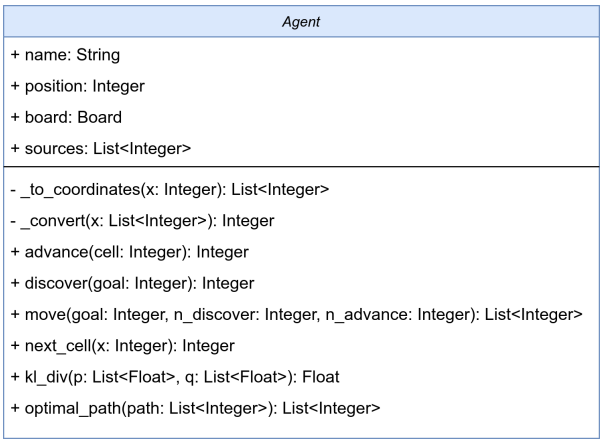


Figure 3.1: Example of implementation of the reward in a more general grid

Compared to the previous version, the following methods have been added/changed

- ***optimal_path*** crop the path in input to remove self-loop
- ***next_cell*** is the new and more general target function

- ***discover*** is a method that move the agent on the board. It takes as input a goal cell and return the next position of the agent in the direction to reach the goal
- ***advance*** is similar to *discover* but move the agent to the next adiacent cell given in input
- ***move*** is the main method, it calls *discover* for n_steps time on a copy of the grid and build a path. The output of the discover is optimized by *optimal_path* and after, *move* calls the method *advance* a number of times equal to the n_advance parameter that can be max equal to the length of the optimized path

3.2 The phase of discovery

The discovery phase can be seen as a kind of dream in which the agent explores the surrounding space for an indicated number of steps. During this phase the agent moves on a copy of the surrounding environment. The implementation of the method is shown in the Algorithm 5

Algorithm 5 Pseudocode of the discover

```
1: Input: The board and the goal position
2: if pos == goal then
3:     return goal
4: end if
5: if value of pos in board is less than SAT then
6:     return −1
7: end if
8: behavior ← next_cell(pos, goal)
9: v ← [ ]
10: for i = 0 in S do
11:     v[i] ← kl_div(S[i], behavior) − expectation(pos, S[i])
12: end for
13: j ← first index corresponding to the smallest value of v
14: for i = 1 in S do
15:     if j == i then
16:         set_cell_reward(pos, +1)
17:         pos ← new position that depends on i
18:     end if
19: end for
20: set_cell_reward(pos, −10)
21: return pos
```

The discovery phase first checks whether the current position corresponds to the target cell, if so it returns that position directly without performing other operations. If, on the other hand, the cell is different from that target, the algorithm proceeds to check whether the current position has not reached a saturation value (indicated by SAT in the pseudocode). If there are no saturation conditions the algorithm continues performing operations similar to the move

method described in **Section 2.3**, then the current position is updated and returned.

3.2.1 The saturation check

The saturation was introduced to avoid anomalous behavior of the algorithm, in particular in the case in which the target cell is not physically reachable. In fact, in this case, if there was no saturation check, the algorithm would cross the obstacles or exit the grid: the agent would move repeatedly in the space allowed to him, increasing more and more the penalty associated with the cells he crosses up to the point where these cells assume a greater penalty than the obstacle itself, inducing the agent to cross it. In Figure 3.2 we can see the difference in the path taken by the agent by introducing saturation or not.

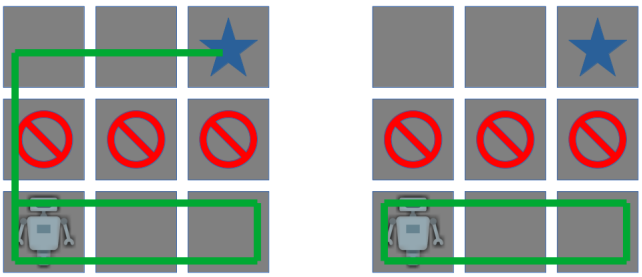


Figure 3.2: Absence of saturation vs. presence of saturation

Ultimately, therefore, saturation serves to understand if the path to the goal is physically feasible.
Different values for saturation are admissible taking into account that it must be greater than the value of an obstacle and less than 0. In particular:

- A large saturation value corresponds to a good chance of finding the path to the target cell if it exists, even if there are many obstacles in the way. However if the path does not exist the agent could make a lot of unnecessary movements
- If the saturation value is smaller, the chances of finding the path to the goal decrease if there are many obstacles on the grid. However, the advantage of this situation is that the agent is avoided unnecessary movements in the case of an unreachable goal

In our project a fixed value is used for the saturation threshold set at -29 which, following the policy adopted for the allocation of the reward, is equivalent to saying:

- "The agent cannot stay on the same cell for more than three consecutive steps"
- "The agent cannot pass through the same cell more than four times"

Studies on the optimal saturation value are not the subject of this work.

3.3 The phase of advancement

The advancement phase is very similar to the discovery phase except for the fact that it receives as input the adjacent cell to move to. This phase acts on the real grid and the values set by the method are visible to future invocations of the discover that act on a current copy of the grid. The advance therefore leaves the history of the actual path taken by the agent as it physically moves the agent. In the Algorithm 6 we can see that its behavior is common to discover.

