POLITECNICO DI TORINO

Collegio di Ingegneria Informatica,
del Cinema e Meccatronica
Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Fast and continuous path planning of ground robots for aerospace exploration

Supervisors
Prof. Alessandro Rizzo
Prof. Marina Indri

Candidate
Gabriella Pace

October 2022

Abstract

The path planning problem is a computational problem which is directed to search for sequences of positions (or, equivalently, movements) that can take a vehicle from a starting point to an end point, or even to allow the vehicle to pass through a predefined sequence of points. The simplest versions have the sole objective of avoiding areas where the vehicle could not physically travel or where it would sustain damages, but this almost always generates more than one valid solution. For this reason, many algorithms also deal at the very least with choosing the shortest, fastest or safest of these alternatives to adopt. In addition, a more extensive combination of factors might be wanted to be taken into account, thus entering into multi-objective path planning.

This is precisely the cardinal topic of this thesis, in particular it is designed to be used by rovers for Martian exploration. In fact, the planner is designed to be integrated into the system created by TAS-I to be deployed on the Adept Mobile Robots Seekur Jr on their property.

The objectives to be optimized considered here aim to achieve the shortest and safest path, taking into account the necessary energy consumption or solar exposure during the journey (since recharging is usually done through solar panels), and this is achieved through the MOD* Lite incremental search algorithm (an enhancement of D* Lite).

Contents

Acronyms

Dynamic SWSF-FP	Dynamic Strict Weak Superior Function - Fixed-Point
LPA*	Lifelong Planning A*
D*	Dynamic A*
MPOs (MOOP)	Multi-Objective Optimization Problems
NP-hard	Nondetermistic Polynomial-time hard
MOPP	Multi-Objective Path Planning algorithm
MOGPP	Multiobjective Genetic Path Planning
MOEA	Multi-Objective Evolutionary Algorithm
SPEA2	Strength Pareto Evolutionaty Algorithm - 2
MOA*	Multi-Objective A*
MOD* Lite	Multi-Objective D* Lite
IMU	Inertial Measurement Unit
ToF	Time of Flight
SINAV	Soluzioni Innovative per la Navigazione Autonoma Veloce
TAS-I	Thales Alenia Space Italia
RoXY	Rover eXploration facilitY
AutoNav	Autonomous Navigation
ROS2	Robot Operating System - v2
Nav2	Navigation 2
HAL	Hardware Abstraction Layer
BT	Behavior Tree

1. Introduction

In the sphere of space exploration, there is a pressing need to develop increasingly precise and lightweight methods of making rovers autonomous in order to minimize the damages reported and optimize the time and resources available. The most common situation is to have a terrestrial rover (at most aided by the eyes of a drone, but anyway not in real time) having to locate itself and then navigate the rough terrain such as the one found on Mars, with almost no prior knowledge of the surrounding environment, which moreover is mutable over time.

The longest drive in one sol has been 100.3 m, registered by Curiosity on July 21 (or sol 340) 2013. (Woods, et al., 2014)

In fact, due to the poor lighting conditions and the low computational power available on-board, the rover needs to stop frequently to recalculate the best route to take, and this takes longer than the time it actually needs to travel it.

On the contrary, the need for remote human intervention, or even just to have data processed by more powerful computers located on Earth, makes making decisions and carrying out research successfully increases the time required for each action considerably. A Mars mission is typically forced to limit itself to one downlink and one uplink in an entire sol (Washington, Golden, Bresina, Smith, & Anderson, 1999). There is therefore a compelling need to make the rovers increasingly autonomous in their decision-making and for longer periods. At the moment, Autonomous Navigation drives are at least three times slower than drives with directed commands. (Woods, et al., 2014) This requires improving both their perception of the surrounding environment and their ability to explore it safely.

It would also be advisable for the rovers to be able to perform more than one task at the same time, in order to maximize the time available to each mission and the limited communications available. In this case, however, many new difficulties arise, including the need for the rovers themselves to be able to find compromises that can carry out multiple actions without losing or overly penalizing any.

The aim of this thesis is to implement an autonomous navigation algorithm that takes into account multiple objectives simultaneously.

In particular, it is considered a situation in which the rover in issue is asked to autonomously reach a point selected by a human operator. Such task is however done by trying to take the shortest route that will get it to its objective safely, without draining more of its available energy than necessary in an attempt to overcome avoidable obstacles, and at the same time trying to maximize the light absorption of its solar panels, i.e., preferring the most sunny areas along the way. The last objective becomes interesting since many rovers currently in use employ solar power as one of their major sources.

Such an algorithm is, among other things, thought to be embedded in and integrated with an entire autonomous navigation system that would also provide mapping, localization, control and locomotion solutions. This autonomous navigation system is represented by the infrastructure created in the SINAV (Soluzioni Innovative per la Navigazione Autonoma Veloce) project carried out to test new navigation possibilities and solutions for space exploration.

To achieve this goal, the current possibilities and needs of the SINAV project (explained in more detail later) were evaluated, after which several algorithms that could integrate well with it were analysed. Among them, the MOD* Lite algorithm was selected and implemented successfully.

Unfortunately, for various reasons, it was ultimately not possible to attempt such integration, either in the virtual environment or on the real vehicle. They are therefore mentioned as necessary developments for the future.

1.1 Thesis structure

The next chapter introduces the problem of path planning, the difficulties involved, and some approaches for its resolution.

Thereafter, the hardware technologies commonly used to provide rovers with the necessary information about the environment in which they find themselves will be sounded out. The third chapter then analyses the most commonly used methodologies for holding and processing the information thus collected.

In the sixth chapter, a selection of significant algorithms aimed at finding solutions to the path planning problem are reviewed, exploring in depth, for those most significant to this discussion, the general operation, strengths and weaknesses. The next chapter takes a closer look at one of these algorithms, MOD* Lite, which was chosen for the practical implementation.

Afterwards, the different tests carried out to verify the functionalities of this algorithm and optimize its execution are presented.

The last chapter draws conclusions from this work and leaves suggestions for future developments.

2. The path planning problem

The path planning problem is a computational problem which is directed to search for sequences of positions (or, equivalently, movements) that can take a vehicle from a starting point to an end point, or even allow the vehicle to pass through a predefined sequence of points. This is a purely geometric problem, as it does not take into account the flow of time in any way.

In the case of rovers for Martian exploration, path-planning plays a very important role due to the slow rate of communication. The operator has no choice but to indicate a desired area to the rover and let the vehicle decide which path to follow.

The simplest versions have the sole objective of avoiding areas where the object (vehicle) could not physically travel or where it would sustain damages, but this almost always generates more than one valid solution. For this reason, many algorithms also deal at the very least with choosing the shortest, fastest or safest of these alternatives to adopt.

In addition, a more extensive combination of factors might be wanted to be taken into account, thus entering into multi-objective path planning.

Another important characteristic to consider is the environment in which planning is to take place, how much of it is known in advance, how variable it is over time, and the extent of the area itself.

When the size of the area in which the vehicle has to navigate is large but at the same time the accuracy required is high, it is often decided to break the problem down into two phases, a global planning and a local one.

2.1 Global path planning

The *global planners* deal with the identification of an indicative route on a low-resolution map, leading theoretically from the starting position to the ultimate goal of the crossing. This type of planner generally performs very well in known, static environments, but can lead to dangerous mistakes in the case of dynamic circumstances.

These algorithms are in charge of defining paths in a way that reduces the total cost to reach the objective, without ever putting the safety of the equipment at risk. In fact, the ultimate objective is to collect as much data as possible, so compromising some instrument or worse the entire vehicle would call for the entire mission to be aborted.

This type of algorithm, has a very large research history on graph theory and artificial intelligence, going all the way back to the Dijkstra algorithm (Knuth, 1977), which was designed for the offline resolution of the Single Source Shortest Path problem.

2.2 Local path planning

Local planners have instead the task of detailing the route in the immediate surroundings of the vehicle, using a higher resolution map (which may have been captured by the vehicle itself). These planners are therefore much better suited to dynamic settings but are also much slower, leading to them being able to operate only on small portions of the total desired path.

Local planners are generally conceived not so much to establish general crossing strategies, but rather to promote safe and smooth behavior. To this end, they often come to take into account the specific kinematic and dynamic characteristics of the vehicle so as to prevent skidding or other hazards. (Helmick, Angelova, Livianu, & Matthies, 2007) (Jain, et al., 2004)

An important example is the Grid-based Estimation of Surface Traversability Applied to Local Terrain (GESTALT) (Goldberg, Maimone, & Matthies, 2002), developed by The Jet Propulsion Laboratory (JPL) and implemented in Mars exploration rovers. Whenever a new terrain capture is provided, the traversability map is updated. Then GESTALT calculates the cost of each possible local trajectory by integrating the traversability values along that curve, choosing the primitive with the lowest cost.

A complete planner will then be able to integrate these two features into a single coherent output. In this way, the rover will be put in good enough conditions to take fairly long routes, avoiding both large obstacles and small ones. (Bombini, Coati, Medina, Molinari, & Signifredi, 2015)

First the global planner will be called to operate, and its response will be provided to the local planner, which will take action if necessary to refine it. Thereafter, it is most likely that changes to the surrounding environment will be detected only locally, and thus it will still be the local planner who will intervene to divert the path where necessary, avoiding straying too far from the original trajectory.

The basic concept is to start with satellite maps, which provide a very low resolution of potential obstacles, and generate an initial path hypothesis having as the final goal an operator specified location. The rover can then scan its surroundings (local scan) and use this much more detailed information to correct the route immediately ahead, repeating this process periodically.

This thesis will focus on algorithms typically used as global planners, but which are progressively refined enough to sufficiently handle some dynamic changes in the environment.

2.3 Sensors typically employed

In robotics there is a continuous research, development and production of instrumentation that can improve the machines' perception of their surroundings and their position and orientation in space. Although most of these are very versatile, they can be broadly divided according to their use, whether it is to sense the environment or to detect one's location in it.

2.3.1 Mapping

Mapping refers to the phase in which the robot gets to know and construct a map of his surroundings.

Stereo-cameras can provide outputs in the form of color or black and white images, but also as point clouds.

At the moment, this is the best method for a navigation on Earth and is also widely used for space exploration (Guan, Wang, Fang, & Feng, 2014). As a matter of fact, it provides an accurate and complete description of the ground, but on the other hand, it takes a lot of memory and computing power to process such complex images. A possible workaround to the problem is to process them as point clouds, but these still involve a considerable effort.

Another technology is represented by **spectral sensors** obtain information from across the electromagnetic spectrum. They might use infrared, the visible spectrum, the ultraviolet, x-rays or some combination of the above.

Between these, some require a huge amount of memory (like the Hyperspectral datacube) but are mostly used to perform a recognition of objects, feature that anyway does not perfectly fit the purpose of rough terrain recognition. (Nieto, Monteiro, & Viejo, 2010)

Others, such as Near-Infrared (NIR) cameras (about 0.78–3 μm), require less computing power and could provide useful information about the type of ground surface.

Thermal cameras have also been considered since widely used on Earth (Milella, Reina, & Nielsen, 2021), but not knowing the expected performance given the very low Martian temperatures (on average -63 °C), they are not among the favoured methods.

With **laser sensors**, measured data can be represented in a number of ways, from raw data to parameterized models. They are widely used in robotics and all kind of autonomous vehicles, since they can provide three-dimensional measurements in real-time (Geiger, Ziegler, & Stiller, 2011).

With the objective of mapping a desertic terrain, the excellent results in the recognition of articulated objects could be useless, but the use of laser range sensors could prove beneficial instead.

In particular, **Lidar** (Light Detection and Ranging) is currently widely used both on Earth's surface and ocean depths by varying laser wavelengths. It pulses a laser, or laser grid, as the light source for its measurements. With these measurements, it is able to build a point-cloud, which can be left as it is or used itself to build a three-dimensional map or an image. As seen in these situations, it performs very well even in situations with small to no light available. Therefore, it offers very promising outlets for mapping the Martian terrain. (Rekleitis, Bedwani, & Dupuis, 2009)

ToF (Time-of-Flight) sensors are instead based on light detection, typically using a standard RGB camera. This means that, compared to Lidar, it requires much less specialized equipment. From the measurements obtained, it is able to create depth maps. The shortcoming of such sensors is exactly that, namely the fact that depth maps are generally more difficult for computers to process than point-clouds (Xiao, Liu, & Zhu, 2021).

2.3.2 Localization

The localization phase, alongside the mapping phase, allows the robot to make an estimate of its position and orientation in space at any given time.

IMUs and **odometers** are commonly deployed for various purposes among which is the localization of the vehicle (e.g. the IMU is typically used for the anti-tilt system) and can be useful also in adding information to the mapping of the environment. (Guan, Wang, Fang, & Feng, 2014)

On their own they are currently not considered sufficient to this task, due to the fact that they are subject to several sources of error and that such errors in navigation accumulate over time. For short distances, however, they demonstrate decent accuracy and can therefore lead to real-time autonomy.

The **Pamcam**, used by Nasa (Mars Exploration Rovers, s.d.), is able to determine the position and the direction in which the rover is facing using the Sun's movement. Having to wait for the Sun to make an arc in the sky, it needs to collect data for a long time (about 10 minutes) before providing an estimation. Additionally, it relies on prior knowledge of the terrain from **satellite images** to position itself in space.

2.4 Data structures for terrain representation

One of the biggest bottlenecks in streamlining location processes is the representation of data on maps from the various sensors.

Terrain scans are carried out with three main goals in mind. The first is obviously the planning of a safe path. The second, is the refinement of the location estimate, and thus of the trajectory executed, by observing the terrain freshly traversed. The third, closely related to the second, is the creation of a more complete map useful for later activities (Rekleitis, Gemme, Lamarche, & Dupuis, 2007).

For example, laser sensors are able to directly provide a three-dimensional **point-cloud** without the need for further processing, so the shortest option would be to use it as it is provided (Geiger, Ziegler, & Stiller, 2011). There have been many studies aimed at solving precisely this problem, managing the high volume of data and the non-uniform density of the scans, and some of the available options are now analysed.

One of the possibilities are the tree-based data structures, among which we can mention quad and oct-trees.

Quad-trees lead to ease of data processing, however they are mostly suitable for representing two-dimensional spaces, thus not sufficient for our purposes.

Oct-trees provide a three-dimensional representation but with a finite resolution (even the maximum may not be sufficient in some cases). Moreover, it involves a very high computational load for each update.

The DEM (Digital Elevation Model) is a simple regularly spaced grid of elevation points that represent the continuous variation of relief over space.

It is, as quad-trees, a 2.5D representation, so they are not able to correctly represent concave geological structures such as overhangs and caves, which are by far not negligible in our applications.

The Grounded Heightmap Tree is data structure for terrain representation built as a generalization of the DEM.

2.4.1 Grid construction

Starting from the solutions already implemented and rooted in the SINAV project, which are detailed in the section 5. , the way in which to capture the terrain features was chosen.

Through the stereo-cameras and sensors, a 3-D map of the terrain, in the form of a point-cloud, surrounding the rover is generated every few seconds. From this, the necessary data are then extrapolated.

Solutions commonly applied to provide robots with maps are occupancy grid maps, semantic ones, and topological ones (Jinming, Xun, Lianrui, & Xin, 2022). Of these, the one that offers the most advantages is the grid map, which is essentially represented as a quadrilateral grid, and it is the one that will be considered here. “The occupancy grid map discretizes the spatial environment perceived by robots into equally sized grids and then applies a specific probability of occupation to assign the attributes of the grid.” from (Elfes, 1989)

The entire terrain is subdivided through a tuneable porosity grid, to get an idea of orders of magnitude, in global planning cells of $3m \times 3m$ are usually chosen, while in local planning smaller cells of about $1m \times 1m$ are chosen.

Scores are then assigned to each cell, with the purpose of maintaining the initial map information. To give practical examples, these scores usually refer to the average altitude of the terrain in that cell, or its local slope. Or even, as in our case, the solar exposure to which it would be subjected at the time of the analysis.

Another important feature of such grids to be established is the specific point in each square where the rover is considered to be when it is claimed to be in a given cell, that is, whether in its center, on one of its sides or on one of its edges. The most common choice, also used here, is to place it in the center.

In addition, it is essential to determine how the rover will be able to move from one cell to another, whether it is only between adjacent cells or also diagonally (as depicted in Figure 2.1.). As will be explored later, this will give rise to 4-edge-connected graphs in the first case and 8-edge-connected-graphs in the second. The second case obviously offers greater mobility and leads to far more efficient paths, although it involves slightly higher computational loads by having to explore more possibilities (Wang, et al., 2022). In this discussion we will focus on movements of the second type.

