Chapter

The Relationship between "C-Space", "Heuristic Methods", and "Sampling Based Planner"

Emanuele Sansebastiano and Angel P. del Pobil

Abstract

Defining the collision-free C-space is crucial in robotics to find whether a robot can successfully perform a motion. However, the complexity of defining this space increases according to the robot's degree of freedom and the number of obstacles. Heuristics techniques, such as Monte Carlo's simulation, help developers address this problem and speed up the whole process. Many well-known motion planning algorithms, such as RRT, base their popularity on their ability to find sufficiently good representations of the collision-free C-space very quickly by exploiting heuristics methods, but this mathematical relationship is not highlighted in most textbooks and publications. Each book focuses the attention of the reader on C-space at the beginning, but this concept is left behind page after page. Moreover, even though heuristics methods are widely used to boost algorithms, they are never formalized as part of the Optimization techniques subject. The major goal of this chapter is to highlight the mathematical and intuitive relationship between C-space, heuristic methods, and sampling based planner.

Keywords: motion planning, C-space, optimization techniques, heuristic methods, sampling based planner, educational dissemination

1. Introduction

Starting from the concept of configuration space (known as c-map as well), this chapter highlights the necessity of finding an alternative way to perform path search than computing the whole configuration space deterministically. Even simple 2D scenarios appears to be quite difficult to be handled by calculating the whole configuration space.

This chapter comes from the necessity of recalling the fact that such powerful and well-known algorithms like sampling based algorithms are strictly connected to the configuration space. Configuration space is the most ancient concept root of motion planning. Moreover, motion planning algorithms are generally imagined by the people as deterministic analysis of the environment because humans expect robots to be foolproof. Sampling based planners, instead, are coming from heuristic approaches which are the exact opposite of deterministic analysis. Most of the disseminating books and papers [1–6] are focused on explaining the algorithms rather then report how and why those algorithms were born. The main contribution of this chapter is reporting how and why sampling based planners

algorithms were born and which challenges older planning algorithms could not solve. Knowing the history and the ideas behind each planning algorithm allows scientists to fully understand its capabilities. Moreover, it allows scientists to know which algorithm is more eligible for a specific problem a priori without testing all of them.

Section 2 explains how to compute a full configuration space and reports several examples in order to fully address the practical meaning of c-map. Moreover, this section describes accurately which are the major drawbacks of computing the whole c-map. In many cases, computing the c-map is practically impossible.

Section 3 describes heuristic methods and their ability to tackle very convolute or large problems quickly without losing much information. Those methods do not ensure any result because they are not deterministic, but they do find a very good solution if they are well designed. This chapter is not reporting any specific heuristic method because they must be designed according to the problem.

Section 4 gives a quick overview of sampling based planners. Sampling based planes comes from the necessity of exploring configuration spaces by using heuristic methods. In particular, this section reports two of the most famous sampling based planes: Probabilistic road map (PRM) and Rapidly-exploring random tree (RRT). Those planners or a variation of them are probably the most use all around the world.

2. Configuration space (C-space)

The parameters which define the configuration of a system are called generalized coordinates, and the vector space defined by these coordinates is called configuration space of the system. The configuration space represents all possible states that the system might have. According to the number of degrees of freedom of a system, its configuration space has to be defined by a vector having the same length. Due to this fact, the representation of configuration spaces on paper is limited to systems having 2 degrees of freedom.

Configuration spaces are often used in robotics to represents all possible mechanical configurations which a robot can have. Moreover, configuration spaces are used to motion planning by stacking several configurations into one motion. Given a starting configuration and a goal configuration there are infinite possible motions which lead the robot from the stating point to the goal one. Assuming that the path search is restricted to geometrical constraints (no kinematics and no dynamics is involved), the cleverest path would be defined by a straight line connecting the starting point and the goal point on the configuration space (**Figure 1**). However, path planning algorithms are mostly used to carry robot through obstacles. Assuming that obstacles are not moving or that they are moving slowly enough to be considered static comparted to the robot motion speed. Configuration space can be computed to describe all possible robot configurations not colliding with any obstacle. These configuration spaces are called "*collision free c-space*".

Creating a configuration space is simple if there are no obstacles in the scene. Knowing the range of values that each degree of freedom can have is enough to build a fully function configuration space map. Including obstacles into the scene increases significantly the complexity of building configuration space map.

Before continuing this chapter, it is better to highlight that in order to simplify the handling of the topic and help the reader to visualize all examples, all presented robots and obstacles lay into a 2D plane. Moreover, all robots are going to have just 2 degrees of freedom or less to allow 2D representation of configuration spaces.

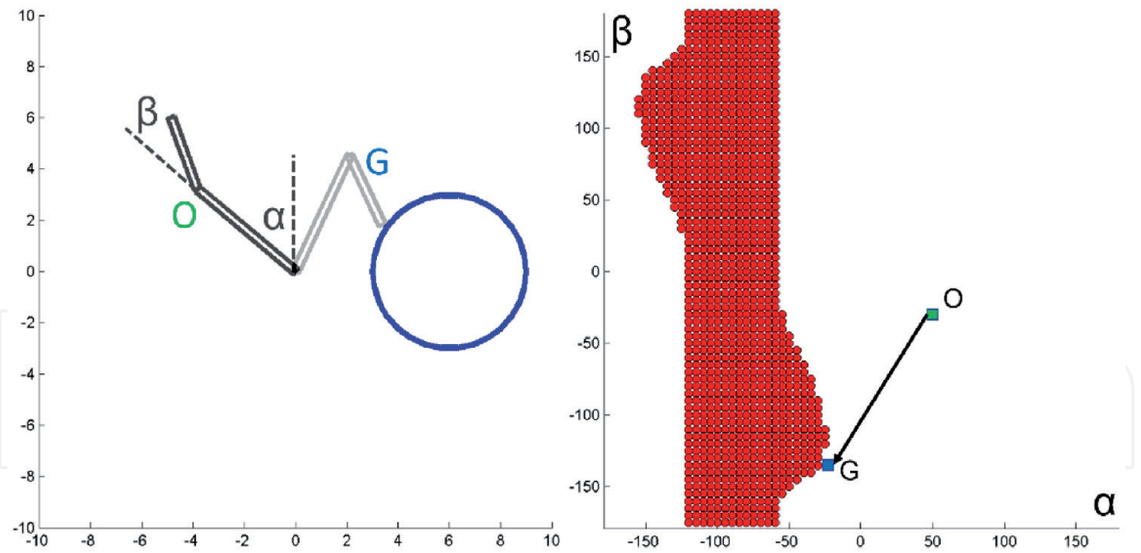

Figure 1.
C-map of a planar robot defined by 2 links and 2 rotational joints.

