Head Star (H*): A Motion Planning Algorithm for Navigation Among Movable Obstacles

Halim Djerroud
ESIEE Paris, Gustave Eiffel University, France

Keywords: Motion Planning, Navigation Among Movable Obstacles, Bug Algorithms.

Abstract: The objective of Navigation Among Movable Obstacles (NAMO) is to optimise the behaviour of robots by giving them the ability to manipulate obstacles. Current NAMO methods use two planners, one for moving through open spaces and a second for handling obstacles. However, these methods focus on providing a solution for obstacles handling and neglect movement in free spaces. These methods usually assume using classical obstacle avoidance algorithms for moving in free spaces. However, they are not suitable for the NAMO. This paper proposes a new path planning algorithm Head Star (H*) adapted for NAMO in free spaces. It is inspired by Bug's algorithms by adding a graphical representation and heuristics on the distances allowing it to bring the robot as close as possible to its goal while keeping in memory the areas already visited.

1 INTRODUCTION

Designing a robot that can navigate in a congested environment and move obstacles in its path is a field that roboticis have been interested in for several decades, known as Navigation Among Movable Obstacles (NAMO) (Stilman and Kuffner, 2005). NAMO is an important area of research in motion planning in congested environments, as it gives mobile robots a better ability to reason about the environment and the possibilities of choosing which obstacles to manipulate, in order to make their way through (Charalampous et al., 2017). NAMO thus makes it possible to solve problems that are difficult or even impossible to solve with a conventional obstacle avoidance methods.

Current work in the NAMO field can be divided into two broad categories: *off-line planning* and *online planning* (Moghaddam and Masehian, 2016; Renault et al., 2019). Offline planning assumes that all the information of the space in which the robot is moving is known in advance, while in contrast to online planning, the robot has only partial knowledge of its environment, and it can modify its initial plan according to new information acquired during its movement. In most previous work that has focused on offline NAMO, the results show that this approach is not effective. The current trend is towards online NAMO, which seems to be more promising and better suited.

There is not much work on online NAMO, but the article (Renault et al., 2019) gives an almost exhaustive list of algorithms with a classification according to the type of NAMO used. The approaches *online* are those whose mobility is not given in the Table 1 in (Renault et al., 2019)

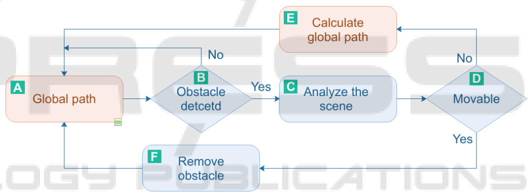

Figure 1: Overview of a method for solving NAMO: (A) From a map of the environment the algorithm first calculates a collision-free path to its goal. (B) During the execution of this path, the robot continuously searches for obstacles not recorded in the map that may block its path. (C) If such an obstacle is detected, the robot initiates a scene analysis to determine the mobility of the obstacles. (D) In case the obstacle is not removable, then (E) the robot re-plans its path. Otherwise (F) the robot removes the obstacle.

In most of the approaches the authors use two planners, one for the movement in free spaces to calculate an optimal path and a second one for the handling of obstacles. According to our knowledge the set of proposed approaches focuses on the operation of the planner that handles obstacles. For what concerns the global planner that allows to compute a path in free spaces the authors only assume that those planners uses a motion planning classical algorithm. This

Djerroud, H.
Head Star (H*): A Motion Planning Algorithm for Navigation Among Movable Obstacles.
DOI: 10.5220/0011637200003393
In *Proceedings of the 15th International Conference on Agents and Artificial Intelligence (ICAART 2023) - Volume 3*, pages 211-219
ISBN: 978-989-758-623-1; ISSN: 2184-433X

article seeks to resolve this lacking by proposing an alternative path planning algorithm for NAMO.

In order to illustrate how the two planner methods work, we propose the diagram in Figure 1 which proposes a method for solving NAMO whose objective is to lead the robot to a determined position by manipulating the obstacles that hinder its passage. In this method two planners are assumed, the first one (red rectangles) allows movement in free spaces. The second one (blue rectangles) allows the management of obstacles. In this proposal, the method consists of bringing the robot as close as possible to its objective. If a non-removable obstacle is in the proposed path then a new path is calculated.