Algorithm 6 Pseudocode of the advance

```
1: Input: The index of the near next cell next
2: for i = 1 in S do
3:     if next == i then
4:         set_cell_reward(pos, +1)
5:         pos ← new position that depends on i
6:     end if
7: end for
8: set_cell_reward(pos, −10)
9: return pos
```

3.4 The method move

The move method exploits the presence of the two methods presented previously: it first calls the discover on a copy of the grid for a certain number of times, building a sort of temporary path which, after being optimized, it uses to invoke the advance. The pseudocode is shown in Algorithm 7.
The behavior of the method can be described in the following steps:

1. For first we check if the current position is the goal. If it is, we return a list containing only the goal position, otherwise the algorithm continues
2. We store the current position
3. We make a copy of the board
4. We call the discovery n_{adv} times on the copy of the board and we build a path
5. We save the next cell indicated by the path
6. The path is optimized by the function optimal-path removing self-loop (how will be explained in the subsection)
7. If the optimization removes all the next position we have only a path with the current position so we decide to move to the next position stored previously
8. If the path contains other position in addition to the current we call the method advance for every next position present in the path
9. At the end we return the path of the crossed position

Algorithm 7 Pseudocode of the move

Input: The index of the goal cell *goal*, the number of steps of the advance n_{adv} and the number of steps of the discovery n_{dis}
Output: Returns the list of cells crossed [i, i + 1, ..., i + n]
if *pos* == *goal* **then**
    return [*goal*]
end if
$x \leftarrow pos$
$forward\_path = [x]$
board ← copy of the current state of the board
while $n_{adv} > 0$ **do**
    $pos \leftarrow$ ***discover***(*goal*, *board*)
    if $pos == -1$ **then**
        return -1
    end if
    $forward\_path$.***append***(pos)
    $n_{adv} = n_{adv} - 1$
end while
$next\_cell \leftarrow forward\_path[1]$
$forward\_path \leftarrow$ ***optimal_path***($forward\_path$)
$pos \leftarrow x$
if the lenght of $forward\_path$ is lower than 2 **then**
    $forward\_path$.***append***($next\_cell$)
    advance($next\_cell$)
    return $forward\_path$
end if
$forward\_path \leftarrow forward\_path$ without the first position
$path\_lenght \leftarrow$ number of elements in $forward\_path$
if $n_{adv} > path\_lenght$ **then**
    $n_{adv} \leftarrow path\_lenght$
end if
for $i = 0$ to n_{adv} **do**
    advance($forward\_path[i]$)
end for
return $forward\_path$ up to the index n_{adv}

3.4.1 The optimization of the path

Calling discover n_{adv} times, the constructed path is not necessarily optimized in fact it could happen that the same cell is present more than once in the same path: this corresponds to the fact that the agent makes a cyclic path. If we provide the advance with a path that presents cycles, the agent will make unnecessary movements. The advantage of having divided the movement of the agent into discover and advance lies precisely in this, namely the possibility of providing optimized output. In Figure 3.3 we can see how a self-loop is detected and cropped.

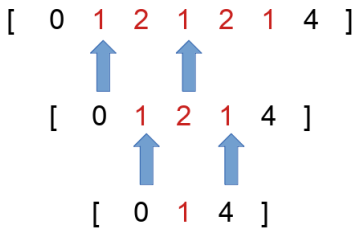


Figure 3.3: Removing a self-loop from a path

A particular case occurs when the path is a complete cycle, that is, the initial cell corresponds to the final cell. This behavior can be deduced by observing Algorithm 8. In this case the optimization cuts the path leaving only the initial (or final) cell which also corresponds to the position where the agent is currently located. This situation is solved by the move method in the previous paragraph.

Algorithm 8 ***optimization*** procedure

```
Input: The path to be optimized
len ← lenght of the path
final_path ← [ ]
k ← 0
while k < len do
    cell = path[k]
    if cell in final_path then
        i ← index of the first occurence of cell in final_path
        final_path ← final_path without elements after index i + 1
    else
        final_path.append(cell)
    end if
    k ← k + 1
end while
return final_path
```

3.5 The new target behavior

For this version of the algorithm the new target function ***next_cell*** is used which adapts to grids of various sizes. This function tends to bring the agent

back to a diagonal position with respect to the target position and then to continue on this diagonal. The function always returns a probability mass function indicating the probability of following one of the 5 previously defined directions. The new target behavior is described in Algorithm 9.

Algorithm 9 Target function ***next_ cell***

Input: The current position of the agent x and the goal position *goal*
$\Delta x \leftarrow$ distance between the target cell and the current cell on the x axis
$\Delta y \leftarrow$ distance between the target cell and the current cell on the y axis
if $\Delta x == 0$ **and** $\Delta y == 0$ **then**
    return [0.95, 0.1, 0.1, 0.1, 0.1]
end if
if $\Delta x > 0$ **then**
    if $\Delta x >= abs(\Delta y)$ **then**
        return [0.1, 0.1, 0.1, 0.95, 0.1]
    end if
end if
if $\Delta x < 0$ **then**
    if $abs(\Delta x) >= abs(\Delta y)$ **then**
        return [0.1, 0.95, 0.1, 0.1, 0.1]
    end if
end if
if $\Delta y > 0$ **then**
    if $abs(\Delta x) < \Delta y$ **then**
        return [0.1, 0.1, 0.95, 0.1, 0.1]
    end if
end if
if $\Delta y < 0$ **then**
    if $abs(\Delta x) < abs(\Delta y)$ **then**
        return [0.1, 0.1, 0.1, 0.1, 0.95]
    end if
end if

Chapter 4

Final version of the algorithm

This version of the algorithm improves the previous one expecially in the phase of optimization and introduces a forgetting factor in order to avoid saturation phenomena that could lead to deadlock.

4.1 Implementation of the algorithm

In the implementation we decided to generalize the methods for calculating the KL divergence and the expectation by making them external from their respective objects so that they can also be used as library methods. The updated architecture is the one shown in Figure 4.1.