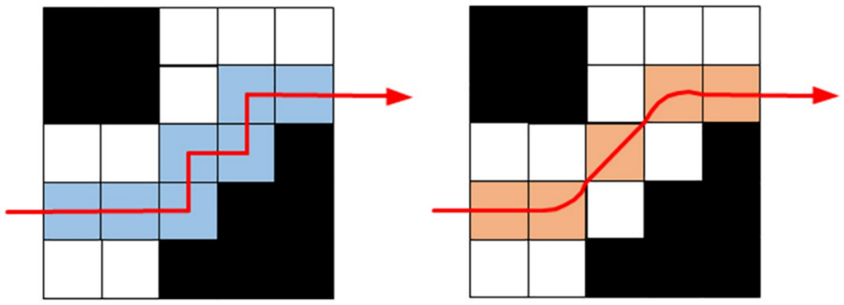


Figure 2.1. Cell traversability. On the left, the vehicle can only move between adjacent cells (4-connected). On the right, diagonal movements are allowed (8-connected)

2.4.2 Scoring strategies

Once the grid is obtained, a score must be assigned to each cell based on the average characteristics of the terrain it represents.

If, for example, we consider path safety as an objective to be optimized, it can be associated with different terrain characteristics, such as slope, type of terrain (sand or rock), or proximity to obstacles that cannot be traversed. Once the parameters contributing to this assessment have been established, thresholds are set to differentiate the various possibilities. Within each threshold, scores, often percentages, are assigned, ranging from the optimal to the worst situation.

For features such as the elevation of a zone, the scoring can trivially be a measurement of it.

An example can be seen in Figure 2.2. where the cells marked by the white Xs are non-traversable obstacles. The other black cells surrounding them are a safety measure to keep in consideration the size of the vehicle (schematized in green) to avoid even partial collision with such obstacles. The remaining cells have a gradation of grey that visually represents their score, given solely on the basis of their distance from such off-limits areas.

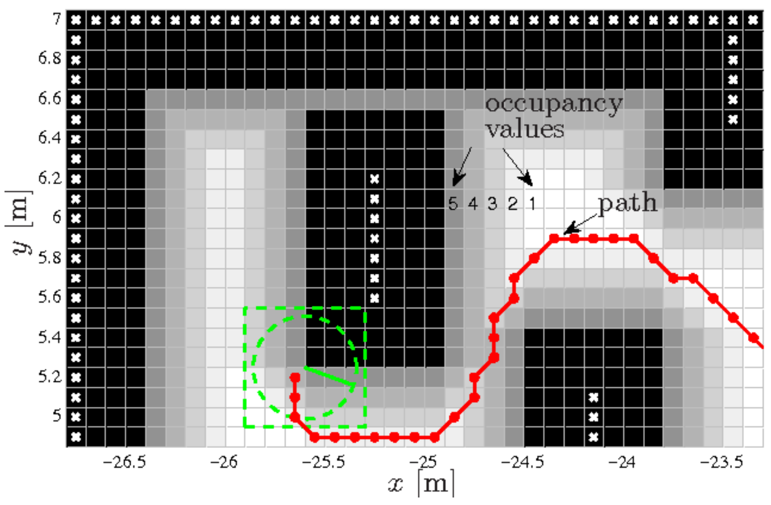


Figure 2.2. Example of gird scoring. From (Dakulovic & Petrović, 2011).

2.4.3 Graphs

Quoting from *Graph Theory: A Problem Oriented Approach* (Marcus & America, 2008): “A graph consists of points, which are called vertices (or nodes) and connections which are called edges and which are indicated by line segments or curves joining certain pairs of vertices.” In the graph in the Figure 2.3. we can see six vertices A, B, C, D, E, connected by five edges.

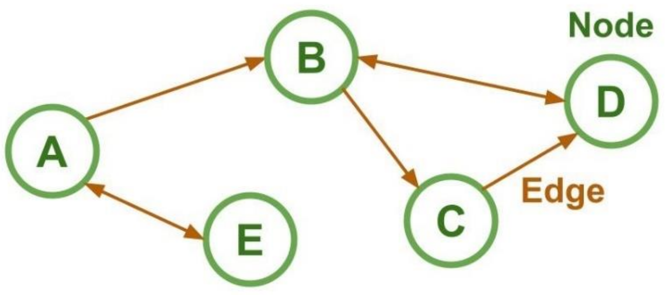


Figure 2.3. Example of a generic graph.

Edges can be identified by the names of the nodes they connect, so the edge connecting nodes B and C is named BC. Vertices connected by edges are called adjacent, so B is *adjacent* to C but not to E. The shape and length of the edge are completely irrelevant.

Throughout this entire discussion we will consider only *finite graphs*, that is, containing a finite amount of vertices and edges.

In principle, it is not necessary for edges in graphs to involve directions, but in our case it is instead an important feature, and the reason why will become clear shortly. Such graphs are for self-evident reasons referred to as *directed graphs*, or *digraphs*.

There are also graphs that contain a mixture of directed and undirected edges, called *mixed graphs*, but these will not be explored further.

In graph theory, a path is a sequence of vertices and edges in a graph about which it can be said to alternate constantly between vertices and edges, beginning and ending with vertices, and that each edge in the sequence joins vertices that follow one immediately after the other in the said sequence.

A feature of graphs that is not strictly necessary but indispensable for our purpose is the possibility of assigning each edge a "crossing cost". So far, we actually have considered each edge as equivalent, but they can be given a weight that makes them more or less easily traversable. The cost of getting from vertex $n3$ to vertex $n2$ may thus be different from the cost of getting from $n3$ to $n4$, which is obviously influential in choosing the best path to pursue. An example of such configuration is shown in Figure 2.4.

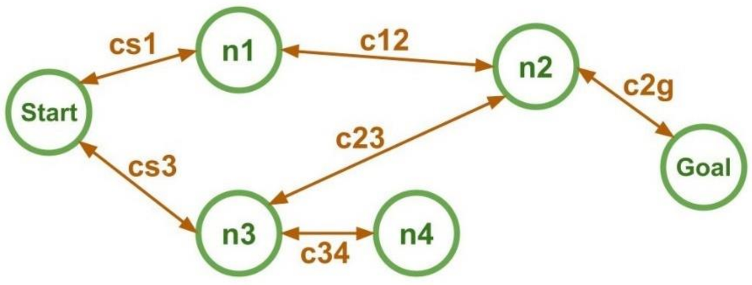


Figure 2.4. Graph with crossing costs.

Taking then into account the possible directionality of the edges described earlier, it is possible to have different crossing costs between the outbound and inbound. So for example looking at the Figure 2.5. it can be said that going from $n1$ to $n2$ has a cost c_{12} different from the c_{21} cost to make the opposite transition.

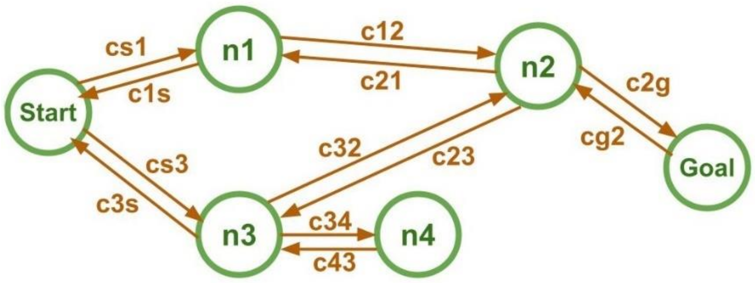


Figure 2.5. Graph with directional crossing cots.

From the extracted grid described earlier, the algorithm presented will have the task of creating a coherent graph. (Dakulovic & Petrović, 2011) Figure 2.6. helps with a visual representation to explain the process.

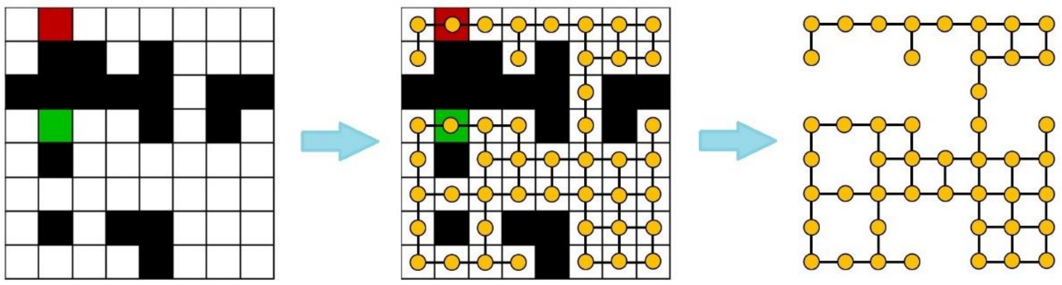


Figure 2.6. Generic generation of a graph from a grid.

Each of the cells will be associated with a node, the properties of which will be discussed in more detail later. All connections, i.e., edges, between the newly created nodes will then be created. Based on the coordinates, each node will be connected to its eight neighbors (8-edge connected graph), or less in case it is at the edge of the map or in areas that are not fully mapped. An example of the connections between eight nodes, belonging to as many adjacent cells, is shown in Figure 2.7. Each of these edges will be given traversal costs, the details of which may vary depending on the application, as we will see later.

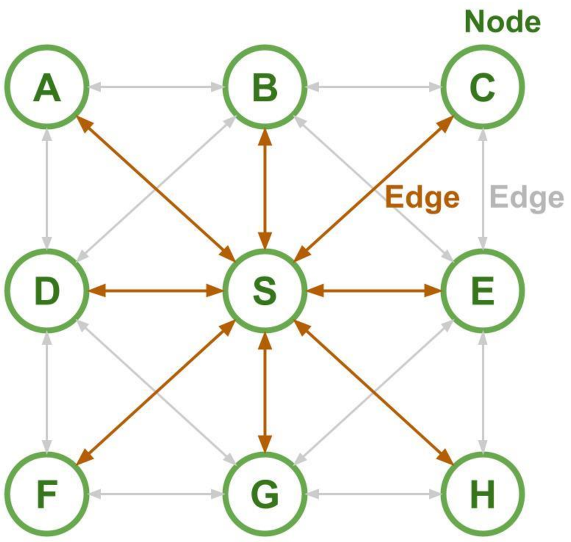


Figure 2.7. Example of connection between eight nodes.

In some cases, among which the one under consideration falls, costs are initially associated with cells, and thus with nodes instead of edges. It is therefore necessary to identify a conversion criterion from the score associated to a node to a crossing cost associated to its edge, and there are several currently considered valid. (Marcus & America, 2008)

They can be divided into isotropic and anisotropic, in the former the cost is the same for both directions of arc travel, while in the latter the direction of crossing is influential.

A simple example of anisotropy may be choosing as a cost the one of the arrival node (or, less often used, the one of the source node).

Examples of isotropic examples involve manipulating the two connected costs, such as selecting the higher one (or, less often, the lower), or computing the algebraic sum $cost_{12} = score_{n1} - score_{n2}$.

For this application, however, it was chosen to conform to the methodologies used by Nav2 in the ROS2 environment, i.e., assigning to the edge between two cells the maximum cost among those of the two cells themselves. This is obviously valid for each of the costs included in the analysis.

2.5 Replanning

An extremely important concern in path planners is the ability to recompute the path as a result of unexpected changes in the map. Each planner can react with different efficiencies to such changes, depending on how the algorithm is implemented. Some are forced to recalculate the entire route from the beginning, while others are able to take advantage of the information gathered to modify the one already obtained. Moreover, some algorithms can handle only changes and not additions or removals of nodes, while others react quickly to all types of changes.

In any case, in the context of graphs such modifications can be represented in one of the ways depicted in Figure 2.8., Figure 2.9., Figure 2.10., Figure 2.11.

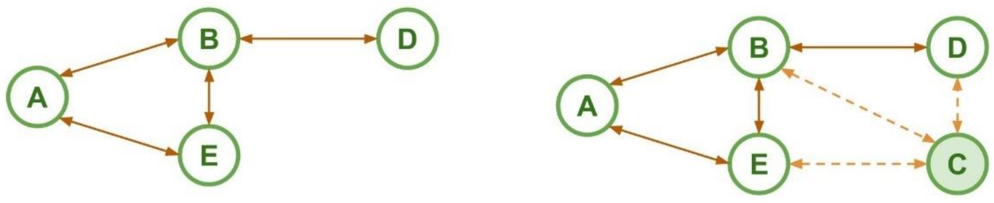


Figure 2.8. Creation of a new node.

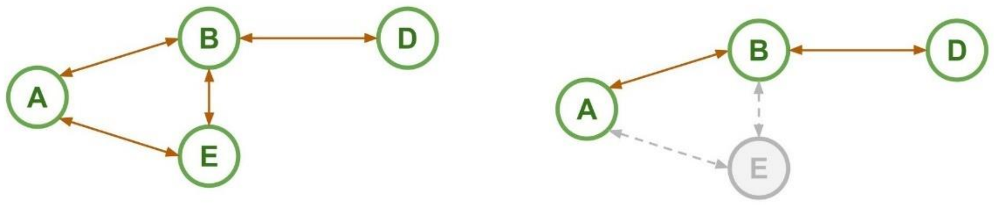


Figure 2.9. Destruction of a node.

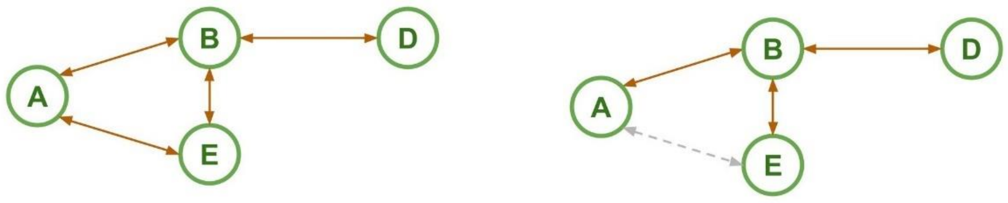


Figure 2.10. Deletion of ad edge, i.e. a connection between two nodes.

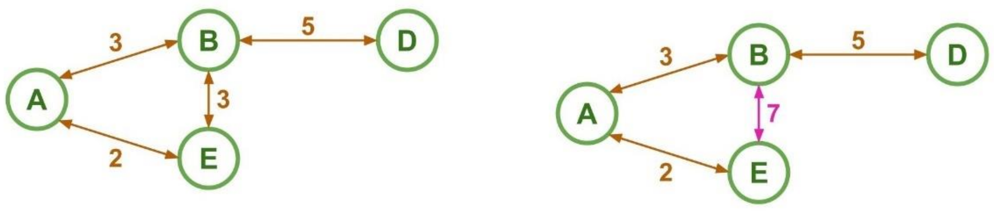


Figure 2.11. Change in the crossing cost of an edge.

When there is a need for replanning, it is particularly challenging if heuristic features are included. This will be explored in more detail in the section 3.3., in which the heuristic features are specifically examined.

We will see in the next section some examples of algorithms. It is emphasized that all of those discussed in depth possess the ability to perform replanning.

3. Search algorithms

Over the years, many different algorithms, more or less advanced and efficient, have been developed to address the path planning problem. The methodologies attempted are among the most diverse, and thus there are also different ways to categorize them, the one proposed in Figure 3.1. is suggested by (Petkov, et al., 2016).

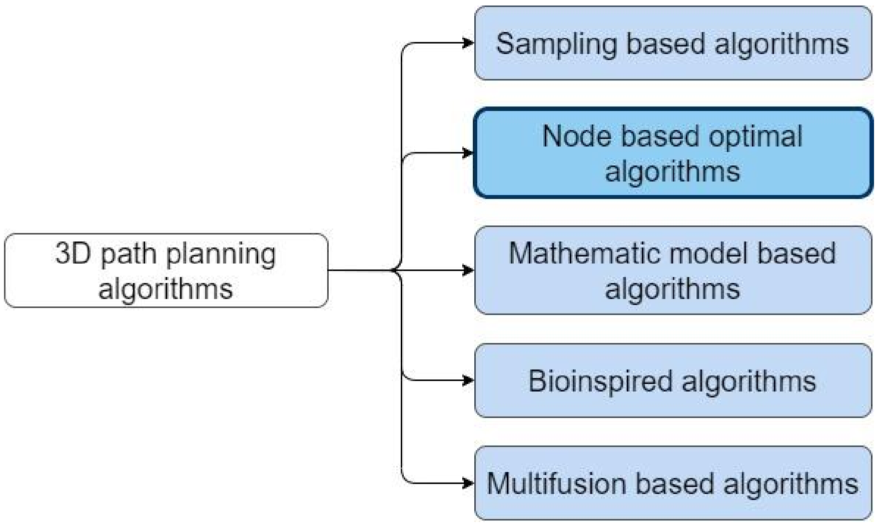

3D path planning algorithms
Sampling based algorithms
Node based optimal algorithms
Mathematic model based algorithms
Bioinspired algorithms
Multifusion based algorithms

Figure 3.1. 3D Path planning algorithm taxonomy.