In order to achieve an efficient global planner we believe it is important to use a path calculation algorithm adapted to NAMO, because classical planning algorithms will try to find a path assuming that all obstacles are impassable. In the case of NAMO with a global planner, to calculate an optimal path it is necessary to try to bring the robot as close as possible to its objective, even if it means finding itself in a blocking position, as long as the local planner can unblock the situation by moving the obstacles.

In this paper we propose a motion planning algorithm adapted for NAMO. The objective of this algorithm is to bring the robot as close as possible to its goal to allow an obstacle manager to remove obstacles that hinder the passage. The algorithm also allows a new path to be calculated quickly and taking into account the areas already explored in the event that the local planner fails to find a solution in the event of a blockage.

In the following we will describe the existing motion planning algorithms and show the limitations of these algorithms for NAMO. Afterwards we will propose a motion planning algorithm for NAMO under the name of HeadStar (H*). Then, we will show the experiments carried out prior to concluding.

2 OVERVIEW OF MOTION PLANNING METHODES

Motion planning is an important step to address NAMO problem. It allows the robot to find a sequence of valid configurations which allow it to move from a start position A to reach a goal B even if this means finding itself in a blocked position.

In conventional navigation, all obstacles are assumed to be impassable. This problem is known in robotics under the term of *motion planning*. Historically, it was described in (Eiben and Kanj, 2017) also called *piano movers problem*. It stipulates that if a robot in an initial position and orientation wants to move towards a final position and orientation, there must be at least one valid path between these two positions. A movement is said to be valid if it is carried out completely without collision. Motion planning, therefore, consists in finding a valid trajectory between two positions. A motion planning method must: either generate a movement such that the robot can reach the final position without colliding with obstacles, or conclude that such a movement is impossible.

Motion planning algorithms, described in (LaValle, 2006) are divided into two main classes: (1) deterministic methods, also called exact methods, which make it possible to find the same path at each execution for a given configuration of the environment. (2) Probabilistic methods, also called sampling methods, on the other hand, can find different paths for the same initial conditions, but they guarantee finding a solution if it exists, or to determine that a solution is definitely nonexistent.

The solutions provided by the first class of algorithms are based on the topology of the free spaces called also configuration spaces, in order to build a graph. So they consist in reducing the problem to a simple path finding in a graph. The technique takes place in two steps: The first consists of constructing a graph conforming to the configuration space. The second step uses algorithms for finding paths in a graph, for example: $Dijkstra$, $A*$, $Bellman$, $D*Lite$, etc. As for the step of construction of the graphs, it is based on a cells division method.

Among these techniques of cell division, one of them consists of dividing the configuration space into a set of square cells. In this technique it is considered that each cell is linked to the eight neighbouring cells. All the cells can be represented in the form of a graph. The cells represent the nodes, and the passages from one cell to another are represented by weighted edges as follows: The passages from one cell to another vertically or horizontally is weighted at 1, diagonal passages are weighted at $\sqrt{2}$. If the cell represents an obstacle, then the edges towards this cell are weighted to infinity.

The methods presented here (the list is not exhaustive, we have only cited the most widespread methods involved in this article) are so-called exact methods; they have the advantage of working well for a robot in the configuration space $C = R^2$. They are generally based on graphs, so it is easy to calculate the properties (example: the shortest path, the total cost, etc.) They also offer certain theoretical guarantees (completeness, limits on the execution time, etc.) But these methods are only used when the environment and the obstacles are known (or partially known on condition that certain algorithms such as $D*Lite$ (Koenig and Likhachev, 2002) are used for example) in advance and can be very slow if the number of dimensions is greater than two.

Another category of algorithms, called *probabilistic algorithms*, is based on the use of randomness to construct a connected graph in a configuration space, thus allowing it to be freed from an exact representation of the environment. These methods perform a random search in free space until the desired final configuration is reached. Among these methods we find the Probabilistic Roadmap Approach (PRM) (Chen et al., 2021). As previously, the technique consists of two phases. The first phase in the construction of a graph (this phase is called *learning phase*) and the second, (called *research phase*), consisting of the search for a path in this graph. The graph represents the free space. To ensure a total exploration of the free space, each node of the graph is drawn randomly according to a uniform distribution. Each time a node is randomly pulled it is verified as to whether or not there is a straight line without collision with the nodes closest to the graph, which respects the constraints of the mobile. If this condition is verified, then we add this new node to the graph and connect it to the closest nodes. There are several variants of this algorithm. Among them, Rapidly exploring Roadmap Random Trees (RRT) (LaValle et al., 1998; Kalpitha et al., 2020), which is based on the same technique but allows convergence more quickly by choosing only the nodes which are relevant to the solution of the problem, resulting in the reduction of computation time. From the initial position of the mobile, a tree is successively constructed by successive integration of the nodes. To create a *new* node, we generate a random sample of the configuration space that we will call *rand* using a random distribution to maximise exploration. The *near* node closest to *rand* is determined, and a new candidate configuration *new* then produces a segment joining *near* to *rand*, at a distance prefixed δ from *near*. Finally, we check that the segment from *near* to *new* is collision-free. If this condition is true, we add *new* to the graph and connect it with a segment *near* to *new*.