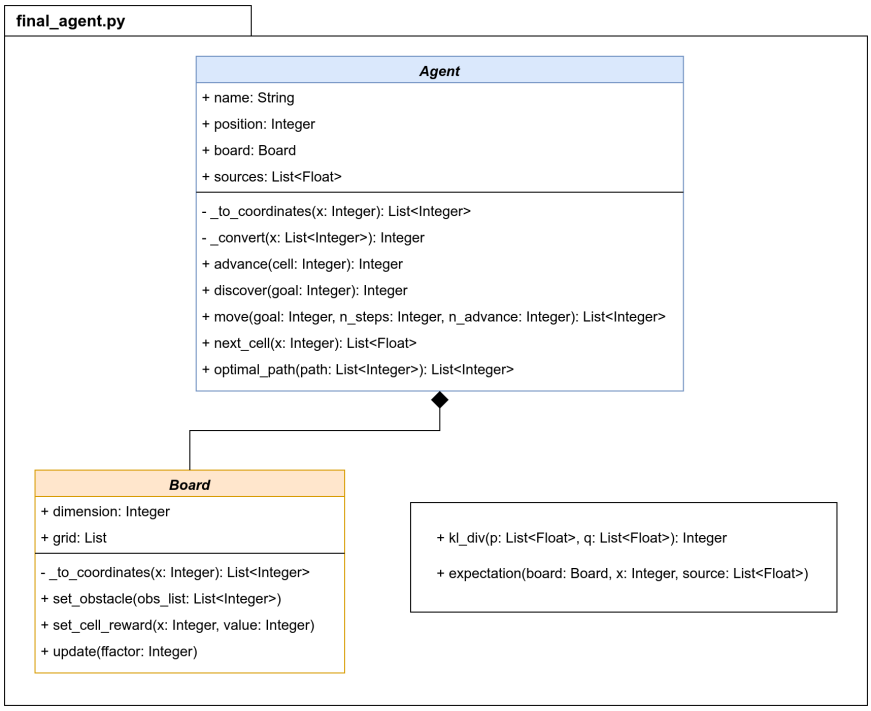


Figure 4.1: Complete UML diagram of the algorithm

4.2 A random choice in the discovery phase

As described in **Section 3.2** on line 13 of the algorithm, discover takes the first index of the minimum of the vector. Since, depending on the target function used, it could happen that there are more minimum values. In order not to always incur in the same choice, it was decided in the case of more minimums to randomly choose the index of one of them. So the line 13 of the ***discover*** becomes the following

13: $j \leftarrow$ ***find_min_index***(v)

The method **find_min_index** in the case of multiple min element stores indexes of this and return randomly one of this indexes

4.3 The improvement of the move

In the previous version of the algorithm, if a self loop was detected it was completely removed from the optimization but it was still decided to enter the loop having previously saved the first position of this. However, this behavior led the agent to perform two useless physical movements as one entered the cycle and subsequently returned to the initial position and then continued on another path. The design choice adopted in the case of a self loop then translates into staying still for one step, always assigning a negative value to the first cell of the cycle in that step so that the algorithm avoids entering it at the next call of the discovery.

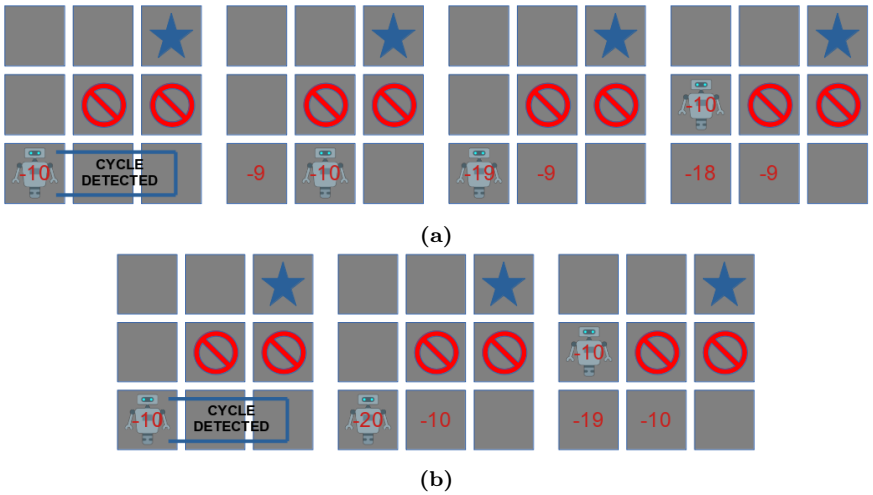


Figure 4.2: (a) Movement of the agent with the implementation of chapter 3 (b) Movement of the agent with the new implementation

A performance improvement has been made by checking the return value of ***discover***: if it corresponds to the target cell, the function is no longer called. In Figure 4.2 we can observe the difference between the two versions of the algorithm in terms of movement and reward assigned to the grid while the modification to the implementation of the method is visible in Algorithm 10.

Algorithm 10 Updated version of the ***move***

Input: The index of the goal cell $goal$, the number of steps of the advance n_{adv} and the number of steps of the discovery n_{dis}
Output: Returns the list of cells crossed [i, i + 1, ..., i + n]
if $pos == goal$ **then**
    return $[goal]$
end if
$x \leftarrow pos$
$forward\_path \leftarrow [x]$
$board \leftarrow$ copy of the current state of the board
while $n_{adv} > 0$ **do**
    $pos \leftarrow$ ***discover***$(goal, board)$
    if $pos == -1$ **then**
        return -1
    end if
    $forward\_path$.***append***$(next\_cell)$
    if $pos == goal$ **then**
        $n_{adv} \leftarrow 0$
    else
        $n_{adv} \leftarrow n_{adv} - 1$
    end if
end while
$next\_cell \leftarrow forward\_path[1]$
$forward\_path \leftarrow$ ***optimal_path***$(forward\_path)$
$pos \leftarrow x$
if the lenght of $forward\_path$ is lower than 2 **then**
    set_cell_reward$(next\_cell, -10)$
    advance(pos)
    return $forward\_path$
end if
$forward\_path \leftarrow forward\_path$ without the first position
$path\_lenght \leftarrow$ number of elements in $forward\_path$
if $n_{adv} > path\_lenght$ **then**
    $n_{adv} \leftarrow path\_lenght$
end if
for $i = 0$ to n_{adv} **do**
    advance$(forward\_path[i])$
end for
return $forward\_path$ up to the index n_{adv}

4.4 The improvement of the optimization

Experimentally it has been noted that cycles can form which start in a cell and end in an adjacent cell and then continue the path. These cases can be seen as a deviation from the path. A daily example could be that relating to a motorway: you exit a toll booth to reach a goal, however the road is blocked and you have to return to the next toll booth to continue on the motorway. Since discover can encounter situations of this type, it is advisable for the optimization phase to detect them. An example on the grid in our case is shown in Figure 4.3 where the deviation that the agent would make without this improvement in optimization is highlighted in red.