This classification is actually valid for both path planning and trajectory planning. Where path planning deals with finding a continuous curve leading from one point to another. Trajectory planning, on the other hand, refers to successive planning concerned with how to move along the calculated path, thus having time as an additional variable.

In this discussion we will focus only on the category of Node based optimal algorithms, which best complies with the requirements. Algorithms that fall into this category share the habit of exploring within a set nodes/cells in a map. This is done by using maps on which data sensing and processing procedures have already been performed.

This type of algorithm is always able to find the optimal path, if one exists.

Algorithms referred to as node-based are also often referred to as network algorithms, alluding to the fact that they perform their research within the generated network.

In particular, the algorithms represented in Figure 3.2. are the ones that lead to the definition of MOD* Lite, the algorithm used for this application, so they will be examined in more detail.

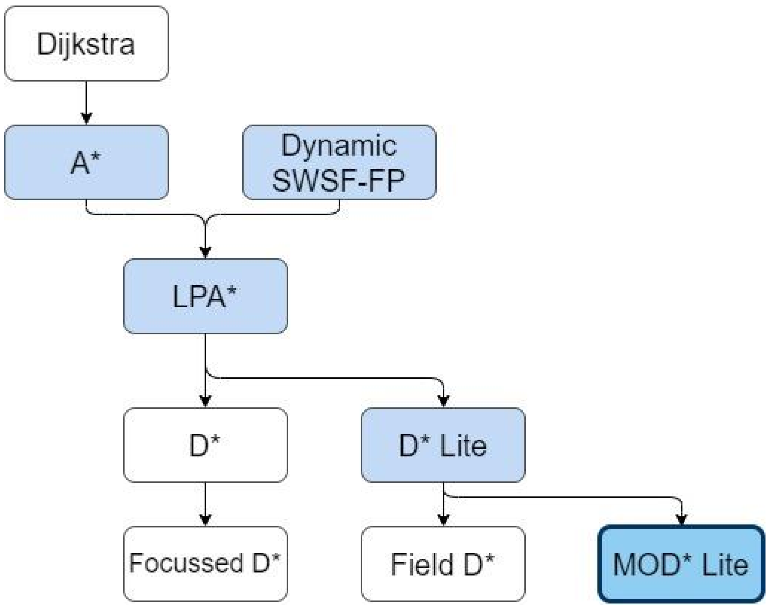


Figure 3.2. Derivation hierarchy of the MOD* Lite planner.

3.1 Single-objective algorithms

As already mentioned, the problem of finding the shortest path was one of the first to be addressed by means of algorithms, and so there are many solutions and versions of them that are more or less heavily modified. Some of those that led to the choices made in this thesis will be exposed below.

3.1.1 A*

Introduced in 1968 (Hart, Nilsson, & Raphael, 1968), the A* is an heuristic search algorithm that functions fundamentally like the classic Dijkstra's algorithm. To define the improvement introduced by the A* is the introduction of the heuristic aspect within the search.

“In 1964 Nils Nilsson invented a heuristic based approach to increase the speed of Dijkstra's algorithm. This algorithm was called A1. In 1967 Bertram Raphael made dramatic improvements upon this algorithm, calling it A2, but failed to show optimality. Then in 1968 Peter E. Hart introduced an argument that proved A2 was optimal when using a consistent heuristic with only minor changes. His proof of the algorithm also included a section that showed that the new A2 algorithm was the best algorithm possible given the conditions. He thus named the new algorithm in Kleene star syntax to be the algorithm that starts with A and includes all possible version numbers or A*” from (Nosrati, Karimi, & Hasanvand, 2012).

To briefly describe how it operates, it can be simply said that this algorithm keeps updated two lists, usually identified by the names "open" and "closed." The "closed" one keeps track of nodes that have already been examined, while the "open" one keeps track of nodes that have yet to be examined. Obviously, at first the former will be empty and the latter will instead contain only the starting node.

Each node (n) carries some essential information, namely, the cost to go from the start node to the node under consideration (which we will call $g(n)$), a heuristic estimate of the cost to go from the node in consideration to the goal node (which we will call $h(n)$), and a function that keeps in memory the estimate for the best possible solution passing through the node. Such a function can be simply defined as $f(n) = g(n) + h(n)$.

It also recalls a pointer pointing to its parent, that is the node from which to come to minimize the value of the function s $f(n)$.

The main loop of the algorithm selects time after time the node with the lowest value of f(n) from the open list, then generates all its possible successors (represented by n'). Each time one of them is examined, it is removed from the open list and placed on the closed list. For each of such successors, if they turn out to be already in the closed list with an estimate f equal or lower they can be safely discarded, and the same can be done if they turn out to be in the open list with an estimate f equal or lower. In fact, the first case means that it is a node that has already been examined and in that examination was part of a path leading to it having a better estimate. The second case tells us that we will have a chance later to re-examine this node and with a better chance of minimizing the path, so it would be unnecessarily wasteful to do so now.

If neither of these situations occurs, it is instead a case of removing each of its copies from the two lists and setting the current node n as the parent of n'. At this point the values of n' will be updated so that $g(n')$ equals g(n) plus the cost of getting from n to n', and $h(n')$ and $f(n')$ are calculated following their definitions. At this point n' is put back into the open list.

If the extracted node turns out to be the goal, the path can finally be said to be complete, so the solution is generated by going back from it to the start through the pointers to the parents. Otherwise, the cycle starts over again.

Thanks to its simplicity, while still managing to maintain good accuracy and speed of execution, A* is still one of the most widely used algorithms in robotics to this day. In fact, even the navigation2 package developed for ROS2 uses a slightly modified version of it as a path planner, A* Lite. The goals of the modifications are mostly to simplify and lighten the execution, without changing the basic operation.

3.1.2 Dynamic SWSF-FP

Dynamic *SWSF-FP* stands for Strict Weak Superior Function - Fixed-Point.

This algorithm was introduced in 1996 (Ramalingam & Reps) as an incremental search algorithm for the generalization of the Shortest-Path Problem. It represents a special case of the grammar problem, introduced in turn by Knuth (1977) as a generalization of the "single-source shortest-path search problem".

The important aspect introduced by this algorithm is the possibility of handling several and diverse changes at the same time. In fact, in a single update there can be a greater or smaller reorganization of the graph, including multiple changes in the cost of the edges but also addition and deletion of the edges themselves.

Algorithms already existed to consider one change at a time, whether it was an insertion, deletion, or modification, but compared to those methods, two ways are identified to greatly improve performance, which are "*combining*" and "*cancellation.*"

"*Combining*" refers to the case where updates are made by the repeated application of an algorithm for unit changes, which, however, can result in a vertex being examined numerous times, each of which will result in that vertex having semi-updated, temporary values. This waste of resources is eliminated in the algorithm for heterogeneous changes, which is able to combine all the necessary changes in a single run for each vertex.

"*Cancellation*" considers the case where an addition and a deletion have mutually counteracting effects, but despite this an algorithm for unit changes will have to perform more than one run anyway, again resulting in waste. Instead, these solutions make it possible, in a single update, to keep untouched the vertices that are not affected by the changes.

This algorithm is often and properly described as a "bounded incremental algorithm," which means that the time it takes to update the solution is bounded by a $||\delta||$ function, which is dependent on the *size* of the change.

This dynamic algorithm, in the case of the grammar problem, takes a time equal to $O(\|\delta\| (\log\|\delta\| + M))$ to find a solution, in which M represents the time limit required to compute any given product function.

3.1.3 Lifelong Planning A*

The Lifelong Planning A* algorithm, shortened as LPA*, (Koenig, Likhachev, & Furcy, Lifelong Planning A, 2004) leverages on the strengths and generalizes the dynamic characteristic of Dynamic SWSF-FP and the heuristic ones of A*. It can be applied to problems where the graph is finite and known from the beginning, but where edge costs may increase or decrease over time, which actually means that this feature can be used to work around the system and add or delete nodes.

LPA* is a *complete* algorithm, meaning it is always able to find the shortest path, if one exists.

It can be seen as an incremental version of A*, where a change in some traversal costs does not result in a complete rebuilding of the graph, but instead it is capable of adjusting to the changes and reducing the amount of recalculation needed.

As is the case in the A* algorithm, the shortest path can easily be traced using the constructed tree and going greedily up from cell to cell following the lowest costs. So during the first execution LPA* works exactly as A* does.

The differences are noticeable when something changes in the map, and to reduce the effort in the new search two different approaches are implemented.

In the first, following what happens in the Dynamic SWSF-FP, all distances to start that have not been changed are not recomputed, and this is done by keeping estimates of distances to goal instead of distances to start. In addition to this, the Dynamic SWSF-FP is stopped immediately when it is sure that it has found the shortest path from the start vertex to the goal.

In the second heuristic knowledge taken from the A* is used, applied in the approximation of distances to goal. And this information is used to discern which distances to start make sense to calculate and which can be promptly ignored.

To summarize its behavior, we can define N as the set containing all the vertices of the graph, $pred(n) \subseteq N$ will then represent the subset of predecessors of the vertex $n \in N$. The notation $g^*(n)$ is usually used to denote the distance of the generic vertex n from the start node, i.e., the cost of the shortest path leading from n_{start} to n. This definition must be subject to two conditions, namely verifying that $g^*(n_{start}) = 0$ and otherwise $g^*(n) = \min_{n' \in pred(n)}(g^*(n') + cost(n', n)$ in all cases where $n \neq n_{start}$. So g corresponds directly to the value of g used by the A* algorithm.

The addition is the definition of a value, usually called $rhs(n)$, that is a one-step lookahead based on the value of g itself, thus potentially being better informed at any point in the search. The name of this variable comes from the Dynamic SWSF-FP, where as seen before it represents the value in the *right-hand side* of the grammar rule. Its value will always be zero for the start vertex, while for all other cells it will be equal to the lowest cost g among all those offered by its neighbors plus the cost of going from that best neighbor to the cell under consideration.

Thus, for each cell the LPA* maintains these two estimates, g and rhs, of each cell's distance from the start vertex.

When a vertex is changed, which may involve a change in its cost or equivalently in its traversability, the vertices in its neighborhood are checked to see if their information is up to date. If they are found to be inconsistent, i.e., with $g \neq rhs$, it means that they need to be updated and are then placed in a priority queue, which will examine them one by one, starting with those most promising for the final path. For each modified vertex there will then be at least one of its neighbors to be updated, which creates a wave going back from the initial modification to the start vertex, passing only through cells that had been explored previously.

3.1.4 Declinations of the D*

Frequently when talking about the D* algorithm it is referred to any of the following incremental search algorithms:

- the original **D*** (Stentz, 1997) which is an *informed* incremental search algorithm. It is the Dynamic version of the A*, which locally modifies the results of previous searches, which allows a decrease of the total time of a search by up to one or two orders of magnitude compared to A*.
- **Focused D** (Stentz, 1995) is an *informed* incremental *heuristic* search algorithm that combines ideas of A and the original D*. **Focused D*** resulted from a further development of the original D*.
- **D* Lite** (Koenig & Likhachev, D* lite, 2002) is an incremental heuristic search algorithm that builds on LPA*.

The one being considered for this thesis is the last one, the D* Lite.

3.1.5 D* Lite

As mentioned earlier, D* Lite (Koenig & Likhachev, 2002) is an algorithm that essentially builds on LPA*, so that it is able to implement the same concepts introduced by D* but with an algorithmically different workflow. In particular it employs a shorter and simpler algorithm, while maintaining at least the same efficiency as D*.

Not negligible is then the fact that it also allows for more agile priority management, as it contains only one tie-breaking criterion.

The algorithm tries to find the path that minimizes the selected objectives (e.g., by looking for the shortest path), and then the vehicle will follow that path until it has arrived at its destination, or it will observe non-traversable cells on its path. In the second case, the best path will be recalculated from the current position, but retaining all information about the cells that do not appear to be altered.

For the descriptions that follow, it is necessary to introduce some important notations.

- n : generic node (or pointer to a node)
- $n' = n1$: parent of node n
- $n'' = n2$: parent of node n'
- N : finite set of nodes (= vertices) in the graph
- $succs(n), pred(n) \subseteq N$: set of successors / predecessors of the node n
- $c(n, n')$: actual cost of moving from node n to node $n' \in succs(n)$. It's always $0 < c(n, n') \leq \infty$
- $n_{start}, n_{goal} \in N$: start and goal nodes
- $g^*(n)$: distance from n_{start} to the selected node n
- $g(n)$: estimate of the distance $g^*(n)$
- $h(n)$: heuristic value associated to the selected node n, estimates the cost to traverse from n and n'. It has to respect the following restrictions:
    - $h(n_{goal}, n_{goal}) = 0$

- $h(n, n_{goal}) \leq c(n, n') + h(n', n_{goal})$ ⇒ valid for all nodes such that $n \in N$ and $n' \in succ(n),\ with\, n \neq n_{goal}$.
- $rhs(n)$: one-step-lookahead values based on the g-values (so potentially better informed than $g(n)$). It has to respect the following equation:

$$rhs(n) = \begin{cases} 0 & if\ n = n_{start} \\ min_{n' \in pred(n)} \big(g(n') + c(n', n)\big) & otherwise \end{cases} \tag{1}$$

If $g(n_{goal}) = \infty$ at the end of the search, then no path is constructed between n_{start} and n_{goal}. However, this would mean that there is no feasible path between those two nodes since the D* Lite algorithm is *complete*.

The path from n_{start} to any node n appearing in the selected path can be constructed by starting from said node n and tracing back, following repetitively the predecessors that minimize $g(n') + c(n', n)$.

A node is defined locally consistent if and only if it results $g(n) = rhs(n)$, conversely it is defined locally inconsistent if and only if it occurs $g(n) \neq rhs(n)$. This second situation can be further divided into the cases where it is locally overconsistent if and only if $g(n) > rhs(n)$ and locally underconsistent if and only if $g(n) < rhs(n)$.

When a node is found to be locally inconsistent, the algorithm will make sure to reprocess it to update its g-value and thus make it locally consistent.

The nodes that need to be processed are kept in a priority queue so that the most promising ones are handled first, allowing the total execution time to be greatly reduced compared to processing in random order. The criterion for ordering such queue takes into account several factors, namely the values of g, rhs and h, combined into a key value $k(n)$ as follows:

$$\begin{aligned} &k(s) = [k_1(n); k_2(n)] \\ &k_1(n) = \min\big(g(s), rhs(n)\big) + h(n, n_{goal}) + \mathrm{k_m} \\ &k_2(n) = \min\big(g(s), rhs(n)\big) \end{aligned} \tag{2}$$

To order the queue, the two components are considered in lexicographic order, so $k(n) \leq k(n')$ if and only if either $k_1(n) < k_1(n')$ or $k_1(n) = k_1(n')$ and $k_2(n) \leq k_2(n')$.

The first component of the key corresponds to what is called f-value in the A* algorithm, and the second component is what in A* is called g-value.

A heap reordering variable, called k_m, is included in the queue sorting rules to account for the rover's proceeding along its traversal. It will in fact be initially set to zero, but incremented whenever any edge cost changes, so as to discourage the algorithm from retracing the newly traversed sections. It is updated cumulatively with the heuristic distance between the goal and the current start node.

The key of each node is constantly maintained, updated each time one the g or rhs value changes.

During execution, the nodes expanded first are those with the smaller key.

3.2 Multi-objective algorithms

In the context of path planning, the vast majority of algorithms aim to minimize path length. Almost always, however, this is not the only parameter that needs to be kept in check. In general, it would be useful to take into account the safety of the route, the energy required to undertake it, perhaps considering that a steeper incline would result in a greater expenditure, or that in shaded areas solar panels would not be able to fully recharge the batteries. All these characteristics can obviously bring with them more or less strict minimum requirements.

In some cases, two or more criteria can be combined to give rise to a combined third criteria that takes both needs into account. But this is only possible when these criteria do not have conflicting needs.

For example, going over a terrain relief may be the best choice for the brevity of the route, but the worst for ensuring the integrity of the vehicle.

In our case the simple brevity of the route is combined with an aspect of route safety.

There exist many strategies for writing algorithms to deal with multi-objective problems, among which to name the classics can be mentioned the Weighted Sum methods, the ε-Constraint methods, and the Weighted Metric methods (Gunantara, 2018).

In particular then there are obviously algorithms for MPOs that focus specifically on path planning. Some of these are described below in more detail in order to frame the choices made for the development of the path planner in this thesis.

Before beginning the discussion, it is important to briefly define the concept of dominance between nodes, which in the case of multiobjective replaces the concept of best node (i.e., minor contribution to the objective function) that was used for single objectives.