2.1 Obstacle definition

Objects have many possible shapes in real environment, but most of them can be accurately represented by the sum of a finite number of convex shapes. For sake of simplicity, only 2D objects are taken into account in this chapter, but similar statements can be derived for 3D objects. Since 3D objects have one more dimension, the computational costs of the algorithms should generally increase by one order of magnitude. In some cases, this computational cost variation makes algorithms feasible for 2D object scenarios and not feasible for 3D object scenarios.

Robots have basically two ways to include obstacles into their path planification:

- Detecting obstacles by themselves.
- Having another system which constantly provides obstacles to them.

Object detection is a very broad field and many sections of it are still opened. As E. Arnold et Al reports in their paper [7], there are a lot of different techniques to detecting objects according to sensor types, sensor configurations and the operative scenarios. Due to this reason, this chapter is not going to deal with object detection. All the objects in the scene are assumed to be known and provided by another system to the path planner.

As it was mentioned before, this chapter deals with 2D objects to ensure 2D visualization. 2D objects can be convex or non-convex. As it is shown in **Figure 2**, convex objects do not have any internal angle larger then 180°. Non-convex objects, on the other hand, have at least one of them. Non-polygonal objects are convex by default if they are regular shapes (circles, ellipses, etc....). If objects are defined by a concatenation of various curves, a possible solution to establish whether the object belongs to convex or non-convex shapes is rearranging the edges of the object by a set of liner segments. According to the resolution of this edge substitution, the computational cost of convex categorization changes. Higher resolution leads to high precision, but high computational cost.

This convex categorization is fundamental because computing convex object collision is very easy and not computationally expensive due to computational

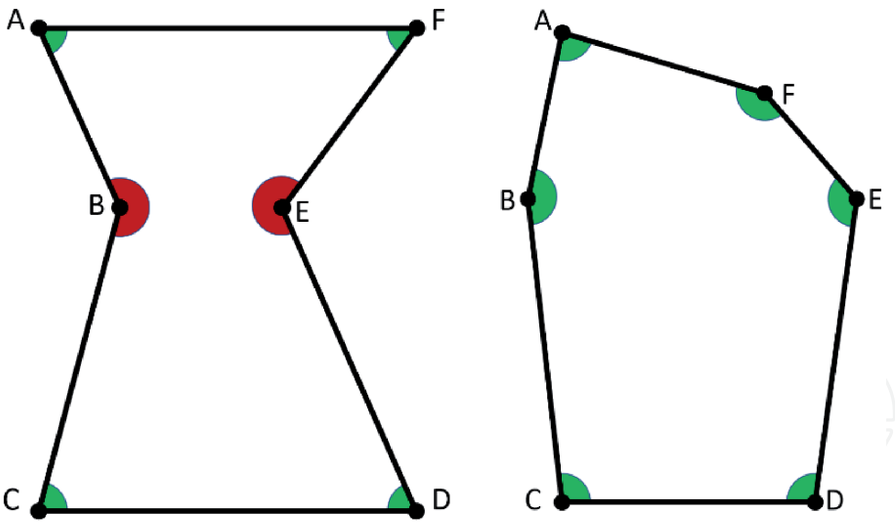

Figure 2.
Non-convex and convex 2D objects.

geometry algorithms [8]. So, it is better to "convert" non-convex shapes to convex shapes. The most used techniques to perform this transformation are:

- Convex hull algorithms (e.g., Graham's scan)
- Subdivision algorithm
- Triangle meshing algorithm

The *Convex hull algorithms* are able to find the smallest convex hull which includes all points of a dataset. As it is described in [8], there are many algorithms which are able to tackle this problem. The *Graham's scan* is probably one of the most famous. Since the corners of a shape can be seen as the points of a dataset (hull), the Graham's scan would be able to wrap all of them into a unique hull erasing all critical corners. **Figure 3** shows that the non-convex shape (ABCDEF) would be converted into a convex one (ACDF) by erasing corners B and E.

The *Convex hull algorithms* are techniques quicker than *Subdivision* if the number of critical corners is quite large, but they significantly reduce the accuracy of the object definition. Practically, those techniques cut out all critical corners and enlarge the area occupied by the original shape. The Graham's scan works perfectly on obstacles defined by a point cloud which, generally, derives from raw data. Since this chapter deals with known objects, there is no reason to take this technique into account.

The *Subdivision algorithm* is a technique which aims to redefine the original non-convex shape into the sum of convex sub-shapes. It does not compromise the original occupied area, because the algorithm tries split critical corners by dividing the original shape into sub-shapes. At each iteration (subdivision), the algorithm has to check whether the new shapes are still non-convex. If yes, the algorithm has to continue the operations. The main drawback of this algorithm is that the final number of sub-shapes is never known a priori. The only known thing is that the larger number of sub-shapes is p-2, where p is the number of corners. In the worst-case scenario, the algorithm has to subdivide the original shape into triangles which are always convex.