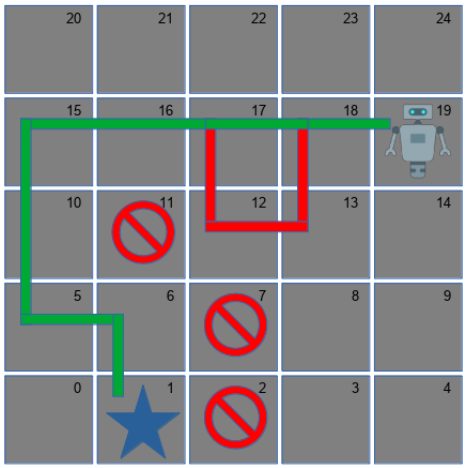


Figure 4.3: Optimization based on adjacency of two cells

With this new version of the algorithm, the case of self loop occurs only when the last cell of the path is already present previously; otherwise if there are two identical cells in the path and one of these is not the last, its adjacent cell will always be present. In Figure 4.4 you can see the two optimization cases.

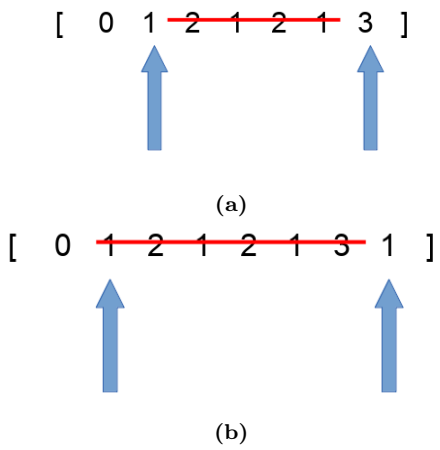


Figure 4.4: The case (a) is the most common one in which the elements between the two adjacent cells are eliminated. The second case (b), on the other hand, reveals a path that ends with a self-loop, therefore the cell considered as being equal to the last one is also deleted.

There is also an implementation difference in the algorithm; in fact in this new version the check of adjacent or identical cell is made starting from the right. This allows you to eliminate a cycle in a single step because the outermost cycle, which is the largest one, is eliminated. We can see the updated method in the Algorithm 11.

Algorithm 11 The updated ***optimization*** procedure

Input: The *path* to be optimized
final_path ← [ ]
while lenght of the path > 0 **do**
    cell = *path*[0]
    i = length of the path −1
    while *i* > 0 **do**
        if *cell* == *path*[*i*] **then**
            final_path ← *final_path* without elements before index *i*
            i = 0
        else if ***_is_near***(*cell*, *path*[*i*]) **then**
            final_path ← *final_path* without the elements between indices 1 and *i*
            i = 0
        else*i* ← *i* − 1
        end if
    end while
    final_path.***append***(*cell*)
    path ← *path* without the cell in the first position
end while
return *final_path*

4.5 The forgetting factor

In contexts where the grid presents many obstacles it could happen that the agent ends up in a sort of trap from which the simplest way to get out is to go back. However, the path already passed presents a negative reward and could act as an obstacle in turn. The implementation choice then was to add a forgetting factor which, based on the set value, gives greater importance to the most recent history. That factor at each step of the algorithm increases by a certain amount (1 by default) the values of the grid cells that are not obstacles (which they always keep a value of -100) up to a maximum value equal to 0. Figure 4.5 shows how the forgetting factor acts on the grid while in Algorithm 12 its implementation is shown under the name of the ***update*** method. It should be remembered that the *update* method is a method of the Board class so it can access the class attributes such as the *dimension* that represents the number of rows or columns of the grid.

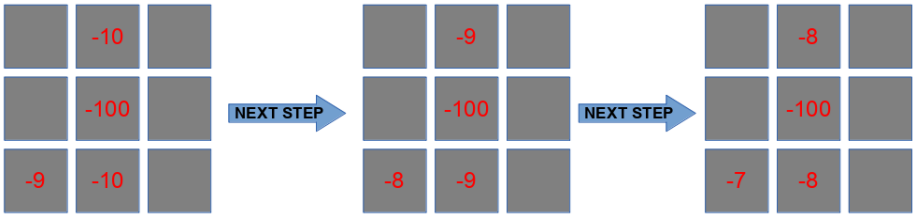


Figure 4.5: Effect of the forgetting factor

Algorithm 12 Pseudocode of the ***update***

```
Input: The value of the fogetting factor f
if f < 1 then
    return −1
end if
for i = 0 to dimension do
    for j = 0 to dimension do
        if grid[i][j] < 0 and grid[i][j] > −100 then
            if grid[i][j] < −f then
                grid[i][j] ← grid[i][j] + f
            else
                grid[i][j] ← 0
            end if
        end if
    end for
end for
```

Using the forgetting factor, however, could slow down the algorithm a lot, especially in the case of large grids because it requires the control of all the cells one by one. Entering 0 or a negative value as forgetting factor disables the option.

Chapter 5

Simulation analysis

The operating procedure for performing a simulation of the algorithm is presented here, showing the possible configurations by the user. Performance will also be analyzed.