Another class of algorithms, which allows robots to blindly navigate in mapless environments, is commonly known as *Bug Algorithms* (BA). As the name suggests, they have a biological origin. They are based upon techniques inspired by the movement of insects. A description and comparison of a wide variety of these algorithms can be found in the following articles : (McGuire et al., 2019) (Sivaranjani et al., 2021). The principle of these algorithms is that they do not know the position of the obstacles in their environment, nor the relative position of the goal to be reached. They will react locally only to contact with obstacles and walls which constitute their immediate environment so as to allow the robot to progress towards its objective by following the limits of the obstacles and walls. The nature of these algorithms is ideal for indoor navigation, where the map of the environment is not known in advance, and/or the structure of the environment is constantly changing. The Bug Algorithms are based on the following criteria: (1) Unlike many planning algorithms, which assume a global knowledge of the environment, these algorithms assume only a local knowledge of the environment and a global objective. (2) Their behaviour is straightforward: (a) follow the contours of obstacles (b) move in a straight line towards the objective. (3) The range of the sensors is limited and admits a certain range of uncertainty of the final position. The typical behaviour of Bug Algorithms is to move in a direct line whenever possible, until reaching the goal. In case the mobile meets an obstacle it follows the contour of the obstacle until it is possible to go in a straight line towards the goal.

Among the simplest algorithms, we find the **Com** (Lumelsky and Stepanov, 1986) algorithms they have been qualified by the authors of (McGuire et al., 2019) as common sense algorithms and they have abbreviated them under the acronym **Com**. The idea is to move in a direct line whenever possible, until waiting for the goal. In case the mobile meets an obstacle then it follows the contour of the obstacle until it is possible to go in a straight line towards the goal. The first point of contact with the obstacle is called **hit-point**, and the point where the mobile leaves the outline of the obstacle to go in a straight line towards the goal is called **leave-point**. This algorithm can solve several situations, but the authors show some problematic cases shown in Figure 2 (a). To solve the problem posed by **Com** the authors suggest the **Bug1** algorithm which proposes to explore the totality of the obstacle while keeping the last point **leave-point** in its memory as shown in Figure 2 (b). However **Bug1** generally proposes a path for this, the authors propose an optimized version of this algorithm, named **Bug2** whose idea consists in drawing an imaginary line **M-line** between the starting position and the goal. **Bug1** explores the obstacle but this time, until we encounter the **M-line**, this situation is illustrated in Figure 2 (c).

In the same article, (Sankaranarayanan and Vidyasagar, 1990) the authors, evince that there are still cases where **Bug2** proposes unoptimised and very long paths. They consider that the main cause comes from the fact that the algorithm does not store the

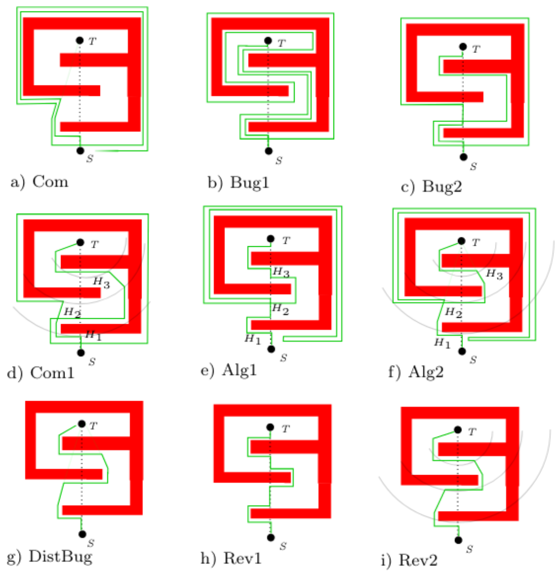

Figure 2: Behaviour of different Bug Algorithms. Here, the environment configuration is chosen to show typical cases, in some cases algorithms may be favoured over others or vice versa. (This Figure is taken from the article (McGuire et al., 2019)).