The *Subdivision algorithm* is theoretically the best technique because it splits just the critical corners until all shapes are convex, At the end of the process,

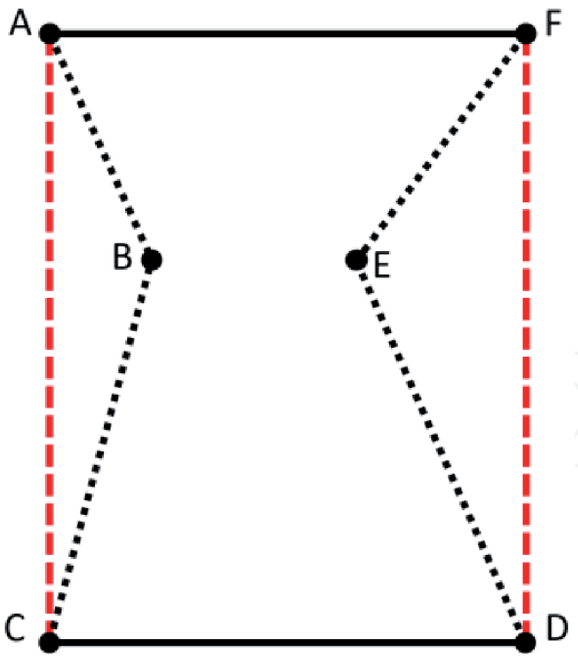

Figure 3.
Non-convex shape converted into a convex one by convex hull algorithms.

the number of sub-shapes is the smallest to ensure the overall convexity, but the algorithm has to check at every iteration which sub-shapes are still non-convex. The process might cost quite a lot according to the original object shape. **Figure 4** reports an example of shape *Subdivision algorithm*.

The *Triangle meshing algorithm* might be seen as special case of the *Subdivision algorithm*. Practically, it subdivides the original shapes until all sub-shapes are triangles like the worst-case scenario of the *Subdivision* technique, but it does not check sub-shapes convexity. Triangles are convex by definition.

The *Tringle meshing algorithm* is quick in term of subdivision if the number of critical corners is quite large because all sub-shapes created by the algorithm are

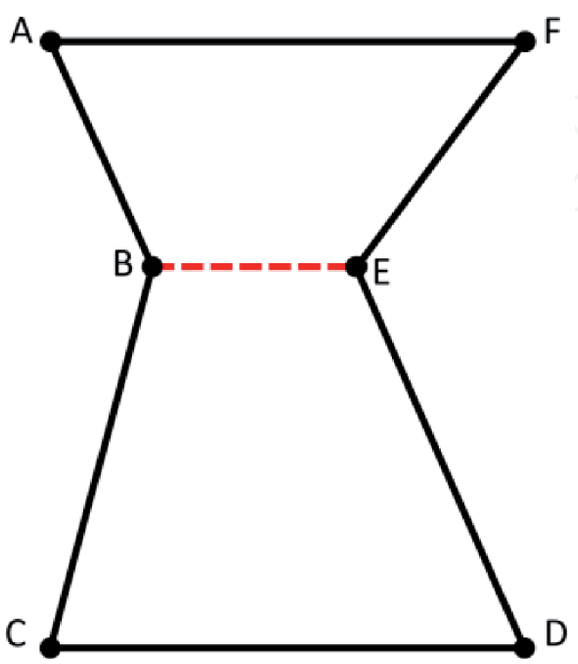

Figure 4.
Non-convex shape converted into a convex one by subdivision algorithm.

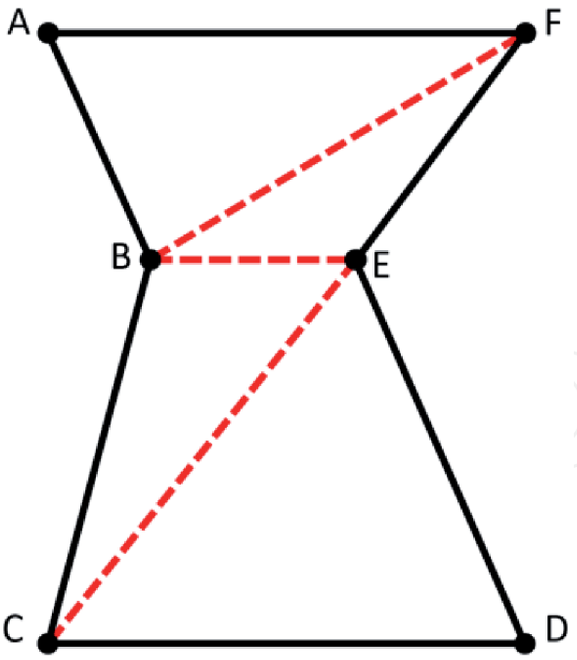

Figure 5.
Non-convex shape converted into a convex one by triangle meshing algorithm.

triangles. Since triangles are convex by default, there no reason to check whether sub-shapes are convex or not. The major advantage is that the number of iterations is related to the number of corners of the original object. In particular the number of sub-shapes is *p-2*, where *p* is the number of corners. **Figure 5** reports an example of this algorithm.

The major drawback of the *Tringle meshing algorithm* is related to the large number of final sub-shapes compared to the *Subdivision* technique. This drawback definitely affects object collision process, because having more objects means performing more collision checks during the c-map creation.

Since this chapter deals with known non-deformable objects, using the *Tringle meshing* technique is quite inefficient. The path planner converts non-convex shapes into convex just one time at the beginning of the whole process. Afterward, it will extensively check object collisions. Having the lowest amount of objects in the scene is crucial to reduce computational cost. To conclude, The *Subdivision algorithm* appears to be the most efficient.

2.2 C-space computation

According to the number of degrees of freedom of the robot, the type of degrees of freedom of the robot, the number of objects in the scene, and the shape type of those objects (circle, polygonal, etc....), computing the "*collision free c-space*" changes significantly. Moreover, the computational cost will increase according to the number of objects in the scene and the number of degrees of freedom. In theory, there is an equation which describes the c-space, but defining it is often very complex.

Let us consider a trivial example: a robot defined by a single link and a single rotational joint. The rotational joint is located to one of its ends and the link can only rotate around one of it end; no translation is involved. Moreover, there just one object in the scene defined by a circle (**Figure 6**).