5.1 Use of the algorithm

To run the algorithm you need to have python installed on the system, then just go to the folder where they are present in the file (in this case we refer to the latest version) and run the command

```
python agent_v1.5.py
```

after which the algorithm will ask us for some configuration parameters to be entered by asking the following questions

- *"Insert grid dimension (e.g. '3' for a grid 3x3):"*

  You enter a value that determines the size of the grid. For example, by inserting 5 a square grid is created consisting of 5 rows and 5 columns with the numbering of the cells starting from 0 at the bottom left up to 24 at the top right. The numbering takes place from left to right and from bottom to top. You can see the example of Figure 5.2

- *"Enter number of obstacles:"*

  First the algorithm asks how many obstacles are present on the grid, after which we must enter the position (cell index) of these obstacles

- *"Insert the initial position of the agent:"*

  We simply have to indicate which cell the agent is in

- *"Insert the desired final position of the agent:"*

  We must indicate which cell the agent wishes to reach

- *"Insert the resolution of the discovery:"*

  The resolution of the discovery determines how many steps forward the algorithm is able to look at starting from a given cell

- *"Insert how many cells advance max in each step:"*

  Given the path provided by the discovery and then optimized, here we decide how many cells to advance in this path

- *"Insert the value of the forgetting factor (0 to disable):"*

  This is the configuration of the forgetting factor

After completing the configuration phase, the algorithm calculates the path by making prints for each step that corresponds to a call to the ***move*** method. At the end it is said whether the goal is achievable or not. If the goal is achievable, the path and simulation time of the algorithm is shown.
At the end by pressing any key it is possible to start the graphic simulation described in **Chapter 6**.

5.2 Use cases

5.2.1 A simple illustrative example

First we want to analyze the advantage of using the discovery phase in the algorithm. In the following examples, the ***discover*** path is highlighted in blue, while the ***advance*** path is shown in green. The discovery, combined with the optimization phase, allows in some cases to avoid unnecessary journeys to reach the goal. This is visible in Figure 5.1 where the agent, after making the first choice to go right, avoids going there because the optimization on the discovery path says it is a self-loop. From the algorithm, the cycle input cell is marked with a negative reward as well as and the agent waits for a step in the current cell. After that, at the next invocation of the discovery, the path on the right is avoided and the one that leads up to the goal is preferred.

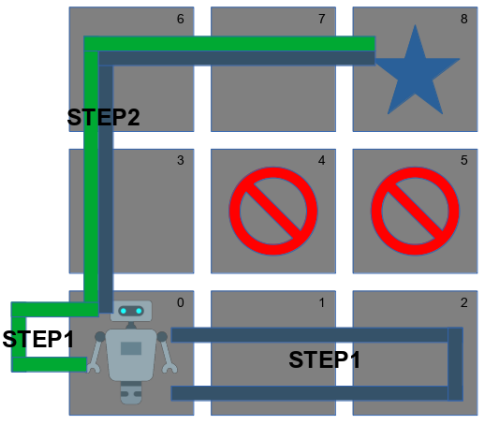


Figure 5.1: Simulation on a 3x3 grid with obstacles in position [4, 5], with discovery resolution of 4 and max advancement resolution

5.2.2 Changing the resolution of the discovery phase

Another simulation is present in Figure 5.2 where it can be seen that if the agent initially takes the wrong path, the resolution of the advance (in this case set to

5) is not sufficient to indicate to the agent to avoid it; the only improvement is that the optimization prevents the agent from going through cell 9. It should be noted that this does not always happen but only when the initial choice is to go to the right. In the example the probability of going up or going to the right is the same, and the algorithm, based on the modification made in **Section 4.2**, randomly chooses one of the two.

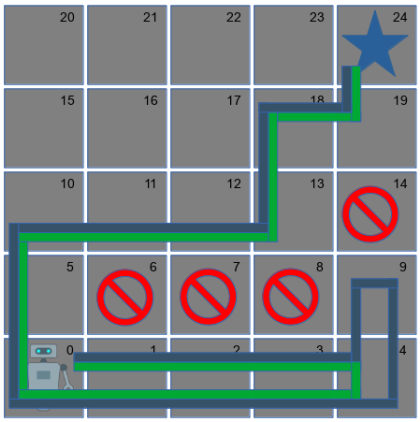
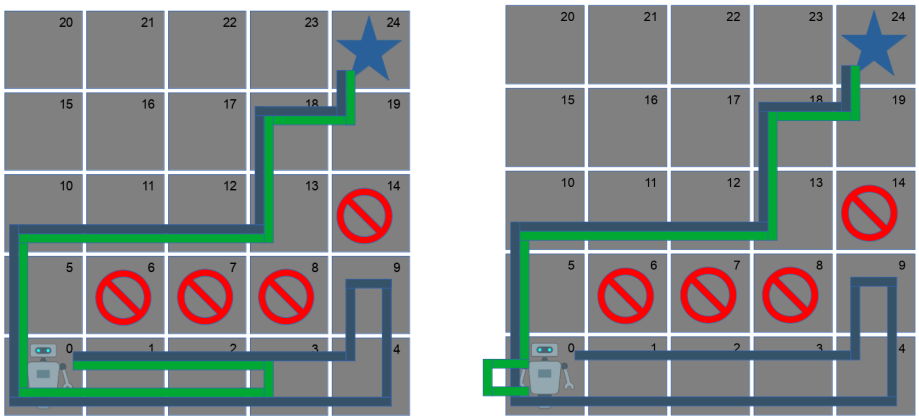


Figure 5.2: Simulation on a 5x5 grid with obstacles in position [6, 7, 8, 14], with discovery resolution of 5 and advancement resolution of 3

We can see from Figure 5.3 that by increasing the resolution of the discovery phase we can improve the path allowing the agent to move on a smaller number of cells.