3.2.1 MOGPP

Very often in real life we come across optimization problems that are NP-hard (Nondeterministic Polynomial-time hard), meaning that an optimal solution cannot be found in polynomial time. In these cases, evolutionary algorithms, which can be classified as "stochastic soft-computing methods," come to our aid.

Among them, MOGPP (Oral & Faruk, 2016) is a stochastic evolutionary algorithm, literally Multi-Objective Genetic Path Planning algorithm, which offers a soft computing genetic implementation that is able to compute paths taking into account the optimization of more than one objective. The algorithm still turns out to be complete, so it can always find a solution if one exists, but it does not guarantee that this solution is the optimal one.

In MOGPP, a path that consistently leads from the starting point to the target point is encoded in a chromosome, defining one of the solutions to the problem. In this genetic analogy, each gene in the chromosome represents a cell in the pathway, so the chromosomes can have variable lengths.

After randomly identifying paths that constitute alternative solutions, an evolutionary process that follows a fitness function is applied. Such function is defined as follows:

$$F(i) = \left[\frac{1}{L(i)^2}, \frac{1}{R(i)^2}\right] \tag{3}$$

In which, for a generic chromosome i, $F(i)$ represents its fitness, $L(i)$ the length of the path that it represents, and $R(i)$ an evaluation of the safety risk involved in that path. This is obviously in the case where the parameters to be optimized are the shortness and safety of the pathway, otherwise these values would be replaced by the current objectives.

To select the parents that will generate offspring chromosomes, through a crossover operation, a selection mechanism called roulette-wheel is used. The child chromosomes generated, will also represent valid paths.

Regarding the mutations, they are introduced by initially selecting during mating a random cell from the chromosome, which will be the point at which to divide the pathway into two sub-paths. Of these two alternatives, the one containing the target point is discarded, and in its place a random pathway is generated that terminates in the target.

All chromosomes are evaluated according to their fitness functions. After a predefined number of interactions, the algorithm is stopped and the paths that bring the best outcomes considering all objectives are selected.

3.2.2 SPEA2

SPEA is an acronym that stands for Strength Pareto Evolutionary Algorithm, which was presented in 1999 by Zitzler and Thiele. Seeing its remarkable results, an updated version was proposed in 2001, **SPEA2** (Zitzler, Laumanns, & Thiele), that would eliminate the inherent flaws of its predecessor and include some new discoveries in the sphere of MO algorithms. It indeed introduces the use of a "fitness assignment scheme" that takes into account for each element all the other elements it dominates or is dominated by. Methods for estimating the density of the nearest neighbor are also embedded, which enables more accurate operation of the search itself. And moreover, a new truncation technique is employed to ensure compliance with the boundary solutions.

SPEA is based on the use of a regular population and an external set, i.e., an archive, which is initially empty. The main loop of the algorithm sequentially performs the following steps: first, it copies all the non-dominated members of the population into the archive. All those dominated or with duplicate objective values are removed from the archive.

However, a limiting size for the elements that the archive can contain is defined, and if this is exceeded the excesses are eliminated following a clustering technique that leaves the non-dominating front unchanged.

Finally, *fitness values* are assigned to all members, according to different criteria for the archive and for the population.

Each element i in the archive receives a strength value $S(i)$ between 0 and 1 (excluded) equal to the number of population members that are dominated by or equal to i, divided by the population size plus one. $S(i)$ also represents the fitness value $F(i)$ of the same element.

For elements j in the population, the fitness value $F(j)$ is defined as the sum of all the $S(i)$ strength values of the archive members that dominate or are equal to j, plus one.

At this point binary tournaments have the job of carrying out the mating selection phase out of all the elements in the archive and population. Finally, the population is replaced by the offspring resulting from recombinations and mutations.

In addition, SPEA2 is able to use density information to its advantage through a more detailed determination of fitness values. Besides, for the occasion in which the non-dominance front exceeds the archive boundary, the clustering technique is replaced by a truncation method that avoids the loss of boundary points. The last essential difference lies in the fact that in this case only the elements of the archive participate in the mating selection process.

3.2.3 MOA*

Multi-Objective A* algorithm, introduced in 1991 (Stewart & White), is a generalization of the A* heuristic search algorithm described above. It falls into the category of offline algorithms, that is, it attempts to find the entire solution before starting the navigation.

MOA* has the capability of identifying all of non-dominated paths going from a specific starting node to a set of target nodes. Like A*, also MOA* is a *complete* algorithm when used with *admissible* heuristic functions.

The algorithm continues to use the sets called "*open*" and "*closed*," as is typical in heuristic searches. Each node is assigned three ratings, g, h and f. The need to provide that a node may have multiple parents (backpoints) leads to the use of some labels. Some definitions:

- $label_k(n', n)$ is a set of accrued costs of the paths from the start going through n and n' and which are not dominated by any of the detected paths up to iteration $k-1$.
- $solution\_costs_k$ is the set of costs associated with the current (iteration k) best solution identified.

- $g_k(n)$ is a set of “non-dominated accrued costs for node n” at any given iteration k. Basically it is the set of all the cost vectors for paths going from *start* to n that are not dominated, discovered within iteration k.
- $f_k(n)$ is the set of node selection values associated with node n during iteration k.

To summarize the unfolding of the algorithm, we begin by saying that the open list begins by containing only the start node and the closed list is initially empty.

A loop then deals with iteratively finding a subset of nodes found in open that have at least one value of f that is not dominated by any solution already found or by any other potential solutions still waiting in open.

If this subset turns out to be empty, we can say that the best solution identified so far is the definite one, and the cycle is terminated.

If, on the other hand, it is not empty, a node is chosen within it to be expanded, and this choice is made according to the heuristic function. That node is then removed from open and placed on the closed list.

At this stage, if the extracted node turns out to be the goal node, it is directly added to the current solution and its cost added to $solution\_costs$, removing from $solution\_costs$ itself any members that turn out to be dominated. In case it is an ordinary node, all its successors are generated. If there are none, again the cycle is terminated, otherwise an internal cycle is triggered to evaluate them one by one. If successor n' has been newly generated, a backpointer is instantiated from it and the value of $label(n', n)$ is assigned as defined above.

If, on the other hand, the successor had been already generated previously, for each path up to n' that was potentially undominated that was discovered it is checked that its cost is in $label(n', n)$ and in $g(n')$. If the cost was not already present in $g(n')$, then that cost and all those associated with paths leading to n' are purged from $label(n', n)$. If n' was present in closed, it is removed and moved to open.

Then the whole cycle is repeated until it is terminated for one of the reasons already described.

When MOA* is applied to problems with a single objective, the result is exactly the same as if A* was used.

3.2.4 MOD* Lite

The Multi-Objective D* Lite algorithm is an incremental algorithm which extends form the D* Lite described earlier.

It has been shown, for example by (Oral & Faruk, MOD* Lite: An Incremental Path Planning Algorithm Taking Care of Multiple Objectives, 2016), that it brings numerous advantages over its counterparts described in this thesis.

This algorithm was chosen for the development of this thesis for a few reasons of different nature. First, as far as integration into the SINAV project is concerned, it has until now only featured path planning algorithms with single objectives. Thus, the multiobjective feature represents an innovative approach to the problem and a challenge for subsequent integration with the mapping modules.

Among the many multi-objective algorithms then, MOD* Lite has advantages due to the specific strategies it employs. To begin with, it is a complete algorithm, so it always provides a solution if one exists, and it is capable of generating optimal or suboptimal but acceptable results in fully observable environments.

It is in general slightly less accurate than MOA*, but it offers worse execution speeds, which is indeed a cardinal aspect of the project. MOGPP, on the other hand, despite having very similar execution times to MOD* Lite, fails often to find optimal or suboptimal solutions, especially for large sized maps. This is of course because MOGPP does not guarantee Pareto optimality. By letting it run for very long times it is able to achieve acceptable solutions, but the requirement for speed in execution would be completely missed.

SPEA2 is, among the multiobjective algorithms described, the one that claims the best execution times. Unfortunately, however, this feature is also accompanied by the worst solutions among such algorithms, to the point of being very often unacceptable.

Therefore, the choice is narrowed down to MOA* and MOD* Lite. (Xue & Sun, 2018) shows, however, that MOD* Lite is able to provide better results even in the case of partially observable or dynamic environments. The choice therefore fell on this one.

The details of the characteristics of MOD* Lite and its functioning will be described in section 4.

3.3 Heuristic functions

Whenever the need to identify a heuristic approximation within the algorithms has been mentioned, it has not been specified how this would be implemented. This is because there are for each case several possible choices, more or less appropriate depending on the problem being addressed. A good heuristic function will allow the algorithm to turn around quickly and find the optimal solution, while a poor one might lead to increasing the execution time without bringing any benefit, or even lead the search to wrong choices, risking to find sub-optimal solutions or no solution at all.

To discern among the possibilities and choose the best heuristic function, it must first be admissible, which means that it must never overestimate the cost required to get from start to goal. For if this were to happen, the cost of the optimal solution might be greater than the actual cost, and this would lead that solution to be discarded over one that is actually worse.

Naturally, the function cannot have excessively low values either, otherwise it would fail to concentrate the expansion of nodes towards the goal node.

Another extremely important aspect to keep in mind is the speed with which the result of the heuristic function can be computed while still maintaining sufficient accuracy. The point for a fair compromise can vary depending on both the specific application and the algorithm in use. For example, some applications may have no time limit but need an extremely precise path, while others may necessarily require a blazing-fast response. In general, it has been proven that in most cases having a function that comes up with a good estimate while taking a short time to do so is far preferable to one that obtains a perfect estimate but takes a huge amount of time to do it.

In choosing the heuristic function to be used for this thesis, in addition to the elements just considered, it was also taken into account the ease with which it could then integrate with the simulation environments. It was therefore chosen to include the heuristic component, as we shall see in MOD* Lite, in the form of the distance between a given node and the goal node.

The function is simply defined as the area line distance between the two nodes, therefore using only the coordinate values to compute it. This makes it a fast computation and certainly admissible, as each cell will then have an additional cost based on its slope or exposure. In the event that the terrain turns out to be perfectly flat, at most the heuristic and true value will be equivalent.

As mentioned in section 2.5., the case of heuristic features being involved in replanning brings up several problems. Let us return for a moment to the problem mentioned earlier about avoiding assigning too low values to heuristic functions. In the case of replanning, it means that the traversability cost of at least one node has changed between one cycle and the next. This means that the minimum cost in the map must be recomputed each time, because it could be increased or decreased. This computation has a linear cost for each of the nodes that have changed. Thus, if heuristic values are included to calculate the priority of the nodes to be analysed, which happens very often, all the values ruling such queue should be recomputed too.

For this problem (Koenig & Likhachev, D* lite, 2002) proposes as a solution the use of an auxiliary variable. A "key modifier" that is incremented each time the rover makes a move. Its definition and use are discussed in more detail in the section 4. In this solution, the heuristic values are quickly updated at the beginning of each cycle by adding to them the current key modifier.

This value should theoretically be subtracted as the vehicle approaches the goal and distances decrease. However, this would involve recalculating all the keys, which could be decreased only within certain limits. Therefore, to make the process faster, it is chosen to perform the reverse operation, adding a uniform offset to the queue. It must be kept in mind, however, that this solution leads to error accumulation in the long run.

4. MOD* Lite

MOD* Lite is a domain-independent search algorithm that can be used whenever the surrounding environment is at least partially observable. The step up from D* Lite is to allow one to define a set of objectives instead of a single one, and for each of them it can be decided whether to minimize or maximize it in the trajectory.

Two objectives that are mutually independent are considered in this study, but the algorithm presented can be applied to a larger number of objectives without any modification (except, of course, the need for more information to be fed to the system).

However, the first criterion is itself made up of a combination of two objectives, namely, the brevity of the route and its safety. This is done by letting each cell have the cost of it being reached from one of its neighbours (brevity) added to the cost related to the local slope (safety).

MOD* Lite inherits all the variables previously considered in the explanation of D* Lite.

In this application, the simple line distance between the two points under consideration was chosen as the heuristic function $h(n, n')$, but this choice can be changed without altering the operation of the whole algorithm.

However, some substantial differences must be pointed out. Primarily, the variable cost must now contain more than one value (for us, two values) per cell, and will thus change from being a single variable to being a vector. Same fate will obviously have g and rhs, which in particular becomes:

$$rhs(n) = \begin{cases} \text{ObjectiveBase} & if\ n = n_{start} \\ \text{nonDom}_{n' \in pred(n)}\ \text{sum}\big(g(n'), c(n', n)\big) & otherwise \end{cases} \tag{4}$$

Where ObjectiveBase is a base vector with as many entries as many objectives we are considering, each of which will be equal to zero if the corresponding objective is to be minimized and equal to inf if it is to be maximized.

An important difference created by having vector costs instead of single values is the fact that the concepts of minimum and maximum are no longer so easily applicable. Indeed, there may be a situation in which considering the first elements of both vectors results in the first being minor, but considering the second the situation is reversed; taking, for example, two vectors $a = (3,4)$ and $b = (5,2)$ we have that for the first elements $3 < 5$, but for the seconds $4 > 2$, so neither vector can be said to be less or greater than the other. This situation falls under what will henceforth be described as non-dominance.

In the case where dominance can be instead established, it can still be further described. With $a = (3,4)$ and $b = (2,1)$ we find ourselves in the situation where a completely dominates b, since every value of a is less than its corresponding value in b, while with $a = (3,4)$ and $b = (5,4)$ we will simply say that a dominates b. Of course, the case of equality, in which all values of the two vectors are two by two equivalent, remains in effect.

From here on, we will use the concept of dominance in two quite different situations. The first, more straightforward, is used to describe the relationship between precisely two vectors, most often g and rhs, or a combination of c and g. But dominance terminology will also be applied with improper language to a pair of nodes. In this case what will actually be compared are the keys of the two nodes under investigation, as these keys can be seen as two-element vectors.

4.1 Structure of Node class

A C++ class was tailored to accurately and adequately represent each node.

The informations it carries include some intrinsic variables, such as a pair structure that stores its coordinates, the information about whether it is the start node, the goal one or any other, a vector for each between $cost$, g, rhs values. Another pair structure contains the key that will be used for sorting in the queue.

Then it contains a vector that remembers the pointers to all adjacent nodes (for, of course, a maximum of 8) and an unordered map that keeps in memory all the parents so far detected.

The only criterion it possesses is the minor operator (<), which allows the priority queue to automatically sort itself according to the rules described here. As mentioned earlier, sorting involves the lexicographic comparison of the keys.

A parallel class to this, dummyNode, was created as a transient container to be used when reading new maps. Its purpose is explained later in the procedure UpdateMap.

4.2 Priority queue

As in D* Lite, MOD* Lite also maintains a priority queue of all the inconsistent nodes, sorted by a combination of g, rhs and h, so that the most promising vertices are expanded first. The sorting criterion also remains the same, based on the keys already described above, with the only change being on the type of variable containing the various costs (from singular values to vectors). At the top of the queue will be the nodes with the minor keys, which could therefore make the final path as little worse as possible, in fact you will see in the description of the algorithm how the elements to be expanded will be extracted precisely from the top of the queue.

To fulfill the functionality of this variable, several data structures offered by different libraries were considered. Through consultation of some benchmarks (one of which is found in (Benchmark of major hash maps implementations, s.d.)) and personal testing of `std::set`, `stl::priority_queue`, and `boost::heap::fibonacci_heap`, the final choice fell on `std::set`. Indeed, this structure offered the best assortment of features for its assigned purpose and the best performances.

4.3 Functions

- ***domination(v1, v2)*** : given two vectors (which will always have the same length), this function returns the type of dominance between the two. The results can be either of the following:

  the first vector completely dominates the second, the first vector dominates the second, the vectors are equal, the second vector dominates the first, the second vector completely dominates the first, neither vector dominates the other.

- ***single_ domination(f1, f2)*** : does the same thing as domination(v1, v2) but for individual values. The results options here are obviously reduced: the first value completely dominates the second, the vectors are equal, the second value completely dominates the first.

  This division between the two functions is necessary to avoid ambiguity in the case of singular values, and to facilitate the writing considering the different types of variables in use.

- ***nonDom(g, rhs)*** : return the non-dominated value between the given g and rhs values. If they're equal or none dominates, the default return is the rhs value.

- ***vector_ sum(v1,v2)*** : simply performs the element-by-element sum of the two vectors, but in addition handles the case where one of them is a null vector, returning the other one as the result.

- ***heuristic(n)*** : represents the heuristic function described earlier, thus with the purpose of approximating values. In the definition chosen, it returns as the value the shortest aerial path length between the node provided as input and the starting node, ignoring the fact that the path will only be able to move from one cell to another.

- ***calculateKey(n)*** : calculates and updates the key values of the input node, following the formulations described in the Eq. (2).