Computing the c-map of this trivial example is immediate: there is just one degree of freedom represented by the angle α of the rotational joint. Since the

DOI: http://dx.doi.org/10.5772/intechopen.99826

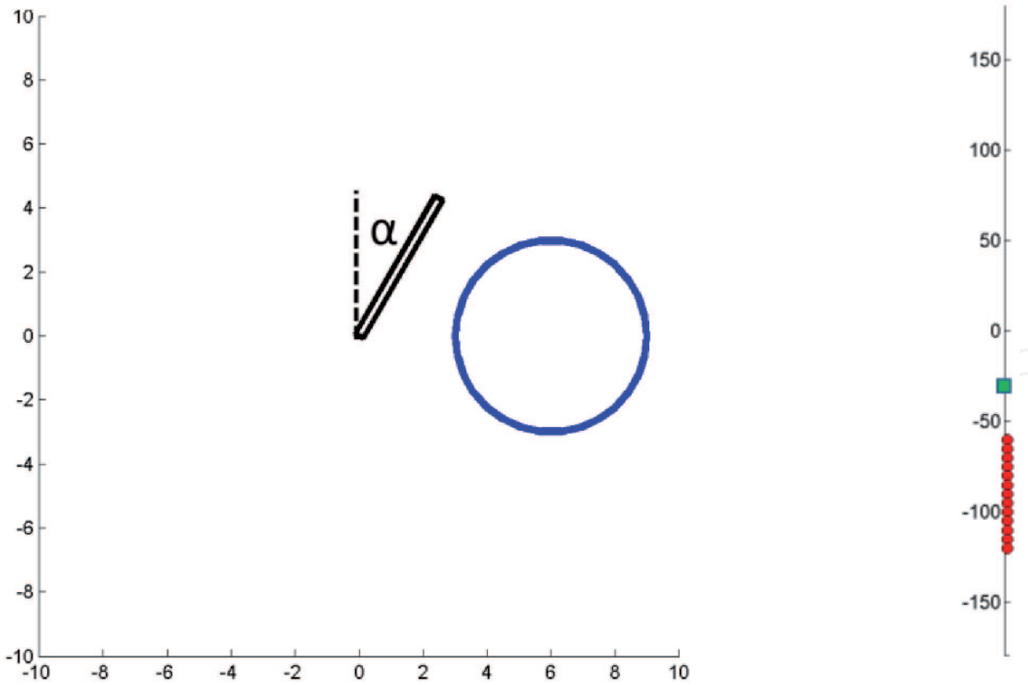

Figure 6.
C-map of a planar robot defined by 1 link and 1 rotational joint.

obstacle is just one, it is sufficient to calculate for which values the link touches the obstacle without crossing its edges to know which is the valid angle range. The c-map of **Figure 6** shows that the *collision free c-map* is basically a bar going from −180° to −120° and from −60° to +180°. The robot collides with the obstacle while α is included in [−120°; −60°] (**Figure 7**).

The computational cost required to compute this c-map is almost zero.

Another trivial example is related to robots able to translate on a 2D map. In the case of a robot represented by a single point, the *collision free c-map* corresponds exactly to the areas not occupied by the obstacles. If robots are defined by other shapes (circles, polygons, etc.), calculating the c-map might looks like trickier than the previous example, but there is a practical way to do it. The algorithm has to make the robot slide around every obstacle to the define the boundaries of the

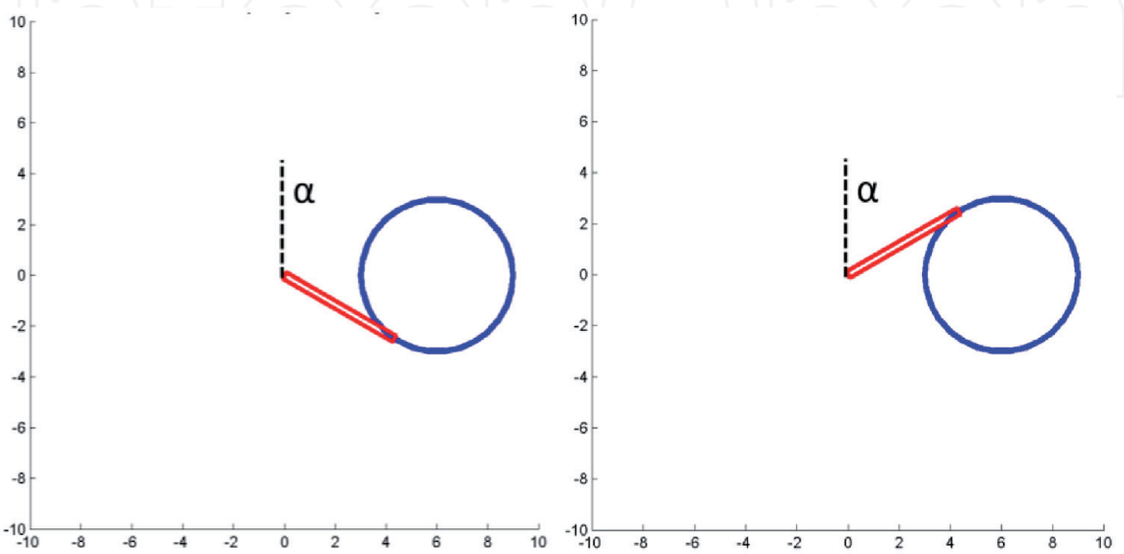

Figure 7.
Planar robot position for α equal to −120° and −60°.

collision free c-map. Afterwards, the algorithm has to fill those area to reconstruct the full map. **Figure 8** shows an example of a polygonal robot able to translate along both X and Y axes in [−8; +8]. In the scene there are two obstacles: a circle and a polygon. The robot cannot rotate around the z-axis and its location is referred to the position of the yellow point placed on one of its edges. The light-blue square represents in **Figure 8** is the area where the yellow point can move.

Let us take into account one of the most popular c-map examples: a two link planar robot having two degrees of freedom represented by the angles of the rotational joint connecting the first link to the ground and the rotational joint connecting the first link to the second one. The simplest scanario includes just one obstacle represented by a circle (**Figure 9**).