points visited previously during the contour of the obstacles. For this, the authors propose an algorithm based on **Bug2** under the name of **Alg1**. Its principle is to change the direction of the contour following, if the mobile encounters a **hit-point** already visited (see 2 (e)). The authors also developed another version of this algorithm under the name of **Alg2**. The idea consists in adding an **M-line**, which allows in certain cases to optimize the trajectories (see Figure 2 (f)). Also, in the same article the author proposes a new version of the **Com** algorithm named **Com1**, the idea consists in memorizing the distance of the last reference point **leave-point** compared to the goal (see Figure 2 (d)).

DistBug is an algorithm described (Kamon and Rivlin, 1997). It is similar to **Alg2** except that it only keeps in memory the last position of **hit-points**, which makes it more efficient at the memory level. Another aspect that characterizes **DistBug** is that it does not impose a direction of bypassing obstacles, it is always done in the direction of the **M-line**, as illustrated in Figure 2 (g). In some configurations this strategy fails completely.

In the (Horiuchi and Noborio, 2001), authors propose two algorithms **Rev1** and **Rev2** (see Figure 2 (h) and (i)). Here, the strategy consists in alternating the direction at each **leave-point** to follow a trajectory that looks like a slalom. The idea behind this strategy is to change direction if the mobile passes the same point, which increases the chances of finding a path.

More recently, (Lentzas and Vrakas, 2020) described the LadyBug algorithm. The algorithm uses a Received Signal Strength Indication (RSSI) of an electromagnetic signal to detect its position in regard to signal source. The algorithm is able to accurately calculate the position of the beacon emitting said signal. The algorithm is based on two functions: Localisation and Navigation. The algorithm is able to locate the source of an electromagnetic signal and navigate to it.

We have described how a few insect-inspired algorithms work, but there are many more. The Bug Algorithms shown here are generally suitable for environments such as labyrinths, they are poorly suited for mobile robots able to see far ahead (without hitting obstacles). Because in an environment where it is possible to see far ahead, this information can be used to improve its trajectory. Moreover, these algorithms are not capable of accurately remembering the places already visited.

3 THE LIMITS OF THESE ALGORITHMS FOR NAMO

The first algorithms described here are based on total or partial knowledge of the environment to calculate paths. It is obvious that these techniques cannot be used if one considers that the environment changing after moving obstacles as in the NAMO environment. In addition, these algorithms do not work immediately, they rely on techniques such as SLAM (Kolhatkar and Wagle, 2021) to get the map of the environment.

SLAM techniques consist in exploring the environment to build a map, then the produced map is used to calculate an optimal path between two positions with using path finding algorithm. In the case of an environment with a lot of obstacles, generate a lot of gray areas[1], it is obvious that most of the mapped regions are not used when calculating an optimal path between two positions. In fact, this technique unnecessarily explores certain regions. Moreover, in the case of a dynamic environment, the constructed map quickly becomes obsolete, if the obstacles change place. In the context of an environment with many obstacles, this technique proves to be unsuitable in the event that certain areas are not accessible.

In the case of Bug Algorithms, it is true that the

[1] Areas where you have to bypass the obstacle to map the environment (behind obstacles, places inaccessible by obstacle avoidance, etc.)

mobiles are immediately operational and do not require a preliminary exploration of the environment. But in certain configurations of the environment (in question of the local minima[2]) these algorithms can lead to these cycles (go through the same path) because they do not memorise the configuration of the environment already explored. In addition, these algorithms do not offer any path improvement in the case where the mobile has to redo the same path already taken before. In addition, these algorithms are not based on metrics or heuristics, which makes it difficult to use path search algorithms such as $D*$ or others. The Bug Algorithms try to find trajectories in a straight line towards the objective, but do not memorise the places already explored.

We believe that it is important to propose a planner for NAMO that benefits from the advantages of both approaches. Namely, to move directly towards the objective and rely on the local planner to solve the obstacles that lie ahead. Otherwise, quickly recalculate a path (as straight as possible towards the goal) without returning to the areas already explored.