(a) **(b)**

Figure 5.3: Simulation on a 5x5 grid with obstacles in position [6, 7, 8, 14], with advancement resolution of 3 and discovery resolution of 8 in case (a) and of 12 in case (b)

Another factor to take into account is the resolution of the advancement phase: the lower it is, the more times a discovery phase is started. Reducing this resolution corresponds to a more prudent move on the grid which could be

useful in the case of dynamic contexts which, however, are not the subject of this work.

5.2.3 Unreachable goal

As discussed in **Section 3.2.1** the saturation control allows us to understand if the target is not achievable. In the case of an unreachable goal, the saturation threshold determines how long it takes the algorithm to notice this. In our case this threshold is a fixed parameter so it is not possible to compare different executions of the algorithm. This study of the saturation threshold could be the subject of future developments. In Figure 5.4 we see two cases in which the goal is not achievable.

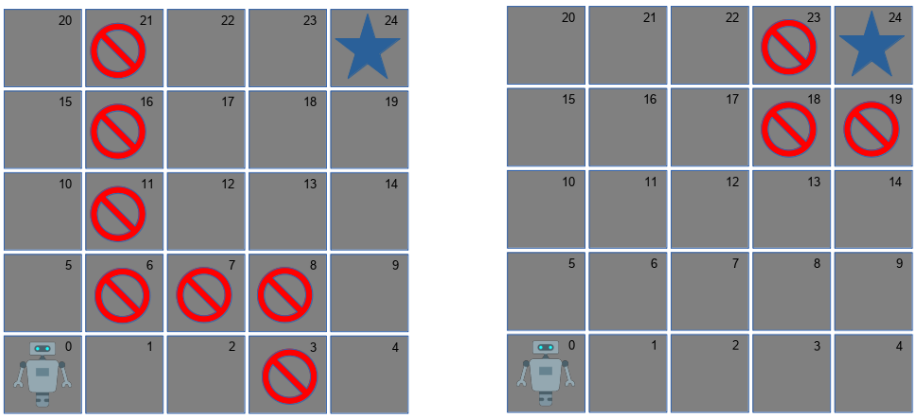


Figure 5.4: Simulation with discovery resolution of 5 and advancement resolution of 3. The simulation time in case (a) is 4.9545ms and in case (b) is 10.2725ms

It is easy to understand how in the case (b) the algorithm takes longer than the (a). Since the space in which the agent moves in the second case is greater than the first, the algorithm takes longer to bring a cell to saturation and this involves more movements and more computational time.

5.2.4 Using forgetting factor

As discussed in Section 4.5 the forgetting factor allows you to give more weight to recent history than to the past. From the computational point of view it is very heavy because it analyzes the grid cell by cell. In future developments, to improve performance, it could be thought of implementing a localized forgetting factor that is dependent on the current position of the agent and which analyzes only a portion of the grid around it determined by a configurable parameter. For representative effectiveness, the same grids as in Figure 5.4 are used, enabling the forgetting factor this time

- By setting the forgetting factor to 1 the simulation time in case (a) is 7.4607ms and in case (b) is 36.6547ms

- By setting the forgetting factor to 2 the simulation time in case (a) is 9.7807ms but in the case (b) the algorithm never terminates

The delay introduced by the forgetting factor is a constant value that depends on how many times we invoke the update function which in our implementation corresponds exactly to the number of invocations of the move. However, the forgetting factor itself influences the number of calls of the move because it modifies the values of the grid by gradually removing the information previously stored and, depending on the context, if it is too high, it prevents the algorithm from converging to a solution.

5.2.5 The effect of the randomness

In the example shown in Figure 5.5 we can see the randomness introduced into the algorithm by the ***find_min_index*** method. This randomness leads to different results in the same static context; this usually indicates that the target function is not the best. This especially happens when the KL divergence returns identical values for two or more contributors and in the directions indicated by these contributors the same expected value occurs (which could correspond to the fact that we have no values in those cells). In general, therefore, it can be said that it is not desirable for KL divergence to return identical values.

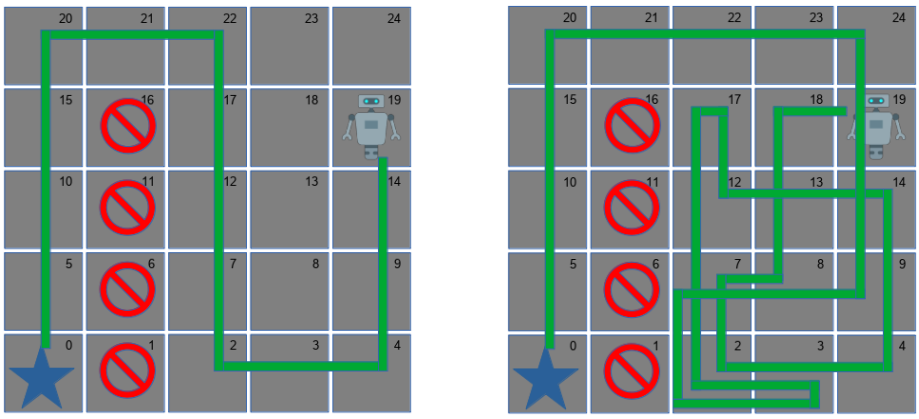
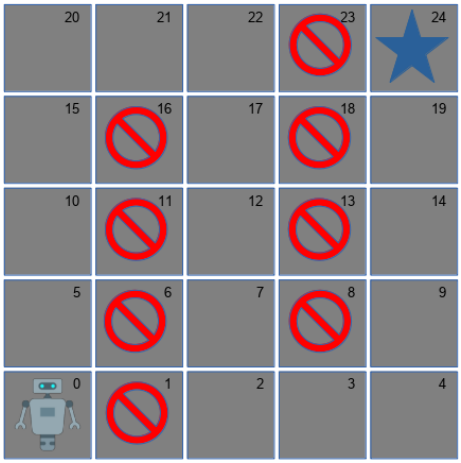