Due to the complexity of defining a single equation which describes accurately the *collision free c-map*, the algorithm had to simulate all possible configurations of the robot and check if it would collide with the obstacle. Obviously, the number of possible robot configurations are infinite because each degree of freedom is defined on the interval (−180°; +180] ∈ ℝ. In order to solve this issue, the algorithm had

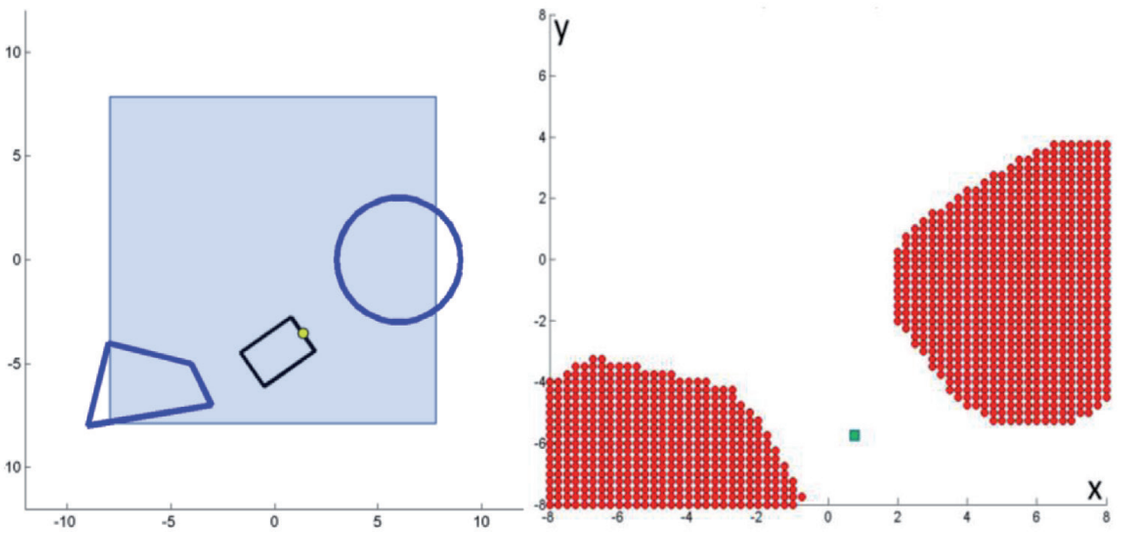

Figure 8.
C-map of a polygonal robot able to translate on a 2D map.

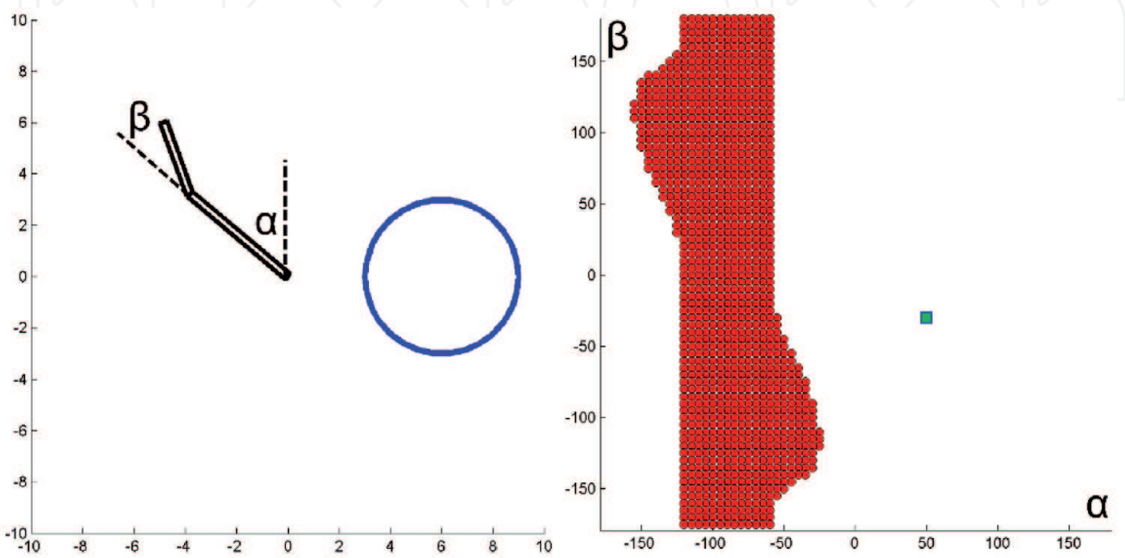

Figure 9.
C-map of a planar robot defined by 2 links and 1 circular obstacle.

to discretize that interval. The c-map represented in **Figure 9** has been computed by discretizing each degree of freedom by 5°. At this resolution, the whole process required 5148 iterations and it took 8 seconds on the test machine. On average, just 1 second have been used to evaluate object collision. The other 7 seconds have been used for reading and writing memory cells.

Before continuing with the chapter, is better to highlight the fact that every experiment has been performed on the same machine using a mono-thread fashion algorithm in order to avoid the possibility of using too powerful computers. Some algorithms, such as the c-map simulation search, have each iteration totally unrelated to the previous ones. This fact gives scientists the possibility to project a system which uses as many cores as the number of expected iterations to return the algorithm result immediately. Unfortunately, this approach might work in theory, but it is practically impossible for most of the cases because it would force the robot to be linked to a supercomputer which cannot be contained inside of the robot. Let us imagine that having an external facility containing this large computer would be feasible and that the algorithm should calculate the *collision free c-map* of a robot shake which has 9 links and each link is connected to the following one by a 3 axes rotational joint. Moreover, since the robot has to face a very narrow scenario, each degree of freedom has to be discretized by at least 1000 samples. The algorithm should iterate $\mathbf{1000}^{3*9}$ which is approximately the number of atoms of the universe.

In order to have a better understanding of the computational resources required to compute the *collision free c-map*, let us take into account the same two link planar robot, but with just one polygonal obstacle in the scene (**Figure 10**). Finding whether the robot collides with a polygonal obstacle is way more expensive as described in book [9, 10]. The number of edges of the obstacle covers an important role, but in this chapter every polygonal obstacle is assumed to be defined by no more than 5 edges for sake of simplicity.

Creating the *collision free c-map* of this scenario required the same amount of iteration as the scenario represented in **Figure 9**, but the whole process lasted 46 seconds. Similarly to the previous example, 39 seconds have been used to evaluate object collision and 7 to write and read memory cells. Computing c-map on a 5-edge polygon obstacle was 39 times longer than computing it on a circular obstalce.