In the next section, we propose an algorithm that merges the two approaches. It allows the robot to go straight, in case an obstacle blocks its path, then it calculates a new path (based on a heuristic) that allows it to move away from the obstacle while approaching its goal. It also keeps track of the areas visited later so that it does not have to re-explore in the event that a path is not found in the direction taken.

4 HEAD-STAR ALGORITHM (H^*)

The **HeadStar** algorithm noted (H^*), is halfway between bug algorithms and heuristic path search algorithms. It uses only a single heuristic[3] $f(x) = h$. Like bug algorithms, H^* does not previously have a map of the environment. To explore its environment, it must move around. The mobile has a sensor with a certain range. Like the classic path finding algorithms, H^* uses heuristics to optimise navigation paths and avoid unnecessary explorations. The H^* algorithm uses cell division to represent its environment. The sensor range is shown with white cells; grey cells show unexplored areas (see Figure 4), obstacles are shown with hatched cells. The heuristic is calculated with respect to the goal cell.

To prevent the algorithm from spreading in width and to force it to go forward, or to follow the contours, we have introduced the notion of *inconsistency*. A cell marked inconsistent will not be explored as long as there are cells in the *open list*. The choice of inconsistent cells is made when moving from one cell to another. Horizontally and vertically, the mobile marks the cells adjacent to the starting square. In diagonal displacement, we mark the cells adjacent to the transition. The different cases are illustrated in Figure 3.

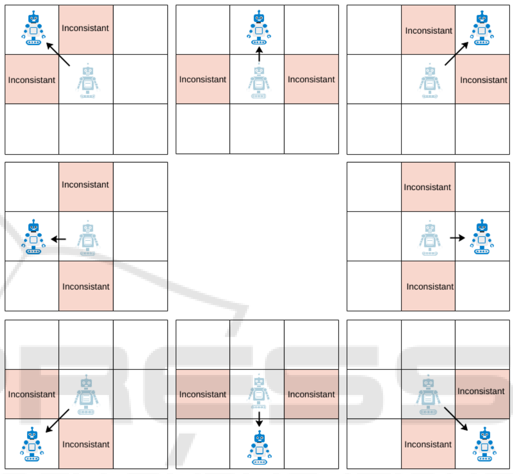

Figure 3: How to determine inconsistent cells.

H^* uses priority queues to store nodes (cells) to be visited, and a stack to store nodes already visited. The priority queues are ordered according to $f(x) = h$; if two elements have the same value then we use g as discriminant $f(x) = h + g$. h is the estimate of the distance between this position and the *goal*, g is the distance between the mobile and this position. The different lists used by the algorithm are as follows:

- **Open list**: priority queue (ordered by h, $g+h$) stores newly discovered vertices.
- **List inconsistent**: priority queue (ordered according to h, $g+h$) stores inconsistent vertices.
- **Closed list**: a stack which stores the vertices already visited.

From the starting position, H^* inserts the starting cell in the *open list* to initiate the process. As long as the *open list* and *inconsistent list* are not empty, we repeat the following process: We calculate the heuristics of the adjacent cells, and we insert them into the priority list, *open list* (if they are not already in one of the two lists). We remove the first element of the

[2]The problem of the local minima occurs when a mobile navigating without a priori knowledge of the environment is trapped in a loop. This occurs especially if the environment consists of concave obstacles, labyrinths, etc. To get out of the loop, the robot must understand its repeated crossing through the same environment, which involves memorising the environment already explored.

[3]Here the heuristic is calculated using Euclidean distance, other calculation methods are possible such as Manhattan distance, Chebyshev distance, etc.

open list. If this list is empty, then we remove the first position in the *list inconsistent*, and we move to this position if the displacement is direct (only one cell displacement). If not, we recall the algorithm recursively (the lists are defined in the stack; during the recursive call, the lists are emptied and find their content when unstacking). During the displacement (in the case where the displacement is of only one cell), we insert the visited position in the *list closed* and we calculate the inconsistent positions and insert them in the *list inconsistent*, each time taking care to remove the positions in their initial lists. The algorithm ends if it reaches the goal or after both *list open* and *list inconsistent* are empty. This algorithm is described in the Algorithm 1.

The function `GetNeighbors(s)` returns a list of adjacent cells that are not obstacles. A cell is said to be adjacent s' if and only if the distance g (distance between the cells s and s') is of a single cell: $g(s,s') \leq \sqrt{2}$ and $h(s') \neq \infty$.