- ***compute_ cost(n)*** : as the name suggests, computes and returns the cost vector to be assigned to the edge connecting the two nodes in input.

The rule chosen is to select the largest cost from the two initially assigned to the two nodes.

- ***findAdjacents(n)*** & ***addAdj(n,coord)*** : these two functions are respectively concerned with finding out whether the eight nodes adjacent to the node in input exist in the grid, and if so, inserting them into the vector that accounts for them.
- ***nonDom_succs(n)*** : this function is used in two slightly different ways. In the first, it is responsible for searching and returning for all successors of the input node that are non-dominated with respect to the multiobjective consisting of $c + g$. In the second, it instead returns the minimum value of the same multiobjective $c + g$ that it can find among all the successors of the input node.

  So the search is the same in both cases, what changes is just the type of output.
- ***update_rhs(n)*** : when called, this function updates the rhs value of the input node with the minimum that can be obtained from the successors (via the nonDom_succs(n) function described earlier), unless the node is the goal, in which case in fact the rhs value is never updated.

  It then recalculates the key values for the node and places it back in the appropriate position of the priority queue.
- ***updateAdjacents(n)*** : simply calls the update_rhs(n) function for all nodes adjacent to the one entered as input.
- ***start_doesNot_dominate(n)*** : is a support function that verifies that the input node is not dominated by the start node. Its practicality will be clear later.

4.4 Procedures

This section describes the main procedures that the algorithm has to carry out during its operation, obviously translated into the form of functions.

4.4.1 Main procedure

This is the main function that is entered as soon as the program is launched.

First, the data is extracted from the first map, and once the choice of the goal node is received, it is placed in the priority queue as the first element. Now the procedure computeMOPaths can be executed for the first time.

At this point a loop begins that will continue until the goal is reached by the vehicle. It consists of collecting all the paths that are generated by the generateMOPaths procedure and, if it turns out to be non-empty, presenting it as the output. In the case that it turns out to be empty instead, the user will be alerted to the temporary inability to trace a consistent path, and the algorithm will remain waiting for changes in the map.

In both cases, the following execution is expected to present changes in the map or the start and/or goal points, so the priority queue is emptied and the updateMap function is called again.

Procedure 1. Main of the MOD* Lite algorithm

```
function MODLite()
    updateMap()
    calculateKey($n_{goal}$)
    queue.insert($n_{goal}$)
    computeMOPaths()

    while($n_{start}$ != $n_{goal}$)
        solutionPaths = generateMOPaths()
        if(solutionPaths.empty())
            no solution found
        else
            send path

        updateMap()
```

4.4.2 Compute MO Paths

This function deals with giving all nodes eligible to be part of the final path the correct g-values, taking into account all objectives to be optimized. Starting from the objective node, we trace back to the starting node.

This procedure requires no input and returns no output, but it performs a series of operations and alterations on the variables involved.

As soon as the function is called, it first updates the key of the starting node, because such value is needed for the check that will be now described.

Then it begins a loop that continues until it happens that the starting node dominates the node that is at the top of the priority queue, or of course that queue is empty.

After passing this check, the first element from the priority queue is extracted, and its key is updated to account for any changes that have occurred between its inclusion in the queue and the current manipulation. If that new key is found to be greater than the initial one, it means that it is probably no longer the best element to expand and is therefore reinserted in the most appropriate place in the queue.

Otherwise, its consistency is checked. If the node in question is overconsistent ($rhs < g$) its value of g can be automatically updated to match that of rhs. Then nodes adjacent to it are updated, which will obviously be affected by this change in cost.

If it is found to be underconsistent ($rhs > g$), it means that the existing value of g is no longer valid and is therefore reset to be infinite. In this case, in addition to updating adjacent nodes, the function is called to also refresh the rhs value of the node itself.

The third and final possible case is that none of rhs and g completely dominate the other (local non-consistency), so the node's g is set equal to the non-dominated element between the two, and the adjacent nodes are updated.

Procedure 2. Compute MO Paths

```
1:  function ComputeMOPaths()
2:      calculateKey($n_{start}$)
3:      while(!queue.empty() &&
                          start_doesNot_dominate(queue.top))
4:          $n_{oldKey}$ = queue.top
5:          queue.erase(queue.top)
6:          $n_{newKey}$ = allNodes.find($n_{oldKey}$) //search the
              corresp. node in the list using the coordinates
7:          calculateKey($n_{newKey}$)

8:          if ($n_{oldKey}$ < $n_{newKey}$)
9:              queue.insert($n_{newKey}$)     //put it back in queue
10:         else if ($n_{newKey}$.rhs completely dominates $n_{newKey}$.g)
11:             $n_{newKey}$.g = $n_{newKey}$.rhs
12:             updateAdjacents($n_{newKey}$)
13:         else if ($n_{newKey}$.g completely dominates $n_{newKey}$.rhs)
14:             $n_{newKey}$.g = $\infty$
15:             update_rhs($n_{newKey}$)
16:             updateAdjacents($n_{newKey}$)
17:         else
18:             $n_{newKey}$.g = nonDom($n_{newKey}$.g, $n_{newKey}$.rhs)
20:             updateAdjacents($n_{newKey}$)
```

4.4.3 Generate MO Paths

Using the g-values determined by computeMOPaths, this function takes care of putting together the actual path, this time starting from the start node and proceeding to the goal.

Internally to this function, a queue named expandingStates is maintained to keep track of which nodes in the neighborhood need to be updated, and it follows a simple First-In-First-Out sorting. The first element inserted in such queue is the start node.

This procedure can be essentially divided into two phases. The first one, involves a loop that examines the entire expandingStates queue until it is emptied.

Once an element is extracted (which will be symbolized by n), its non-dominated successors are identified (through the nonDom_succs function described earlier) and are processed one at a time. Each of these successors is symbolized by $n1$.

If n has no parents, and this is only the case when n is the start node, surely its successors will also not yet have parents, so n is added as a parent of n' with cost equal to the cost to traverse from n to n'.

Otherwise, if node n already has any defined parents, they are used to compute an auxiliary value called cumulative cost, in the code represented by the vector cumulativeCs. It consists, as the name suggests, of the cumulative sum of all the elements of the cost vectors associated with all the parents, plus yet the multiobjective cost of going from n to its successor n' under consideration.

At this point, if n' is found to have no parents assigned yet, it is assigned the node n as a parent with cost equal to cumulativeCs. Otherwise, it is checked whether or not the node n with cumulativeCs cost would be a clear improvement over the parents already possessed. If yes, this is replaced. If, on the other hand, it is found to be in a case of non-complete dominance, each of the values in cumulativeCs is compared with each value of the parent n'' costs. In case of equality or dominance, the corresponding cost is removed from the list. At the end of the comparison, if all the costs of n'' are dominated, n'' it is removed from the parents of n', and if cumulativeCs still contains non-dominated costs n is added as the parent of n'.

The pseudo code described in Procedure 3 may help to understand the flow of this process.

At the end of the first phase, if n is among the parents of n' and n' is not already in the expansion queue, it is placed in it.

In the second phase the actual output, namely the vector of pointers solutionPaths, is composed. As the last element, of course, is inserted the final goal of the path.

Then, until the start node is reached, the parents of each node are examined by going backwards.

For each node, the parent node that dominates or completely dominates the others is identified, it is inserted into the solution vector and set as the target for the next loop.

If the node being examined has no parent, it means that a complete path could not be found, so the solution vector is emptied and the function is terminated.

Procedure 3. GenerateMOPaths()

```
 1: function GenerateMOPaths()
    //FIRST PHASE
 2:     expandingStates.push_back($n_{start}$)

 3:     while (!expandingStates.empty())
 4:         n = expandingStates.front()
 5:         nonDomSuccs = nonDom_succs(n)

 6:         for (n’ : nonDomSuccs)
 7:             if (n.parents.empty())   //iff n = $n_{start}$
 8:                 n’.parents.insert(p=n, c=cost(n, n’))
 9:             else
10:                 for (n’ : n.parents)
```

```
11:                  cumulativeCs = sum$_{allParents}$(cost) +
                                                    cost(n,n')

12:                  if (n'.parents.empty())
13:                      n'.parents.insert(p=n,
                                                c=cumulativeCs)
14:                  else
15:                      for (n'' : n'.parents)
16:                          if (n''.cost = cumulativeCs  ||
                 n''.cost completely dominates cumulativeCs)
17:                              break
18:                          else if (cumulativeCs
                               completely dominates n''.cost)
20:                              n'.parents.erase(n'')
21:                              n'.parents.insert(p=n,
                                                c=cumulativeCs)
22:                          else {
23:                Ifor (cC : cumulativeCs)
24:                    for (eC : n'.parents(n'').cost)
25:                        if (cC = eC  ||  eC dominates cC)
26:                            cumulativeCs.erase(eC)
27:                            break
28:                        else if (cC dominates eC)
29:                            cumulativeCs.erase(cC)
30:                            break

31:                if (!cumulativeCs.empty())
32:                    n'.parents.insert(p=n,c=cumulativeCs)
                                  }

33:          if (n'.parents.contains(n) &&
                               !expandingStates.contains(n'))
34:             expandingStates.push_back(n')

    //SECOND PHASE
35:    n = n$_{goal}$
36:    solutionPaths.push_back(n)
```

```
while (n != $n_{start}$)
    if (n.parents.empty())     //failed to generate path
        solutionPaths.clear();
        return

    for (n’ : n.parents)
        min_parent = *parent with the minimum cost*
    solutionPaths.push_back(min_parent)
    n = min_parent

return solutionPaths
```

4.4.4 Update Map

The first task performed by this procedure is to call the ReadMap function, which reads the most recent map and extracts data from it for all cells. The nodes of the new map are at this point all contained in a list called newMap.

Each of the nodes in this list is examined, first by determining whether or not it is already among the ones in the map used up to the previous run. If no node with the same coordinates is found, presumably the camera has detected peripheral areas never explored before and it is obviously added to the list and the flag signaling a change in the map is raised.

On the other hand, if the coordinates are already associated with a node, the other characteristics, namely node type and cost, are compared. If a change is detected, the node is updated and the flag for the change is raised. In this case, however, other operations are required, which are useful for keeping the various connections in order, i.e., updating the *rhs* value and resetting parent relationships. In addition, edges connecting such nodes to their neighbours must be updated.

In the event that a cell undergoes a change that makes it no longer traversable, the node associated with it will not be destroyed, but its cost will be set equal to infinity, which will automatically exclude it from any possible path.

If the vehicle has moved, that is the coordinates of the start node have changed, k_m is increased by a value equal to the heuristic distance between the start and the goal.

Finally, the computeMOPaths procedure is called again.

Procedure 4. Update Map()

```
function UpdateMap()
    ReadMap()
    for (n : newMap)
        if (n not already in allNodes list)
            allNodes.add(n)
            newNodes.push_back(n) //keep track of new cells
            nodes_changes = true
        else
            if (n.cost || n.nodeType have been modified)
                update cost or nodeType
                update_rhs(n)
                n.parents.clear()
                nodes_changes = true

    if (nodes_changes)
        for (nn : newNodes)
            findAdjacents(nn)

        if (vehicle_moved)
            k_m += heuristic($n_{goal}$)

        computeMOPaths()
```

4.4.5 ReadMap

This function is responsible for translating the maps that are fed to the algorithm, whatever format they are in.

Section 6. describes in detail all the tests performed, and for each of those this function took on slightly different characteristics.

Specifically, in the initial case of manually constructed maps, this function handled exactly this construction.

In the case of tests with custom random generated maps, these were in the form of images in bitmap format, with each pixel representing a cell. The numerical value in greyscale of each of these pixels would then represent the altitude, slope, or solar exposure score of the corresponding cell. The ReadMap function would simply pick up these values one by one, so that they would be usable by the algorithm.

In addition, to make the simulation more realistic, it was thought to provide the algorithm with a less detailed global map, which would however be more accurate around the vicinity of the rover. Here, too, the ReadMap function took care of reading the data provided by two parallel images, from the more blurred one first, updating only the cells around the current position with pixels from the more focused image.

Of course, to consider the multiobjective feature of the algorithm, a different image was provided for each objective, whose data was then merged into those of the various nodes.

The starting and ending points of the wanted path, could be provided by the user at the beginning of the simulation or set intrinsically. The same applies to the update of the rover's position during the traverse.

The same function was then updated and modified to accommodate the needs of simulations with ROS2, which involved interacting with the map input from other Nav2 packages.

5. Integration with TASI

The exploration of path planning algorithms and the subsequent operational implementation of MOD* Lite were driven not only by the purposes of research but also, and more importantly, by the objective of integrating such a planner into a real-world application.

Specifically, the system aimed at accommodating such integration is part of a TAS-I program. Thales Alenia Space, in collaboration with the Polytechnic University of Turin, is carrying out the SINAV research project for rovers aimed at space exploration.

The rover in question is an Adept Mobile Robots Seekur Jr under their ownership, constantly tested, both in its software and hardware features, in their RoXY facility. RoXY, acronym of Rover eXploration facility, is a structure in the TAS-I Turin complex that houses a reconstruction of the Martian terrain and the various challenges that it may present.

Figure 5.1. shows schematically the complete system that regulates the operation of the Autonomous Navigation thought and designed for SINAV.

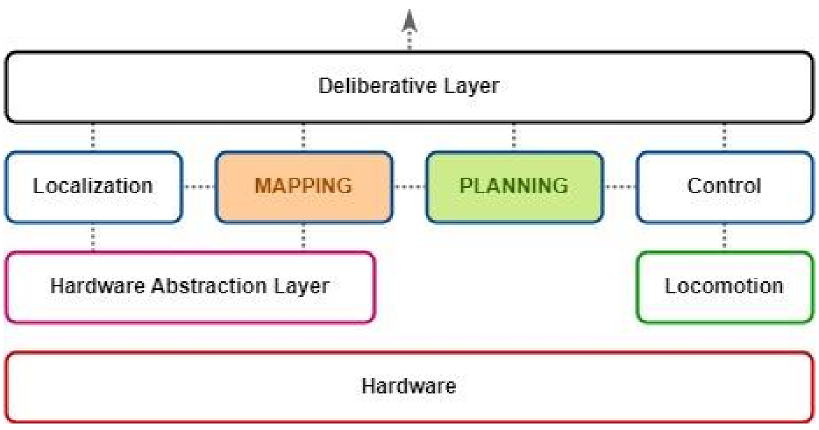


Figure 5.1. AutoNav system architecture.

The HAL, composed of ROS nodes, provides the framework compatibility to all the sensors integrated on the rover.

The localization subsystem collects data from the several sensors and the wheel odometry and from these calculates an estimate of the rover's position. The mapping subsystem collects the depth information and translates it into a Traversability Score Map. The planning subsystem, which is the one mostly considered in this thesis, is responsible for providing a consistent and safe path to traverse the map. The control subsystem is responsible for following to the best of the rover's ability the path provided by the planner, correcting any errors between the desired trajectory and the one actually followed. Finally, the locomotion layer translates everything into explicit commands for each wheel to achieve the necessary movements.

The deliberative layer consists of a behavior tree or user-supplied commands.

The rover's locomotion system consists of four wheels with each side having its two corresponding wheels driven by a single motor. The movements are therefore regulated by skid-steering, which leads to some additional difficulties in odometry measurements, because two instantaneous rotation centres (ICRs) must be taken into account for the computations, which then vary over time.

If, in addition to this, one considers the inevitable involuntary slips that the rover undergoes during its traverse, mechanical odometry, albeit at a high publish rate, can certainly provide a valuable aid but cannot be considered sufficient to verify the performed motions.

As for the rover's perception of its surroundings, it is equipped with several sensors. In particular:

- Stereo-camera : *StereoLabs ZED 2*;
- ToF : *LucidLabs Helios2+*;
- IMU : *Xsens 630*;

With these, an adequate localization of the rover and a description of its environment can be obtained.

In particular, stereo-cameras are able to provide a complete description of the terrain in the form of a point-cloud.

Thus, the element with which the developed planner must interface regarding the perception of the external environment are maps, already captured and partially processed. Concealed in each of these are the values to be used to evaluate each cell according to the objectives to be minimized that have been selected.

Considering the planners already implemented and tested in SINAV, it was decided to opt for a more complex planner that would take more than one element into account, as opposed to classic planners that optimize only for the shortest path.

As for the choice of objectives to optimize, it came down to finding the shortest, safest route that offers the most solar exposure. The latter, proves to be relevant in the case of Martian rovers that rely much of their energy on what they gain from solar panels.

In Figure 5.2. is depicted the flow of information leading from the perception of the external world to the implementation of the desired movements.