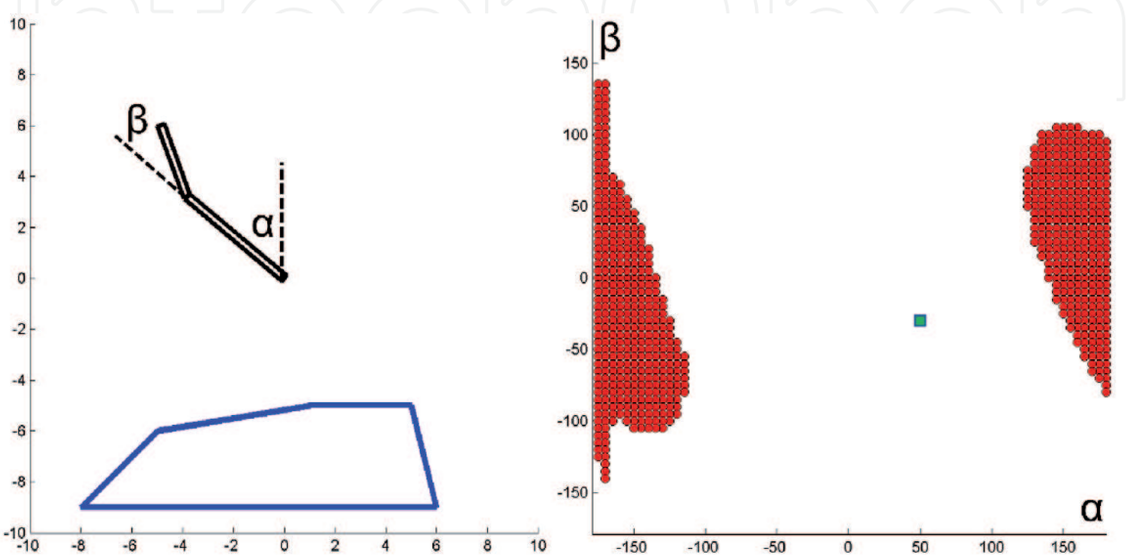

Figure 10.
C-map of a planar robot defined by 2 links and 1 polygonal obstacle.

Adding one circular object in the scene means increasing the computational cost by 1 while adding a polygonal obstacle means increasing the computational cost by 39.

In order to prove that linearly increasing the number of degrees of freedom will exponentially increase the computational cost, a bunch of experiments have been run on three link planar robot. The number of iterations required to find all possible robot configurations with a resolution of 5° is 373248. Calculating the *collision free c-map* of a scenario including one polygonal and one circular obstacle lasted almost 5000 seconds on the machine used to run all experiments reported in this chapter.

2.3 C-space usage

As soon as the c-map has been computed, another piece of the algorithm takes the lead and tries to find the path leading the robot from its initial configuration to the goal one. Sometimes, the path is a single straight line connecting the original point to the goal point on the c-map, but most of the times, the path is defined by a set of segments (**Figure 11**); each point connecting a segment to the following one is called way-point. Unfortunately, not all scenarios have a solution: some c-maps have more than just one obstacle free sector. If the robot is located into one of them, it cannot jump into the other one (**Figure 11**).

As it was mentioned in Section 2.2, the algorithm in charge of computing the c-map had to perform the colliding check of all representative configurations of the robot which means that the algorithm had create a grid of cells defined by two parameters: their position into the map (robot configuration) and a boolean value which states whether the robot collides with an obstacle or not.

Mathematically, the algorithm converts a $\mathbb{R}^n$ space into a $\prod_n d_i$ space, where n is the number of degrees of freedom and d_i is the number of samples for the i^{th} degree of freedom. The algorithm had basically clustered the infinite states of a system into the most representative ones.

According to S. M. LaValle [2], the are many ways to compute a path starting from the c-map. Let us take into account, as an example, the potential field path planner. Briefly, the potential field path planner associates to each cell of the c-map to another value which comes from the sum of the attractive potential field and the repulsive potential field. The first one (a) guides the robot configuration to the goal one, while the second one (b) pushes the robot configuration far from obstacles (**Figure 12**).

Afterward, starting from the initial configuration cell, the algorithm has to check which cell among the surrounding ones has the lower potential value and takes that as the last explored cell. Then, the algorithm has to repeat the search until

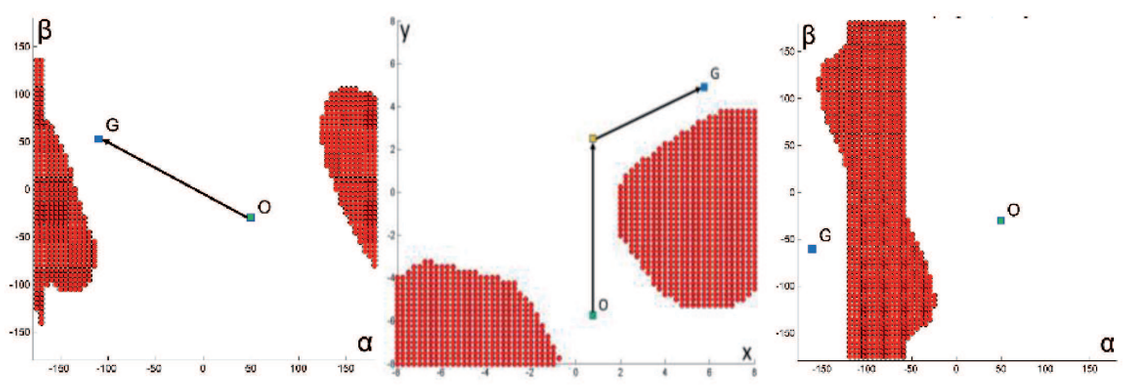

Figure 11.
Typical 2D path solutions.