The function `MinCell(`$OPEN$`,` $INCONS$`,` S_{goal}`)` returns the smallest element in the *open list*. If the list is empty then it looks for the smallest elements in *inconsistency list*.

4.1 Illustration of H^*

Figure 4 shows how H^* works. We start by adding the starting position (D2) in the *open list*, we calculate the Euclidean distance to the objective on the adjacent cells, we assign an infinite value to the impassable cells and we add them to the *open list* (only cells with $h \neq \infty$). We move in the cell with a minimum h (E2) and note the cells (D1 and D3) as inconsistent (add these two cells in *list inconsistent* and remove them from the *open list*). When we move into a cell, we remove it from the two lists (*open list* and *inconsistent list*) and put it in *closed list*. The next cell with h minimum in the *open list* is (E3), so we move to this cell directly (in this case the algorithm makes a recursive call and starts again as the starting position (E3) and arrival position C3). As long as the *open list* is not empty, we continue the same process. If the *open list* is empty then we look to see if there are any elements left in *inconsistent list*.

5 EXPERIMENTS AND RESULTS

Here, we will compare the H^* algorithm to the D^*Lite and A^* algorithms. Results are obtained statistically using 3000×26 tests. The number of obstacles starts at 0 and goes up to 300 obstacles on a surface of 20×20. The number of obstacles is gradually increased in

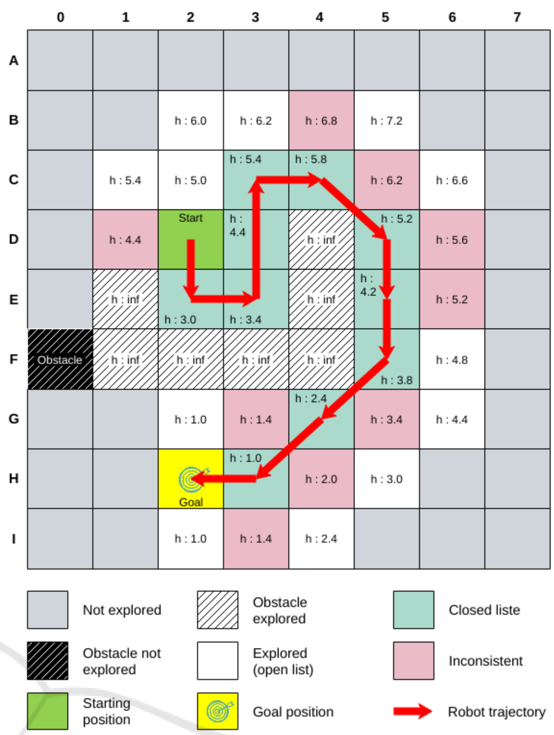

Figure 4: Sample case.

increments of 10 every 3000 attempts. Beyond 300 obstacles on a surface of 20×20, or a density of 0.75, it becomes difficult to find a path between the starting position and the finishing position.

We wish to emphasize that H^* must move to explore its environment, unlike the algorithms with which we compared it. To make a fair comparison, we have assigned tests assuming that D^*Lite has no prior knowledge of its environment. Technically, this operation is carried out by generating the obstacles after the `ComputePath()` function. Figure 6 (A) graph shows equivalent performance—in terms of distance traveled— (slightly in favor of H^*). But D^*Lite is not able to work in an environment where the goal is not known in advance or where the goal is moving. This explains why H^* travels a greater distance.

The graph in Figure 6 (B) shows a comparison of the three algorithms in terms of distance, execution time and nodes in *open list*. We notice that D^*Lite is time consuming when the number of obstacles is low. This is due to the fact that many nodes are found in the *open list* (as we can see in 6 (C)), and the key calculation is based on two criteria (in the implementation used here, the *open list* is implemented in a dictionary, the key being a pair of integers indicating the abscissa and the ordinate of the cell. The value is composed of a pair of doubles $<min(g(s),rhs(s)) + h(s_{start},s), min(g(s),rhs(s))>$. The process of get-