Figure 5.5: Simulation with discovery resolution of 6 and advancement resolution of 4. The simulation time in case (a) is 4.6631ms and in case (b) is 8.9935ms

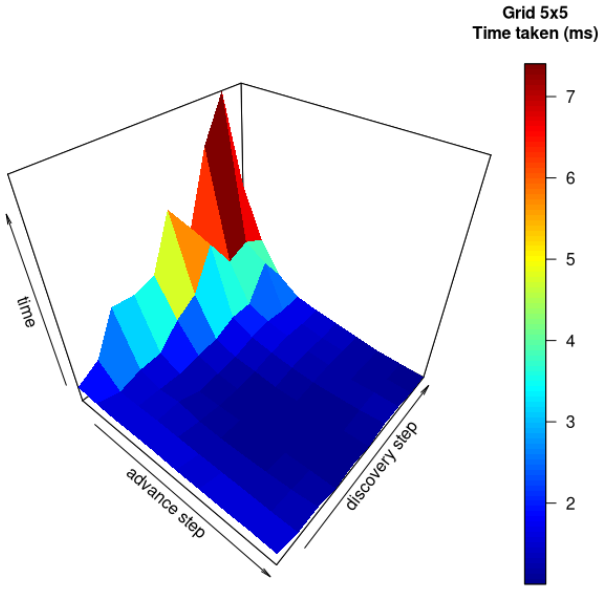
5.3 Computational aspects

In the previous examples we have noticed that the computation time depends on various configurable factors within the algorithm and above all on the static context: size of the grid, position of obstacles, starting point and end point. However, it has always been interesting to us, once a grid has been set, to observe the computation time as the resolution of the discovery and advancement phases vary. Figure 5.6 shows the example grid and a graph showing the execution time of the algorithm in the various configurations. In particular, the resolutions

were changed from 1 to 10 and all the combinations were analyzed. It should be noted that as indicated in Algorithm 10, if the resolution of the advancement phase exceeds the length of the available path, it is set to the maximum possible length.



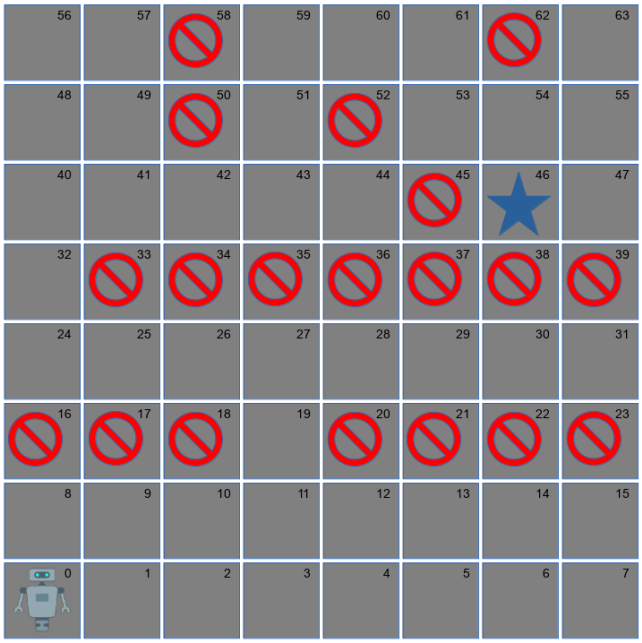
(a)



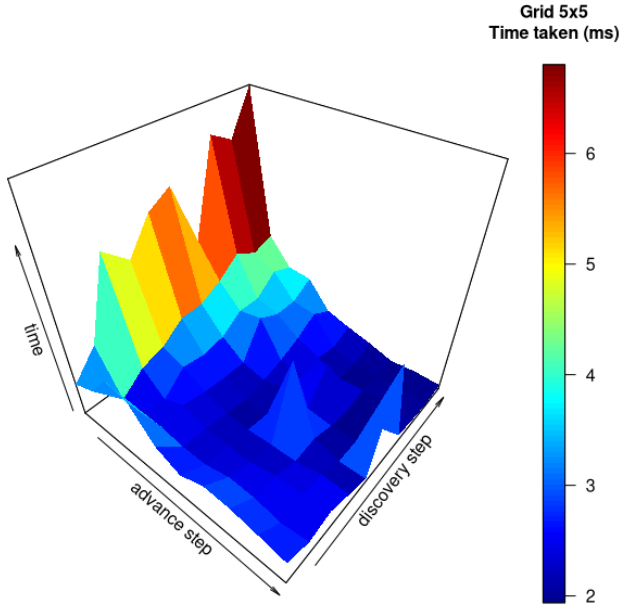
(b)

Figure 5.6: (a) The grid of the simulation. (b) The results of the simulation

As you can easily guess from the graph, having a very low resolution of the advancement phase means invoking the ***discover*** several times and therefore taking more time. However, as mentioned earlier, this ensures greater accuracy should the grid change over time. A more complex example of temporal simulation is present in Figure 5.7 where the effect of randomness is more visible in the time graph which manifests itself in the form of peaks.



(a)



(b)

Figure 5.7: (a) The grid of the simulation. (b) The results of the simulation

Chapter 6

Graphical interface

Our work is completed with a graphical interface that allows you to see the movement of the agent on the grid. This interface was created in Python using the pygame library [2]. The architecture is presented in Figure 6.1.