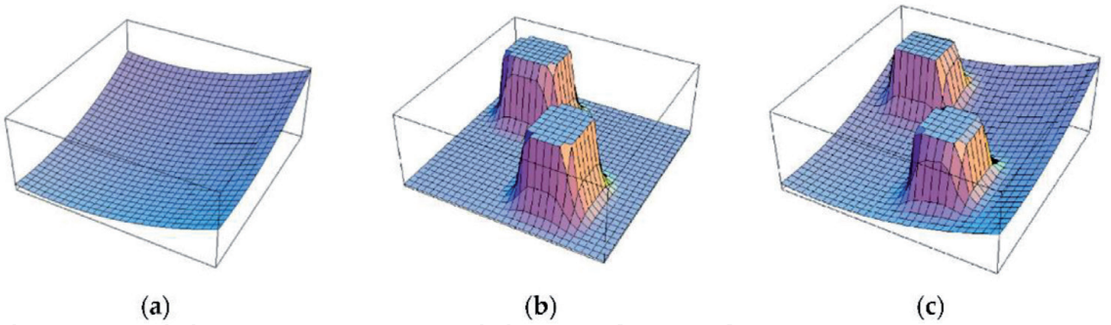

Figure 12.
Graphical visualization of a generic 2D potential field map having two obstacles in the scene (attractive field + repulsive filed) [11]. (a) shows the component of the potential field that attracts the robot towards the goal configuration; whereas (b) is the component that pushes the robot away from the configurations in which it would collide with the obstacles; (c) shows the final combined potential field.

the goal cell has been reached or it gets stuck into a local minima. This approach is no longer the state-of-the-art because it has many drawbacks which are not reported in this chapter. However, it has been used as an example because it is very simple and gives the perfect picture of performing a path search on a cell map.

As it was shown in **Figure 11**, creating the path by analyzing cell by cell might be very inefficient. Sometimes, there is even a straight line connecting the goal configuration and the initial one, but the algorithm has to waste time performing the whole c-map computation anyway. A possible strategy to overcome this issue is using a heuristic method.

3. Heuristic methods

Heuristic methods become very popular because they are able to find a good solution (non-optimal) in a very short time. They are specifically popular to for solving scheduling problems. Heuristic methods are not analyzing the problem step by step like most of the deterministic algorithms; they are starting directly from a potential solution. Then, they adjust this solution iteratively in order to reduce the outcome of a cost function. Heuristic algorithms look for alternative solution randomly or pseudo-randomly at every iteration until a certain time or number of iterations has been reached. In [12] a variety of heuristic methods are described. Scheduling problems find a perfect match for these algorithms because sorting randomly all items the algorithm have to schedule is already a solution. The algorithm is not supposed to create or erase some items. Path planning algorithms, instead, do have to create a set of waypoints connecting the initial configuration to the goal one on the c-map. Way-points are a set of scheduled configurations that the robot has to cross sequentially in order to reach the goal. In path planning algorithms way-points are the items to be scheduled.

Nowadays, heuristic algorithms able to create or erase items from the original schedule pool are included in the family of genetic algorithm [13, 14]. Within each generation, the number of items of the schedule is constant. From one generation to another one the number of items might change.

Coming back to the potential field planner, it practically tries to connect the initial configuration and the goal configuration by selecting the less expensive cell in term of potential field at every iteration. Each cell is a sort of waypoint that the robot has to cross. The outcome of the cost function related to potential field planners is the sum of the potential costs of all cells the algorithm plans to cross. The potential field planner deterministically finds the path by analyzing the map

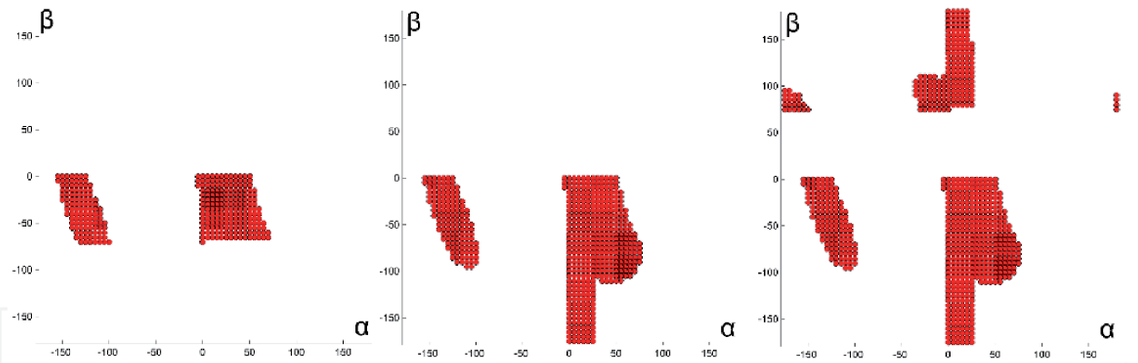

Figure 13.
C-map computation progresses (20%, 50%, 80%).

step by step. An alternative way to find a feasible solution is drawing a straight line connecting the goal point to initial one at first. Afterwards, if the path intersects an obstacle of the c-map, the algorithm should introduce a waypoint and making it slide on the map randomly for some iterations. If a feasible solution has not been found, the algorithm should introduce another waypoint and so on. The reader of this chapter might argue that this process might get stuck if there are no solutions because there is no way for the algorithm to understand it deterministically. (S) He would be right, but this is the drawback of using heuristic algorithms. They are very quick to find solutions if they exist, but the algorithm's target is not finding the optimal solution. However, if the algorithm is well designed heuristic algorithms are one of the most powerful weapons to tackle path planning problems quickly.

Unfortunately, the heuristic algorithms described so far are performing the path search on the c-map. It means that the c-map has to be computed in advance anyway. Apparently, using heuristic methods speeds up the path search, but do not solve the time issue raised up by the *collision free c-map* computation appointed in Section 2.5. However, looking closer to the process of creating the c-map the reason of such a time-consuming process is obvious: the c-map computation algorithm is deterministically discretizing all possible robot configurations. At the end of the iterations the statistical distribution of configuration samples is perfectly flat, but it is unbalanced for the whole process. **Figure 13** shows the progresses of a c-map computation (20%, 50%, and 80%).

The accuracy of the map is very high where the c-map has been computed, while it is null where the map has still to be processed. A feasible way to keep the accuracy homogenous over the whole map during the entire computation process is computing cells randomly on the map. Due to the "Monte Carlo's simulation", picking points randomly on a dataset converges quite quickly to the statistical distribution of that dataset. In this case, the dataset are the cells of the map and their statistical distribution is flat because cells are equally distant from each other.