Algorithm 1: Algorithme HeadStar.

```
function COMPUTEPATH(S_start, S_goal, Path)
    OPEN ← ∅
    INCONS ← ∅
    CLOSE ← ∅
    s ← S_start
    OPEN.insert(s)
    while OPEN.size() ∉ ∅ or INCONS.size() ∉ ∅ do
        ADDTOCLOSELIST(s, OPEN, INCONS, CLOSE)
        Path.insert(s)
        if s = S_goal then
            return True                                        ▷ Path found
        end if
        for all s' ∈ GETNEIGHBORS(s) do
            ADDTOOPENLIST(s, OPEN, INCONS, CLOSE)
        end for
        next_pos ← MINCELL(OPEN, INCONS, S_goal)
        if next_pos = ∅ then
            return False                                       ▷ No path available
        end if
        if |next_pos − s| ≤ √2 then
            inc_list ← GETINCONSISTANT(s, next_pos)
            for each s' ∈ inc_list do
                ADDTOINCONSISTANTLIST(s, OPEN, INCONS, CLOSE)
            end for
            s ← next_pos
        else
            if COMPUTEPATH(s, next_pos, Path) then
                s ← next_pos
            end if
        end if
    end while
    return
end function
```

ting the minimum cell in the *open list* is implemented in the `TopKey(open_list)` function, using the sort function of the **STL** (see Listing 1)). The two remaining algorithms, occupy linear time although H^* is a little more efficient than A^*. Again, this is due to the number of nodes found in *open list*; it is larger in A^*.

```
bool
compare(pair<pair<int,int>,
        pair<double,double>> x,
        pair<pair<int,int>,
        pair<double,double>> y){
  return (
    x.second.first <= x.second.first &&
    y.second.second <= y.second.second
   );
}

pair<double, double>
D_Light_Star::TopKey(op_lst &ls_open){
    ls_open.sort(compare);
    return ls_open.front().second;
}
```

Listing 1: Cell with the lowest key.

Image 5 shows the use of H^* with the robot *TurtleBot3* in the *Gazebo* simulator and *RViz*. Image (A) shows the environment of the robot in the Gazebo simulator. The robot's start position and the goal position are shown with red circles. The robot maps its environment using a LIDAR. The figures from (1) to (8) are generated by *RViz*; they show the different stages through which the robot goes to reach its objective. The areas in dark grey show unrecognised areas and, in light grey the free spaces. Obstacles are represented by black areas. The robot's position is always indicated with the red circle. Figure (B) shows the Robot's trajectory from its starting position to its goal

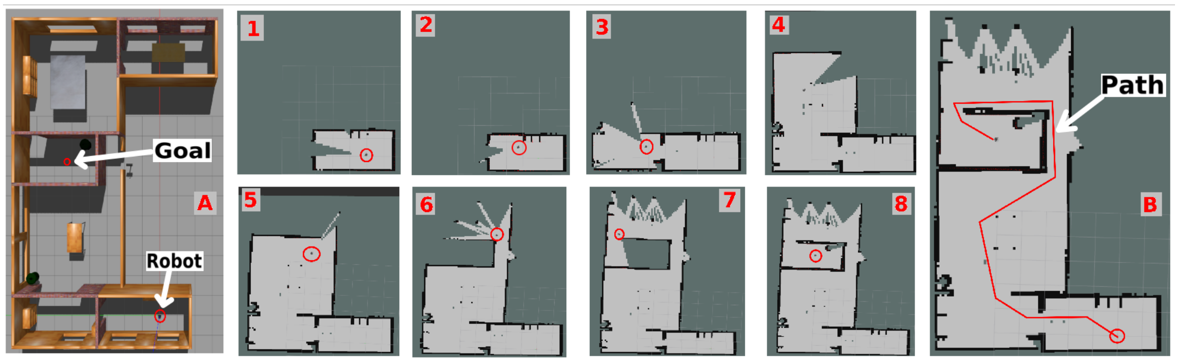

Figure 5: Simulation of H^* with the *TurtleBot 3* robot in *Gazebo*.

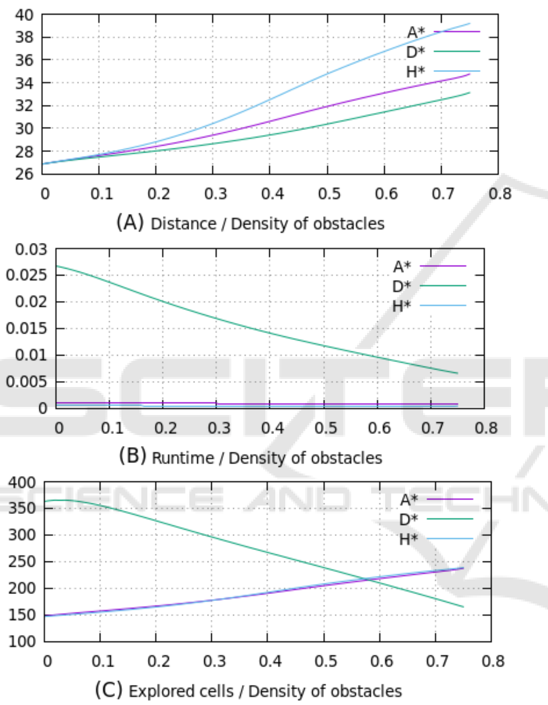

Figure 6: Comparison of H^* with A^* and D^*Lite algorithms.

position.