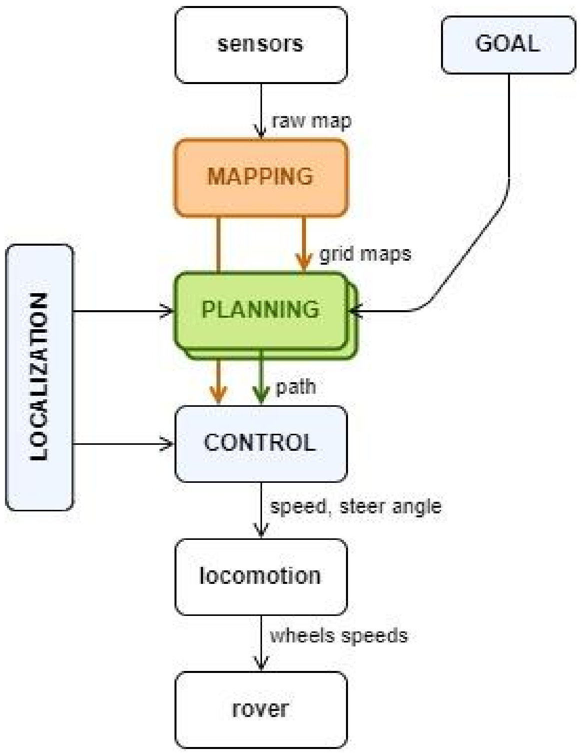


Figure 5.2. Flow of information between the AutoNav systems.

For this thesis, the general operation and strategies used in the state-of-the-art MOD* Lite algorithm were studied in depth. This study was used to be able to write out a code that would fulfill all its functionalities in a complete planner. For this purpose, the C++ language was chosen to be used, which allowed to obtain a suitably efficient software.

After thoroughly testing the operation of the generated code, followed a phase of optimizing it from the standpoint of solution accuracy, execution speed and memory occupancy.

Then the entire code was wrapped in a plugin that would make it possible to include this planner in the simulation environment used in the SINAV project, namely ROS2's Nav2 package. This will be discussed in more detail in the section 6.4 .

5.1 Simulation environment

Before getting to test the various software in RoXY, computer-simulated tests are performed, so as to initially minimize the risks of damaging the equipment and, above all, to have a more controlled environment in which to be sure of proper operation. In this way it is also possible to simulate extreme situations that would be difficult, expensive or dangerous to reproduce in real life.

ROS2, widely used in space research, is also the tool used in SINAV to fulfill this purpose.

The Nav2 project, developed for ROS2, has its roots in the benefits found by the Navigation Stack used in ROS. It is useful in all applications involving robotic navigation, the most common of which is to find a safe route for a mobile robot to move between two points. But at the same time it is useful for the purpose of following dynamic points. "This will complete dynamic path planning, compute velocities for motors, avoid obstacles, and structure recovery behaviors."

Each action is represented by its own separate node, which communicates with the BT through ROS2 action servers.

To create complex navigation behaviors, it is possible to combine more than one controller plugins, planners and recoveries in each of its servers.

In order to insert the built planner into the Nav2 environment, it is necessary to create a new ROS2 package that contains it. A plugin was written for this purpose, which would then allow interaction between the new package and the rest of the environment. To simulate the other functionality needed for navigation, the already available packages were exploited, for localization, mapping and control.

6. Testing

To comply with the computational limitations outlined above and to simplify integration with the rest of the system, the main body of the planner was written using the C++ language.

First, all necessary unit tests were performed to ensure the proper functioning of each function under all verifiable conditions.

The earliest practical tests were carried out on small, manually generated maps, so that the expected global situation could be known precisely at each moment of execution, and thus check the correct operation of each function.

Later a short script was written using Python language. This was done in order to be able to perform some tests that would ensure the greatest amount of scenarios covered. Such a script is able to generate in the form of bitmap images a large number of random maps, while still respecting parameters to make them realistic. For example, a Gaussian function was used to soften the fluctuations in values. It is also possible to tune some parameters, such as total area, average slope variation and of course the amount of maps to generate.

The results for altitudes and those for slopes are connected by a function that through a derivative obtains the latter from the former.

The bitmap images are then read by the planner (a mechanism that mirrors the behavior it should have during normal operation), which then gets the necessary data from them and turns them into nodes.

Subsequently, the tests were moved to the ROS2 environment for greater integration with the other components that form an autonomous navigation system.

All these tests will now be exposed in more detail.

6.1 Hand generated maps

As mentioned, the first tests were carried out through small maps crafted purposely by hand, each with specific characteristics. In fact, in this way it was possible to verify the correct functioning of the algorithm not only in its entirety, but also by looking at each intermediate step, so that any inconsistencies could be easily detected.

An example of the desired process is shown in Figure 6.2., it is highly downsized from the actual dimensions used to facilitate visualization. Figure 6.1. can be observed for a description of the adopted symbology.

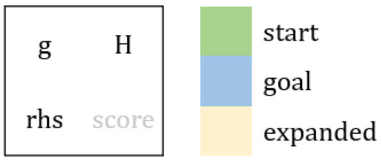
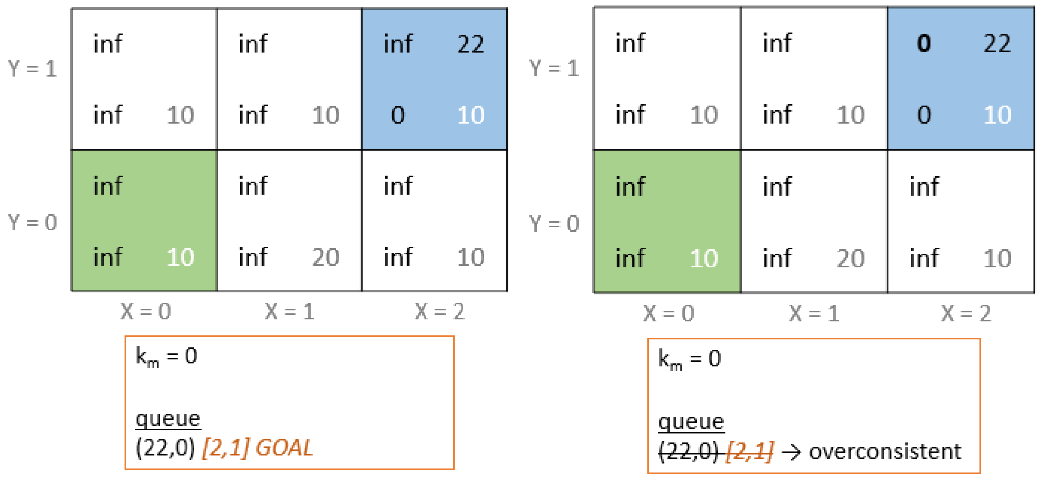


Figure 6.1. Legend for nodes.



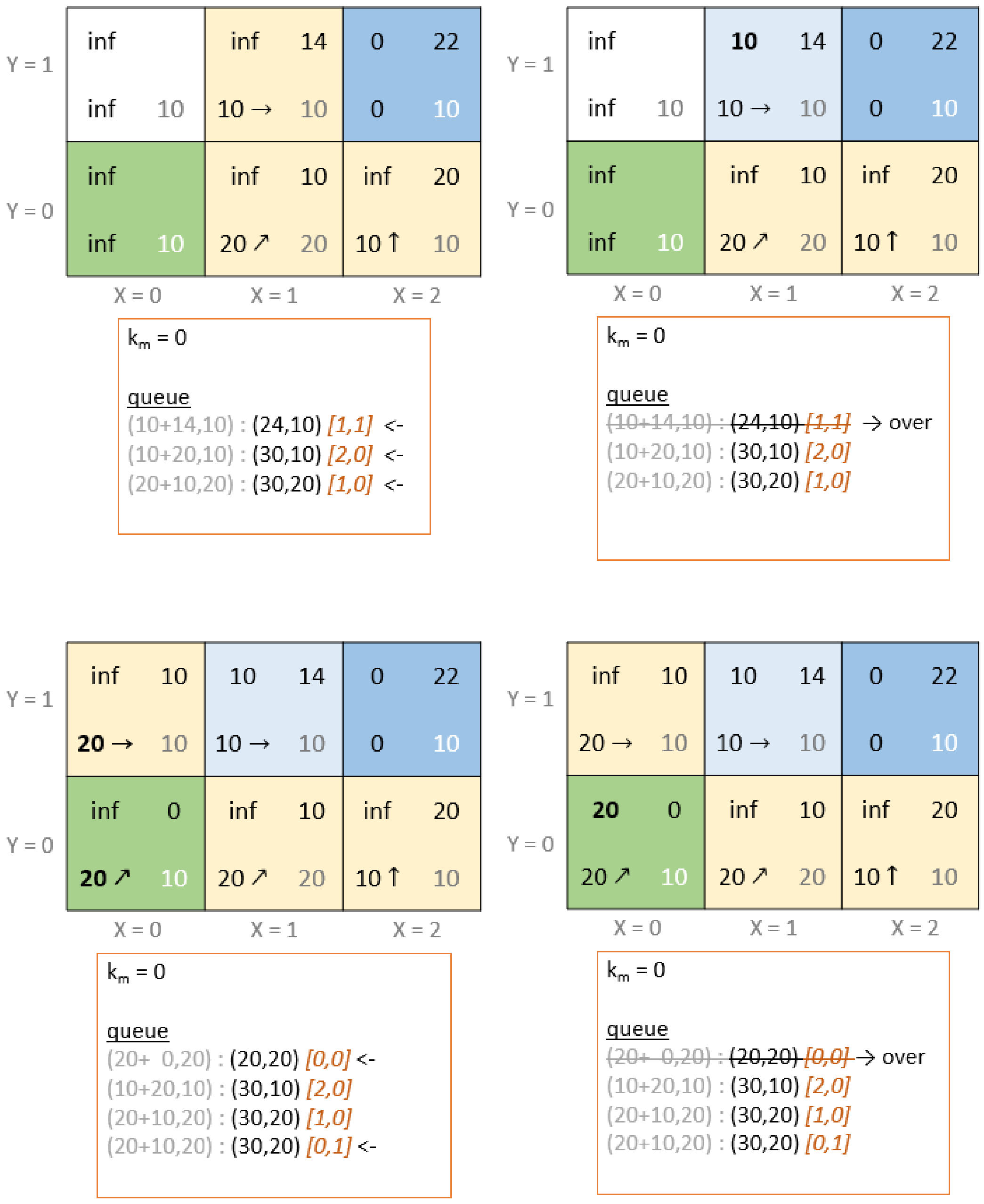


Figure 6.2. Execution progression for path finding.

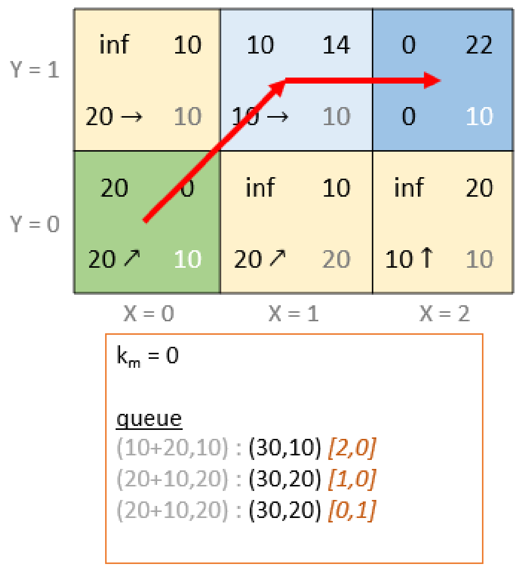


Figure 6.3. Resulting path for the first iteration.

Once the first run is finished and a route is identified, it may happen, for example, that a node changes its crossing score, as highlighted in Figure 6.4.

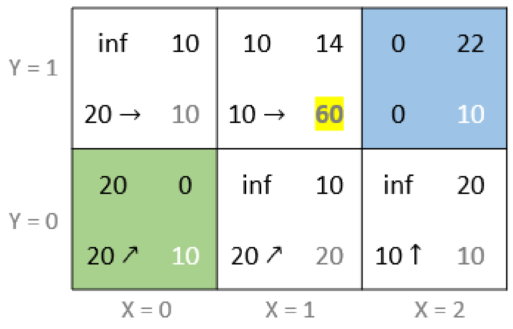


Figure 6.4. A new map reading is provided. Node (1,1) experienced a change in its crossing cost (highlighted in yellow).

With a similar unfolding, but starting this time with some already known data, the algorithm takes care of adjusting the route, if necessary, as is seen in Figure 6.5. Note that in this case the rover has not yet moved, and therefore the start node has not been changed.

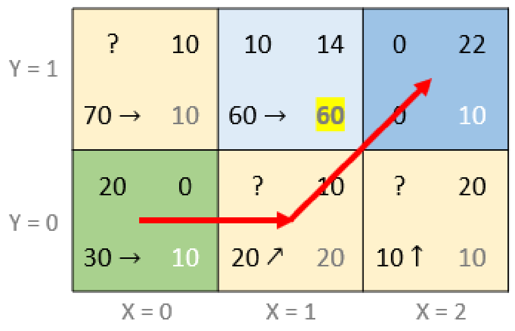


Figure 6.5. Result of the re-planning.

All the cases listed in the section 2.5. have been tested several times, with varying and also interconnected scenarios.

6.2 Randomly generated maps

Once the correct performance was established in the case of small, custom-built maps, the need to expand the tests was addressed. This was done to make sure that significantly larger maps were available to test performance with larger data loads. The other motivation was to introduce the random aspect into the tests, to try to include all possible configurations, complications and special cases that might not come easily to mind.

A Python script was used to provide the algorithm with a large number of different maps. Once some parameters have been set for the desired variance and size, as well as the type of data to be represented (altitude, slope, or solar exposure), the script takes care of generating images that simulate the scanning of a map. The images were saved in greyscale bitmap format. Each pixel of such images represents a map cell, where its value in grayscale is an evaluation respectively of its altitude or slope.

An example of the maps 1000x1000 thus generated is shown in Figure 6.7. and Figure 6.9., representing altitude and slope, respectively. A visualization, for representational purposes only, of the same maps in three-dimensional view is shown above the respective two-dimensional versions, in Figure 6.6. and Figure 6.8.

The coloring of the maps in these images is for illustrative purposes only; those used for testing are, as anticipated, in greyscale.

An example of such a map is shown in Figure 6.10. For better visibility of the computed path, smaller maps, 500x500 in size, will be represented from here on. This figure shows the slope of the area, where the darker areas represent a gentler slope, which is therefore more convenient to travel on.

Instead, the map for solar exposure is processed from the altitude map, as shown in Figure 6.11., assuming in this the light source positioned on the right of the image.

As a matter of fact, it can be seen in Figure 6.12. how the path generated for this map, highlighted by the green line, tends to move more in the darker areas, while still maintaining the tendency not to deviate from the shortest path (which would obviously be a straight line).

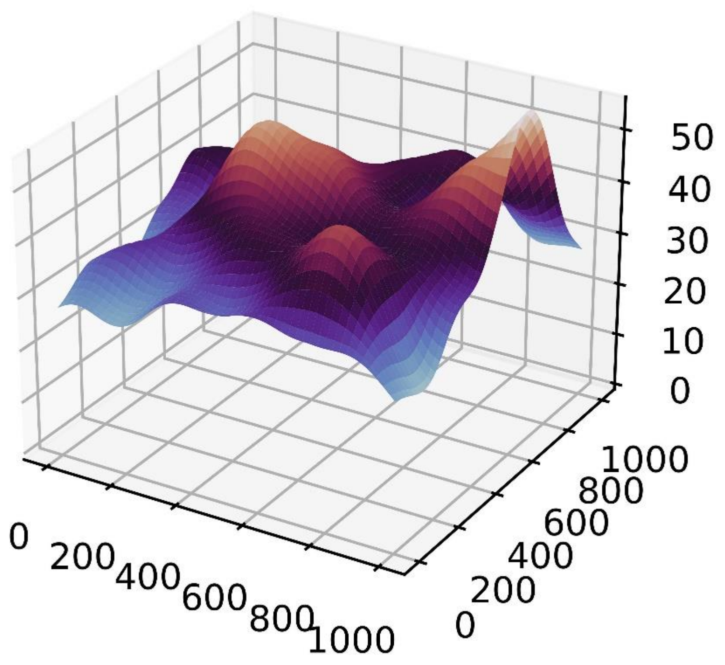


Figure 6.6. Three-dimensional view of the height map.