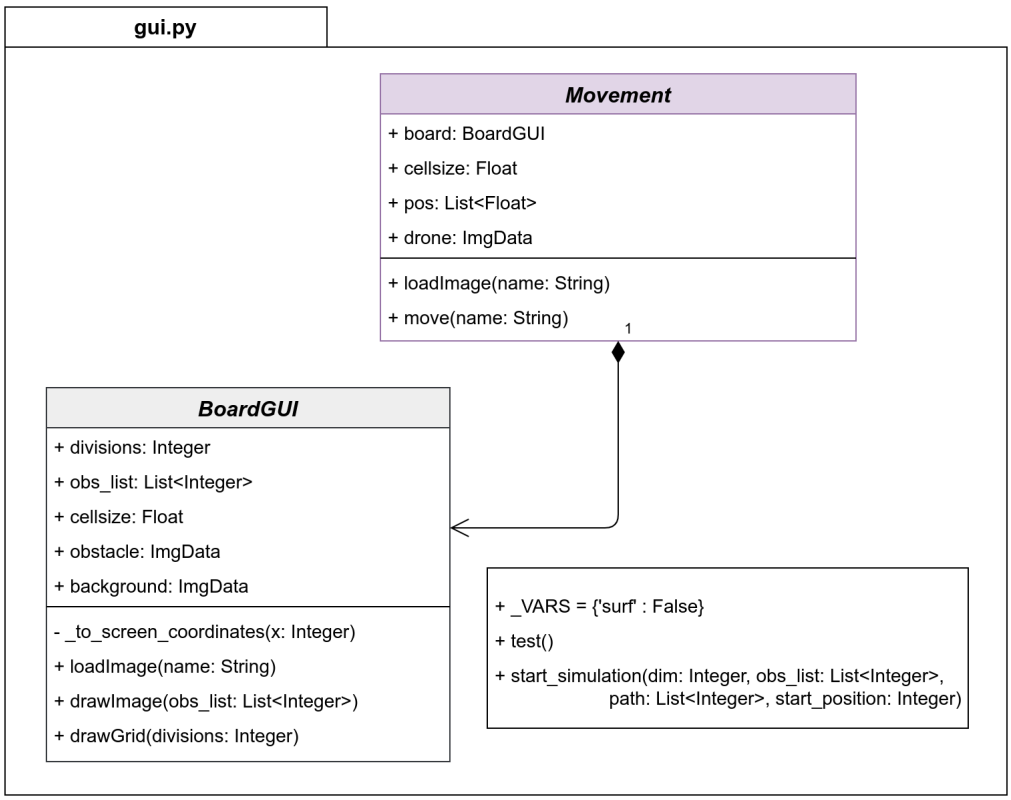


Figure 6.1: UML diagram of the graphical interface

As package methods we have ***test*** which is used to simulate the graphical environment with preset parameters and ***start_ simulation*** which is invoked by the crowdsourcing algorithm passing the following parameters in order:

- the size of the grid, that is the number of rows or columns of which it is composed
- a list containing the indexes of the cells in which there are obstacles
- the path calculated by the algorithm
- the agent's initial position

There are two classes in the package: one that deals with tracing the movement of the agent on the grid (**Movement**) and another that deals with defining the grid and its graphic appearance (**BoardGUI**). The display in which the graphical interface operates is contained in a shared dictionary accessible by both classes with the 'surf' key. Some global variables such as $SCREENSIZE$ which defines the window size or some color constants are defined in the package.

Listing 6.1: The method ***start_simulation***

```
def start_simulation(dim, obs_list, path, start_position):
  pygame.init()
  _VARS['surf'] = pygame.display.set_mode(SCREENSIZE)
  b = BoardGUI(dim, obs_list)
  m = Movement(b, start_position)
  for c in path:
    checkEvents()
    m.move(c)
    pygame.display.update()
  final()
```

6.1 BoardGUI

This class needs to know the number of rows or columns and the list of obstacles to represent the board. Given a fixed size of the window, once the number of rows or columns is known, proceed with the calculation of the actual size of the cell and proceed with loading the images. Everything is done using the following methods:

- ***_to_screen_coordinates*** that takes in input a cell index and return the coordinate in pixel of the cell that corresponds to the point to start drawing
- ***loadImage*** that load the image data into a variable
- ***drawImage*** that draw the image loaded prevously in all the cells present in the list passed in input
- ***drawGrid*** which is the method that calculates the position of the dividing lines between rows and columns and draws them on the screen

6.2 Movement

The Movement class needs an instance of boardGUI from which it takes the information relating to the grid and which it also uses to redraw the grid itself

at each movement of the agent. It must also know the agent's initial position. The methods present in the class are the following:

- ***loadImage*** that load the image data into a variable
- ***move*** takes care of moving the agent by drawing it in different positions translated by a certain step. The difference along the axes between the current position and that of the next cell is calculated and the agent position is cyclically updated and redrawn until the two positions coincide. Each time the method is called, the grid is also updated

6.3 Graphical example

The simulation of the algorithm results is performed in a fixed resolution 600x600 window with the setting shown in Figure 6.2.

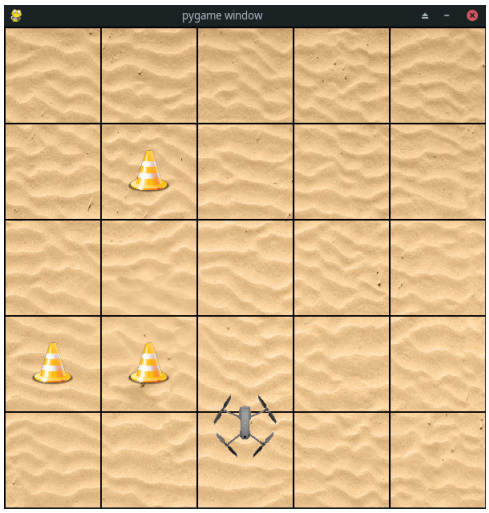


Figure 6.2: GUI of the algorithm

Bibliography

[1] Giovanni Russo, Senior Member, IEEE (2020), *On the Crowdsourcing of Behaviors for Autonomous Agents*, IEEE CONTROL SYSTEMS LETTERS, VOL. 5, NO. 4, OCTOBER 2021

[2] *https://pypi.org/project/pygame/*, pygame is a free and open-source cross-platform library for the development of multimedia applications like video games using Python