6 CONCLUSIONS

In this article, we have presented H^*, a motion planning algorithm for NAMO. The principle is to combine the techniques of bug algorithms and path search algorithms which use heuristics. The first results obtained in simulation are very interesting, they show that H^* is able to find a path without providing it with a map beforehand. In our experiments, we noticed that H^* provides a path equivalent to D^*Lite in most typical environment configurations. Among the possible improvements that we are experimenting with is that of finding the best way to determine inconsistent cells in order to improve the performance of H^* in terms of distance traveled. We have also started to experiment with our algorithm on a mobile robot in a real environment.

REFERENCES

Charalampous, K., Kostavelis, I., and Gasteratos, A. (2017). Recent trends in social aware robot navigation: A survey. *Robotics and Autonomous Systems*, 93:85–104.

Chen, G., Luo, N., Liu, D., Zhao, Z., and Liang, C. (2021). Path planning for manipulators based on an improved probabilistic roadmap method. *Robotics and Computer-Integrated Manufacturing*, 72:102196.

Eiben, E. and Kanj, I. (2017). How to navigate through obstacles? *arXiv preprint arXiv:1712.04043*.

Horiuchi, Y. and Noborio, H. (2001). Evaluation of path length made in sensor-based path-planning with the alternative following. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164)*, volume 2, pages 1728–1735. IEEE.

Kalpitha, N., Murali, S., et al. (2020). Optimal path planning using rrt for dynamic obstacles. *Journal of Scientific and Industrial Research (JSIR)*, 79(06):513–516.

Kamon, I. and Rivlin, E. (1997). Sensory-based motion planning with global proofs. *IEEE transactions on Robotics and Automation*, 13(6):814–822.

Koenig, S. and Likhachev, M. (2002). D^* lite. *Aaai/iaai*, 15.

Kolhatkar, C. and Wagle, K. (2021). Review of slam algorithms for indoor mobile robot with lidar and rgb-d camera technology. *Innovations in Electrical and Electronic Engineering*, pages 397–409.

LaValle, S. M. (2006). *Planning algorithms*. Cambridge university press.

LaValle, S. M. et al. (1998). Rapidly-exploring random trees: A new tool for path planning.

Lentzas, A. and Vrakas, D. (2020). Ladybug. an intensity based localization bug algorithm. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 682–689. IEEE.

Lumelsky, V. and Stepanov, A. (1986). Dynamic path planning for a mobile automaton with limited information on the environment. *IEEE transactions on Automatic control*, 31(11):1058–1063.

McGuire, K. N., de Croon, G. C., and Tuyls, K. (2019). A comparative study of bug algorithms for robot navigation. *Robotics and Autonomous Systems*, 121:103261.

Moghaddam, S. K. and Masehian, E. (2016). Planning robot navigation among movable obstacles (namo) through a recursive approach. *Journal of Intelligent & Robotic Systems*, 83(3):603–634.

Renault, B., Saraydaryan, J., and Simonin, O. (2019). Towards s-namo: socially-aware navigation among movable obstacles. In *Robot World Cup*, pages 241–254. Springer.

Sankaranarayanan, A. and Vidyasagar, M. (1990). A new path planning algorithm for moving a point object amidst unknown obstacles in a plane. In *Proceedings., IEEE International Conference on Robotics and Automation*, pages 1930–1936. IEEE.

Sivaranjani, S., Nandesh, D. A., Gayathri, K., Ramanathan, R., et al. (2021). An investigation of bug algorithms for mobile robot navigation and obstacle avoidance in two-dimensional unknown static environments. In *2021 International Conference on Communication information and Computing Technology (ICCICT)*, pages 1–6. IEEE.

Stilman, M. and Kuffner, J. J. (2005). Navigation among movable obstacles: Real-time reasoning in complex environments. *International Journal of Humanoid Robotics*, 2(04):479–503.