At this point, a quite simple question might come out: "Is analyzing all cells mandatory in order to find a feasible path?". The answer, of course, is "no". This process of configuration space discretization is required to create a full c-map, but even rarefied c-map can lead to feasible and acceptable paths. The family of sampling based planners is the result of this idea.

4. Sampling based planners

Sampling based planners finds their strength into reducing dramatically the number of robot configurations taken into account during the c-map construction

process. Instead of creating the full c-map, few points are randomly selected into the map in order to reduce the number of robot configuration taken into account without loosing the statistical distribution of the map. The most common algorithms belonging to this family are the probabilistic Roadmap (PRM) [3, 4] and the Rapidly-exploring Random Tree (RRT) [5, 6]. Both algorithms are throwing into the scene random points one by one and tries to connect them to the closer one in order to build a graph (PRM case) or a tree (RRT case). A connection is considered valid if the line connecting a point to another one is not crossing an obstacle. Practically, the algorithm is locally computing a c-map while it tries to connect two points. At first, it might look not so efficient because the algorithm seems to replicate the classical c-map construction with the drawback of having just a graph or a tree and not the full map. However, as soon as the number of degrees of freedom overpasses 2, the computational cost difference between building the full c-map and using a sampling based planner the significantly increases.

The first is optimal for multi agent planning algorithms because graphs do not have a starting node. The initial configuration of the robot can be located anywhere on the map and linked one node of the graph. Similarly, the goal configuration can be linked to a node of the graph. Afterwards, a graph search algorithm (Dijkstra's algorithm [15], A* [16], etc....) will be in charge of finding the best path on request. Exploring trees, instead, are based on node hierarchy. Every node is connected the others using a father-child fashion hierarchy. Every node must have a father except for the root node. The root node is the one corresponding to the initial robot configuration. This means that exploring the tree is extremely very fast and simply but, the algorithm has to compute the tree if the initial robot configuration changes.

5. Conclusions

Nowadays, heuristic and meta-heuristic methods are widely used because they are incredibly efficient in term of computational cost. Moreover, if they are well designed, they are able avoid local minima which are far from the global minima. Philosophically, heuristic methods are the family of problem-solving algorithms closer to human thinking. Humans uses to solve problems by attempting it practically, simulating it in their mind, or doing both at the same time. However, humans require much more time to perform this search than computers. This is one of the main reasons why drug development is so quick today. Scientists do not practically mix compounds all the time; most of their job is simulating chemical reactions.

Computing c-maps deterministically is very expensive or even impossible for a large variety of scenarios. Heuristic methods appear to be a solution because they are very quick and allows scientists to have at least a non-optimal feasible path. So, the sampling based planner family comes from the necessity of unifying c-map computation and heuristic methods into a new path planning technique.

Acknowledgements

This research has been funded by Generalitat Valenciana (PROMETEO/2020/034).

Author details

Emanuele Sansebastiano and Angel P. del Pobil*
Robotic Intelligence Lab, Jaume I University, Castellón de la Plana, Spain

*Address all correspondence to: pobil@uji.es

IntechOpen

References

[1] M. Elbanhawi and M. Simic, "Sampling-based robot motion planning: A review," in IEEE Access, vol. 2, pp. 56-77, 2014, DOI:10.1109/ACCESS.2014.2302442.

[2] Cambridge University Press. "Planning Algorithms", by S. M. LaValle. Available at http://planning.cs.uiuc.edu/; 2006.

[3] L. E. Kavraki, P. Svestka, J. Latombe and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," in IEEE Transactions on Robotics and Automation, vol. 12, no. 4, pp. 566-580, Aug. 1996, DOI:10.1109/70.508439.

[4] Yan, F., Liu, YS. and Xiao, JZ. Path planning in complex 3D environments using a probabilistic roadmap method. Int. J. Autom. Comput. 10, 525-533 (2013). DOI:10.1007/s11633-013-0750-9

[5] LaValle, S. "Rapidly-exploring random trees: A new tool for path planning." The Annual Research Report. 1998

[6] Rodriguez, Xinyu Tang, Jyh-Ming Lien and N. M. Amato, "An obstacle-based rapidly-exploring random tree," Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006., 2006, pp. 895-900, DOI:10.1109/ROBOT.2006.1641823.

[7] E. Arnold, O. Y. Al-Jarrah, M. Dianati, S. Fallah, D. Oxtoby and A. Mouzakitis, "A survey on 3D object detection methods for autonomous driving applications," in IEEE Transactions on Intelligent Transportation Systems, vol. 20, no. 10, pp. 3782-3795, Oct. 2019, DOI:10.1109/TITS.2019.2892405.

[8] de Berg, Mark and Cheong, Otfried and van Kreveld, Marc and Overmars, Mark; 2010. "Computational geometry: Algorithms and applications (3nd edn.)", Springer Publishing Company. ISBN 3642096816 DOI:10.5555/1951877

[9] Bayer, Valentina. "Survey of Algorithms for the Convex Hull Problem," Technical report, Oregon State University; 1999.

[10] Gamby, A.N.; Katajainen, J. Convex-Hull algorithms: Implementation, testing, and experimentation. Algorithms 2018, 11, 195. DOI:10.3390/a11120195

[11] Jeon, G.-Y.; Jung, J.-W. Water sink model for robot motion planning. Sensors 2019, 19, 1269. DOI:10.3390/s19061269

[12] Silver, E. "An overview of heuristic solution methods". J Oper Res Soc 55, 936-956, 2004. DOI:10.1057/palgrave.jors.2601758

[13] Beasley, David, Bull, David R. and Martin, Ralph Robert. "An overview of genetic algorithms: Part 1, fundamentals". University Computing 15 (2), pp. 56-69. 1993

[14] Lingaraj, Haldurai. "A study on genetic algorithm and its applications". International journal of computer sciences and Engineering. 4. 139-143. 2016

[15] Dijkstra, E.W. "A Note on Two Problems in Connexion with Graphs." Numerische Mathematik 1 (1959): 269-271. <http://eudml.org/doc/131436>.

[16] P. E. Hart, N. J. Nilsson and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," in IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100-107, July 1968, DOI:10.1109/TSSC.1968.300136.