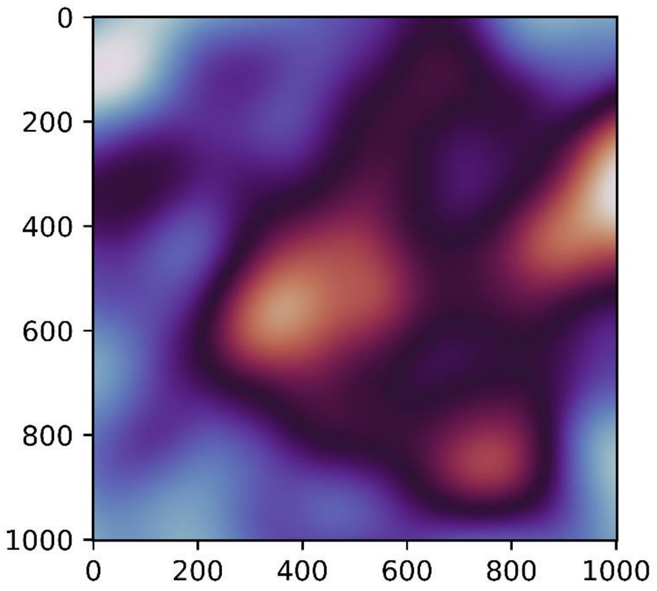


Figure 6.7. Bi-dimensional view of the height map.

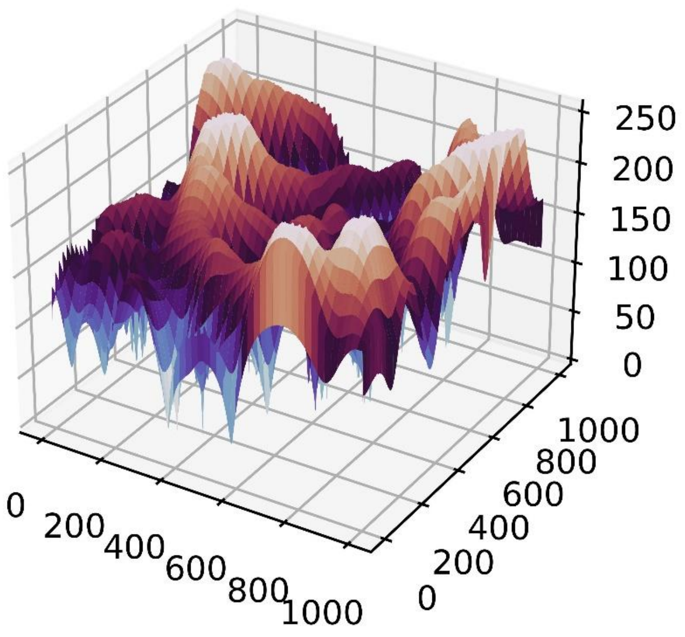


Figure 6.8. Three-dimensional view of the slope distribution.

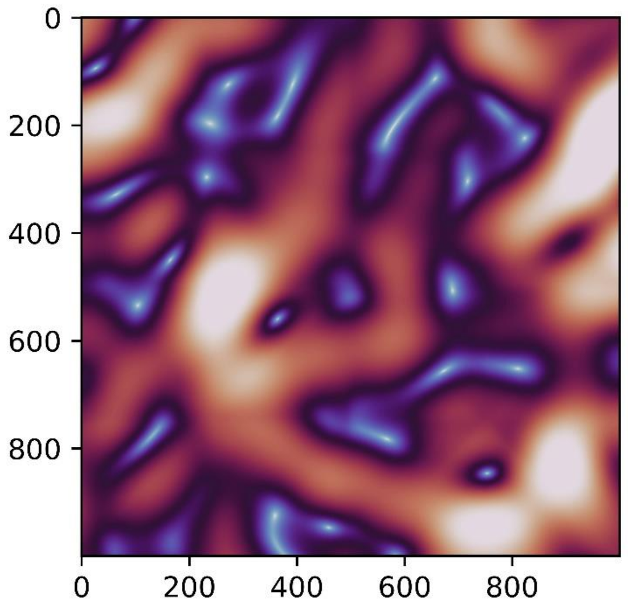


Figure 6.9. Bi-dimensional view of the slope distribution.

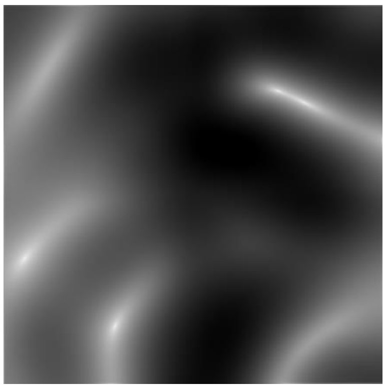


Figure 6.10. Example of a random generated height map in greyscale, 500x500 pixels.

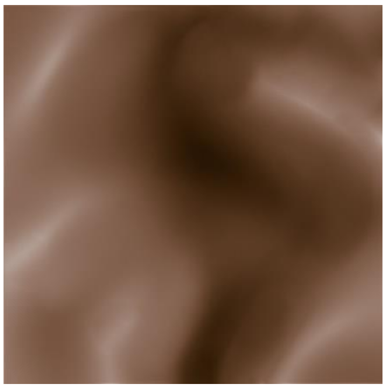


Figure 6.11. Example of solar exposure distribution obtained from Figure 6.10.

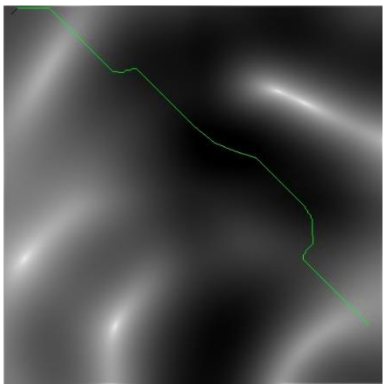


Figure 6.12. Solution path (green line) computed for the map in Figure 6.10.

These tests also made it possible to detect and correct some inefficiencies in the algorithm. In particular, identifying which points in the algorithm took the longest to process, and on which it was therefore important to place more attention so that they would be as optimized as possible. Similarly, some redundancies in variables were resolved to lighten the total impact on memory and avoid buildups.

These tests were carried out on more than three hundred maps of size 1000x1000 cells (or pixels).

6.3 Simulation of map integration from different sources

To simulate a more similar environmental awareness to the one obtained in real applications, some workarounds were adopted to achieve the corresponding mappings.

Once a random map was created, exactly as in the previous tests, a blurred copy was created, in which all details were thus considerably flattened.

During the reading phase, the algorithm could then rely its observations on the entire blurred map and enrich them with detailed data coming only from the area in the direct vicinity of the starting node. More precisely, in a square with semi-side equal to 50 cells/pixel, obviously centered in the start node.

Figure 6.13. shows the result of the first run of the planner using these conditions. Again, for greater visibility of the paths here are depicted smaller maps, with 500x500 dimensions. Note in the upper left corner a piece of the square with more prominent details. Again, the green line represents the path calculated by the planner at the time of the data acquisition.

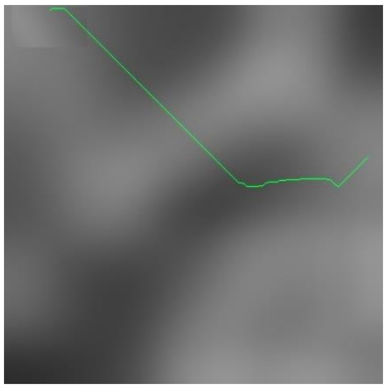


Figure 6.13. Initial path for mixed maps.

In the following images (Figure 6.14.), it is possible to appreciate the continuation of the algorithm as the rover moves forward, until the algorithm terminates due to the objective node being reached. As the vehicle moves to follow the computed path, the awareness of different areas of the map also changes, which leads the planner to eventually refine its assessments.

Again, the planner correctly succeeds in finding the path that best manages to compromise between the various objectives set.

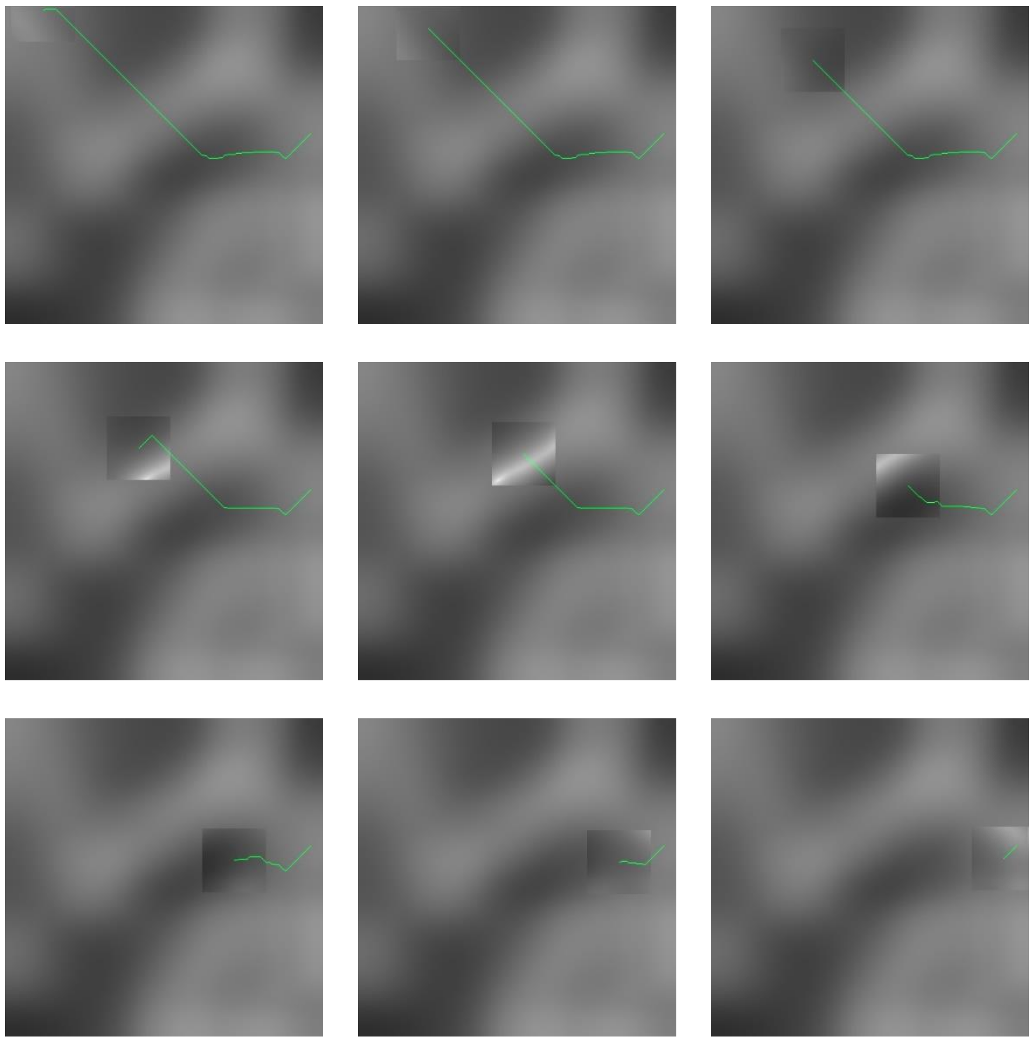


Figure 6.14. Evolution of the map and best path.

Besides the general correctness of the solutions, there is a need to evaluate the performance of the algorithm. For this purpose, detailed evaluations of memory usage and the time required are uncovered in the section immediately following.

An important parameter for evaluating such performance is however the size of the subset of nodes expanded during full execution.

A representation of this exact evaluation can be seen in Figure 6.15., where the nodes that have been examined and expanded are highlighted in yellow. It is important to keep in mind that whenever a node undergoes a change in one of the values of its cost vector, it is automatically placed in the expansion queue. This clearly causes a fairly large subset of expansion in the case under consideration, which involves updating costs by following the rover's movements. Despite this, it is appreciated how nodes that would stray too far from the optimal path are never considered, thus greatly limiting waste.

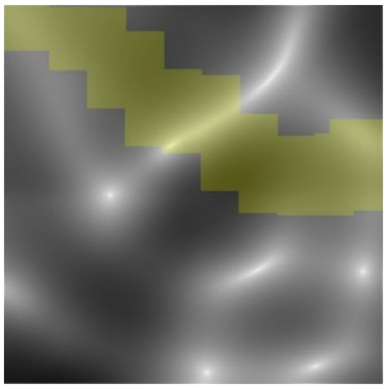


Figure 6.15. Highlighting of expanded nodes.

6.3.1 Analysis of execution times

In order to measure the performance of the algorithm, and improve it where necessary, several estimates were made concerning different areas.

First and foremost, the execution times required to accomplish both a complete path search and the times required for the most significant subroutines were measured. In particular, attention was paid to UpdateMap, which takes care in its first execution to create all the necessary nodes and edges, and in subsequent executions to update the graph where the costs might be found to have changed. Next attention is paid to the functions ComputeMOPaths and GenerateMOPaths, which are concerned respectively with assigning to all nodes that have the potential to be part of the path the correct values of g, and with using those values to backwardly construct a consistent path.

Table 6.1. shows the different periods of time, in seconds, required for these subroutines and for each individual cycle, where a complete cycle is considered from the time a map is started to be read until the output of a finished path.

Each of the columns represents a new algorithm call, that is, a reading of a new map where the rover turns out to have moved along the path.

execution	0	1	2	3	4	5	6	7	8	9
UpdateMap()	0,711 s	0,012 s	0,017 s	0,023 s	0,025 s	0,021 s	0,026 s	0,022 s	0,021 s	0,020 s
ComputeMOPaths()	4,031 s	0,047 s	0,069 s	0,071 s	0,182 s	0,179 s	0,080 s	0,100 s	0,167 s	0,053 s
GenerateMOPaths()	0,004 s	0,004 s	0,004 s	0,003 s	0,003 s	0,003 s	0,003 s	0,001 s	0,000 s	0,000 s
TOTAL	4,747 s	0,065 s	0,092 s	0,103 s	0,215 s	0,208 s	0,114 s	0,129 s	0,193 s	0,077 s

Table 6.1. Example of execution times for a complete path traversal.

Looking at the first row, it can be easily seen that the first translation of the map is the most time-consuming, as it involves creating from scratch the entire graph representing all the necessary nodes and edges. From the second execution onward, in which only a few nodes undergo changes, the execution time is greatly reduced and almost constant.

The same observation can be made for ComputeMOPaths, which at first execution must assign the g values to all nodes that compete to be part of the path to the goal node. In the first execution, this function represents by far the largest expense in terms of time. From the second execution onward, its contribution is limited to re-evaluating some of the nodes and thus the time is shortened. In addition, it can be seen that as the execution progresses, the time still gradually decreases, which is expected since the path yet to be re-evaluated gets progressively shorter, which is understandable.

Also for GenerateMOPaths, despite requiring much less time, a decreasing trend can be identified, caused by the shortening of the required path.

The total time reveals how the first round takes a not insignificant amount of time, while from the second onward the timeframes are almost always well below the $0{,}2$ second mark. This complies with the desired behavior, allowing the rover to perform most of the work before starting the navigation, and instead be able to very quickly perform replanning on the fly, without the need to stand still during the process.

These same measurements were repeated many times, using different maps and situations each time, but maintaining the same distance between the start node and the goal node so as to be comparable. From these data, the averages on execution times shown in Table 6.2. were derived.

	average time (including first ex.)	average time (except first ex.)
UpdateMap()	0,089 s	0,022 s
ComputeMOPaths()	0,492 s	0,114 s
GenerateMOPaths()	0,003 s	0,003 s
TOTAL	0,588 s	0,143 s

Table 6.2. Average execution times for the principal subroutines.

These times are, however, obviously only a very approximative indication, since they are measured while running on a standard personal computer, and not on space grade hardware. It is expected that in real cases all times will be greatly increased. But this does not prevent important assessments from being made.

Indeed, it is clear that to further reduce the time impact of the algorithm the first point to be further optimized is the GenerateMOPaths procedure.

6.3.2 Memory occupation

In space applications, the hardware used has more limited capabilities than what is used on Earth, so the amount of memory used to perform each task is an important characteristic.

Using the tools provided by Visual Studio, tests were conducted on the amount of memory required to run this algorithm.

The data given below in Figure 6.16. refer specifically to the last of the tests carried out with differentiated focus maps, this case being the most wasteful. This is of course after implementing various techniques to improve memory waste where possible.

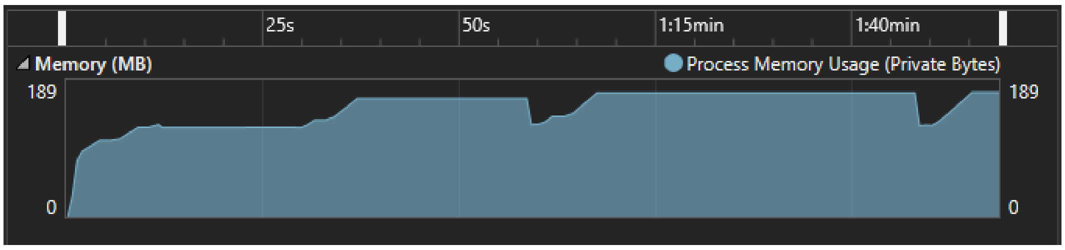


Figure 6.16. Memory occupation.

In the figure it is possible to appreciate the trend of memory occupied during a little more than two cycles of execution.

The descending peaks are due to the deletion of some lists and queues as the path is defined and provided as output. The immediately subsequent rise is due to the renewal of such data and the reading of the new maps for a new execution.

It is clear how this aspect of the algorithm can be further optimized by choosing the most appropriate types of variables and trying to identify further data that are retained longer than necessary.

6.3.3 CPU usage

For the same reasons just described, it is important to keep track of the power usage, since the rover has a finite amount available for all its activities.

Again thanks to the Visual Studio tools, data was collected on CPU utilization during execution, depicted in Figure 6.17. Clearly this is very dependent on the device on which the tests are run, thus of less real value, but it again served to try to optimize the algorithm as much as possible.

In fact, in addition to the total utilization, a close look was taken at the computational effort required by individual functions and sub-functions, so as to identify and improve from time to time those that were most influential.

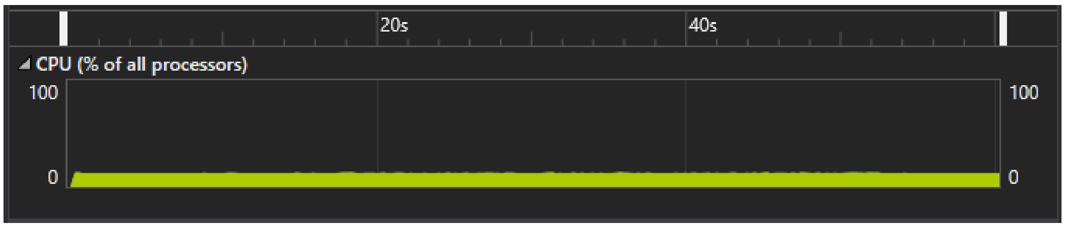


Figure 6.17. Percentage of the CPU utilization.

In this case, apart from the quick initial rise, utilization is maintained roughly constant throughout the entire execution.

Again, it is undoubtedly possible to further optimize the utilization of the resources that are available.

6.4 Implementation in ROS2

After confirming the proper functioning of the planner in all previous tests and optimizing the algorithm as much as possible with regard to required power, occupied memory, and execution speed, it was time to integrate it into an environment that simulates interaction with the other components of a complete autonomous navigation system.

To this end, for reasons elaborated earlier in section 5.1., ROS2 is used (ROS2 Foxy, s.d.), specifically by embedding within the Nav2 environment (Nav2 - Navigation 2 1.0.0 documentation, s.d.).

The Robot Operating System is a collection of open-source software drivers, libraries and state-of-the-art tools for building robot applications of any kind. Nav2 is one of the aforementioned tools, designed specifically for the development of all applications involving the mobility, more or less autonomous, of a robot.

ROS processes can be described as nodes in a graph structure (unrelated to the graphs we have defined for search algorithms). Such processes are connected by edges, which in this case take the shape of topics. Through topics the nodes, or processes, can transmit messages to each other containing data of various kinds, call the functionality of other nodes, and provide themselves services.

A plugin was created specifically to interface with the already modular structure of Nav2. In this way, the new module can use all the functionality already present, such as the costmap layer, controller, and behavior tree.

With the help of Gazebo and ROS packages that handled its functionalities, it was possible to complete the simulations. Gazebo (Gazebo, s.d.) is a simulator for indoor and outdoor environments, robots and robot populations, closely interconnected with ROS2 functionality.

Once the initial position of the rover, initially designated by Gazebo, was found, it was manually communicated to the planner via the user interface. After that, the desired target position could easily be marked on the map. At this point the planner receives information from the various modules about the costmap, the start position, and goal position. Then, it takes care exactly as in the previous cases to calculate an appropriate path. By sending that point-by-point information back to the controller, the rover can move to achieve the desired behavior.

The Figure 6.18. represents a screen capture during a simulation. The rover is the Turtlebot3, hardly visible in the image but present just below and to the right of center of the map, covered by numerous reference systems.

The path computed at the time of capture can be seen represented purple, and it would slightly adjust as the rover gained new data. Indeed, the square area where the rover directly, and therefore more accurately, perceives its surroundings is highlighted in more saturated colors.

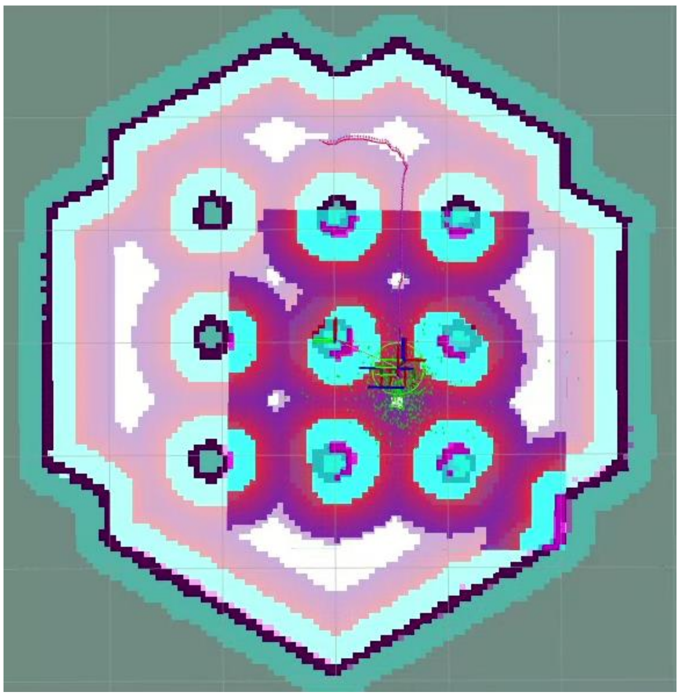


Figure 6.18. Screen capture of the planner operation in a ROS simulation.

Again, the planner was able to calculate and convey the shortest and safest route within the provided map. Unfortunately, it was not possible to test its full potential by adding sun exposure data among the input information. In fact, it is necessary to slightly modify some characteristics of the structure of the Nav2 data flow in order to allow the simultaneous reception of two maps that are completely unrelated to each other. This definitely represents an aspect to be investigated in future.

7. Conclusions and future developments

During the development of this thesis, many of the aspects involved in the successful operation of a path planner were addressed and analysed. After analysing many algorithms, the choice fell on MOD* Lite.

This algorithm was then implemented into a complete, working code. Test phases followed by optimization phases were then performed alternately. In this way it was possible to ensure that the planner was as optimized as possible and that it was able to generate consistent and acceptable paths under potentially any condition. It has also been equipped with features and parameters that make it easily adaptable to different types of tests and allow refined tuning for future tests.

All tests, either purpose-built or in a simulation environment, reported the expected results, assuring once again that the planner is robust and reliable. The planner thus composed was also successfully integrated into the ROS2 environment.

Future work to take this project forward should first involve further low-level optimization of the code that translates the entire algorithm. Optimization should focus both on execution time and, more importantly, on the amount of memory required for the process. These two tasks would obviously be mutually beneficial.

In second place, it would be worth expanding the tests carried out with the help of ROS2, so as to provide for better integration of more than one costmap at the same time, to give proper attention to the multiobjective feature of MOD* Lite. Indeed, at the present time, in these simulations it is able to optimize for the shortest and safest path, but not to receive data on sun exposure. This therefore prevents it from optimizing for this additional objective.

Absolutely no less important, would be the use of the three-dimensional scanning of the entire environment present in RoXY, to replace the generic maps used so far. This would make it possible to test how well the planner works on a terrain that is much more realistic than either the random maps or the maps provided by Gazebo. Since it is also a terrain that has already been extensively tested, it would be interesting to find out how well it performs in this real scenario compared to other planners already in use.

Ultimately, the natural progression of this work would be the integration of this planner into the AutoNav system developed for the Seekur rover, currently used for the test in RoXY.

This would allow, among other things, the use of real data on solar exposure given under different weather conditions. It would remove an additional layer of uncertainty due to human error in artificially creating these data. Some testing under these conditions would also allow for better calibration of the best features for the input maps and for some of the algorithm's internal variables. The most obvious among these is the heuristic function, which could be questioned again if the results required it.

The future of the SINAV project more generally will certainly involve the continued search for innovative solutions in all phases that constitute autonomous navigation, from the selection of the hardware to the software implementations.

List of Figures

List of Tables

Bibliography

(n.d.). Retrieved from Nav2 - Navigation 2 1.0.0 documentation: https://navigation.ros.org/

Benchmark of major hash maps implementations. (n.d.). Retrieved from https://tessil.github.io/2016/08/29/benchmark-hopscotch-map.html

Bombini, L., Coati, A., Medina, J., Molinari, D., & Signifredi, A. (2015). A general purpose approach for global and local path planning combination.

Dakulovic, M., & Petrović, I. (2011). Two-way D* algorithm for path planning and replanning. *Robotics Auton. Syst., 59*, 329-342.

Elfes, A. (1989). *Occupancy Grids: A Probabilistic Framework for Robot Perception and Navigation.* University, Pittsburgh, PA, USA, Carnegie Mellon.

Gasparetto, A., Boscariol, P., Lanzutti, A., & Vidoni, R. (2015, 03). Path Planning and Trajectory Planning Algorithms: A General Overview. *Mechanisms and Machine Science, 29*, 3-27. doi:10.1007/978-3-319-14705-5_1

Gazebo. (n.d.). Retrieved from https://classic.gazebosim.org

Geiger, A., Ziegler, J., & Stiller, C. (2011). StereoScan: Dense 3d reconstruction in real-time. *IEEE Intelligent Vehicles Symposium, IV*, 963-968.

Goldberg, S., Maimone, M., & Matthies, L. (2002). Stereo vision and rover navigation software for planetary exploration. *IEEE Aerospace Conference, 5*(5-2025-5-2036).

Guan, X., Wang, X., Fang, J., & Feng, S. (2014, 11). An innovative high accuracy autonomous navigation method for the Mars rovers. *Acta Astronautica, 104*, 266–275. doi:10.1016/j.actaastro.2014.08.001

Gunantara, N. (2018). A review of multi-objective optimization: Methods and its applications. *Cogent Engineering, 5*(1). doi:10.1080/23311916.2018.1502242

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968, July). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics, vol. 4, 4*(2), 100-107. doi:10.1109/TSSC.1968.300136

Helmick, D., Angelova, A., Livianu, M., & Matthies, L. (2007). Terrain Adaptive Navigation for Mars Rovers. *IEEE Aerospace Conference*, 1-11.

Jain, A., Balaram, J., Cameron, J., Guineau, J., Lim, C., Pomerantz, M., & Sohl, G. (2004). Recent developments in the ROAMS planetary rover simulation environment. *IEEE Aerospace Conference Proceedings, 2*, 861-876.

Jinming, Z., Xun, W., Lianrui, X., & Xin, Z. (2022). An Occupancy Information Grid Model for Path Planning of Intelligent Robots. *ISPRS International Journal of Geo-Information, 11*(4), 231. Retrieved from https://doi.org/10.3390/ijgi11040231

Knuth, D. E. (1977). A generalization of Dijkstra's algorithm. *Information Processing Letters, 6*(1), 1-5. doi:https://doi.org/10.1016/0020-0190(77)90002-3

Koenig, S., & Likhachev, M. (2002). D* lite. In A. P. Press (Ed.), *Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence* (pp. 476-83). Edmonton, Alberta, Canada: Rina Dechter, Michael J. Kearns and Richard S. Sutton. Retrieved from http://www.aaai.org/Library/AAAI/2002/aaai02-072.php

Koenig, S., Likhachev, M., & Furcy, D. (2004). Lifelong Planning A. *Artificial Intelligence 155.1* , 93-146.

Marcus, D. A., & America, M. A. (2008). Graph Theory: A Problem Oriented Approach. *The Mathematical Association of America Press.*

Mars Exploration Rovers. (n.d.). Retrieved from https://mars.nasa.gov/mer/mission/rover/eyes-and-senses/

Milella, A., Reina, G., & Nielsen, M. (2021, 04). A multi-sensor robotic platform for ground mapping and estimation beyond the visible spectrum.

Nidhal, K. T.-O. (2021, 8). Sea Lion Optimization Algorithm for Solving the Maximum Flow Problem. *International Journal of Computer Science and Network Security (IJCSNS), 20*(8), 30-68.

Nieto, J., Monteiro, S., & Viejo, D. (2010). 3D geological modelling using laser and hyperspectral data. *IEEE International Geoscience and Remote Sensing Symposium (IGARSS).*, 1-7.

Nosrati, M., Karimi, R., & Hasanvand, H. A. (2012, November 23). Investigation of the(star) search algorithms: Characteristics, methods and approaches. *World Applied Programming - TI Journals, 2*(4), 251-256.

Oluwaseun, O. M., Adefemi, A. A., Olatayo, M. O., & Bukola, O. B. (2022). An Improved multi-objective a-star algorithm for path planning in a large workspace: Design, Implementation, and Evaluation. *Scientific African, 15*, e01068. doi:https://doi.org/10.1016/j.sciaf.2021.e01068

Oral, T., & Faruk, P. (2016). MOD* Lite: An Incremental Path Planning Algorithm Taking Care of Multiple Objectives. *IEEE Transactions on Cybernetics, 46*(1), 245-257. doi:10.1109/TCYB.2015.2399616

Oral, T., & Polat, F. (2012). A Multi-objective Incremental Path Planning Algorithm for Mobile Agents. In *2012 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology* (Vol. 2, pp. 401-408). IEEE Computer Society.

Petkov, P., Yang, L., Qi, J., Song, D., Xiao, J., Han, J., & Xia, Y. (2016, 07 04). Survey of Robot 3D Path Planning Algorithms. *Journal of Control Science and Engineering*. doi:10.1155/2016/7426913

Ramalingam, G., & Reps, T. (1992, May). An Incremental Algorithm for a Generalization of the Shortest-Path Problem. *Journal of Algorithms - University of Wisconsin Technical Report, 21*(2), 267-305.

Rekleitis, I. a.-L., Gemme, S., Lamarche, T., & Dupuis, E. (2007, 06). Terrain Modelling for Planetary Exploration. 243-249. doi:10.1109/CRV.2007.63

Rekleitis, I., Bedwani, J.-L., & Dupuis, E. (2009, 06). Autonomous planetary exploration using LIDAR data. 3025-3030. doi:10.1109/ROBOT.2009.5152504

ROS2 Foxy. (n.d.). Retrieved from https://docs.ros.org/en/foxy/index.html

Stentz, A. (1995). The focussed d^* algorithm for real-time replanning. *IJCAI*, *95*, pp. 1652-1659.

Stentz, A. (1997). Optimal and Efficient Path Planning for Partially Known Environments. In *Intelligent Unmanned Ground Vehicles: Autonomous Navigation Research at Carnegie Mellon* (pp. 203-220). Boston, MA: Hebert Martial H., Thorpe Charles, Stentz Anthony.

Stewart, B., & White, I. C. (1991). Multiobjective A. *Journal of the ACM, 38*(4), 775-814.

Wang, H., Lou, S., Jing, J., Wang, Y., Liu, W., & Liu, T. (2022, 02). The EBS-A* algorithm: An improved A* algorithm for path planning. *PLOS ONE, 17*(2), 1-27. doi:10.1371/journal.pone.0263841

Washington, R., Golden, K., Bresina, J., Smith, D., & Anderson, C. (1999). Autonomous rovers for Mars exploration. *IEEE Aerospace Conference*, *1*, pp. 237-251. Aspen, CO, USA.

Woods, M., Shaw, A., Tidey, E., Pham, B., Lacroix, S., Mukherji, R., . . . Chong, G. (2014, 10). Seeker – Autonomous Long Range Rover Navigation for Remote Exploration. *Journal of Field Robotics, 31*, 940. doi:10.1002/rob.21528

Xiao, J., Liu, M., & Zhu, Z. (2021). Low walk error multi-stage cascade comparator for TOF LiDAR application. *Microelectronics Journal*, 116. doi:10.1016/j.mejo.2021.105194

Xue, Y., & Sun, J.-Q. (2018). Solving the Path Planning Problem in Mobile Robotics with the Multi-Objective Evolutionary Algorithm. *Applied Sciences, 8*(9). doi:10.3390/app8091425

Zitzler, E., Laumanns, M., & Thiele, L. (2001). SPEA2: Improving the Strength Pareto Evolutionary Algorithm. *TIK-report, 103*.