SAPIENZA
UNIVERSITÀ DI ROMA

Dipartimento di ingegneria Informatica, Automatica e Gestionale
(DIAG)

Planning with Dynamical Obstacles

Professor:
Prof. Paolo Liberatore

Autore:
Marco Nacca
mat. 1978636

Anno accademico 2022/2023

Contents

1 Introduction

Planning with dynamical obstacles is a significant area of research in the field of robotics and autonomous systems. It deals with the challenging task of generating motion plans for agents or vehicles operating in environments where obstacles exhibit dynamic behavior. These obstacles can move, change their shape, imposing additional complexities and uncertainties to the planning problem. The ability to effectively navigate through environments with dynamical obstacles is crucial for a wide range of applications, including autonomous vehicles, aerial drones, mobile robots, and human-robot interaction scenarios. In such dynamic environments, traditional static path planning approaches may fall short, as they do not account for the evolving nature of obstacles and their potential impact on the agent's trajectory. The objective of planning with dynamical obstacles is to generate motion plans that enable an agent to safely and efficiently navigate from an initial state to a desired goal state while avoiding collisions with both static and dynamic obstacles. This requires the agent to consider not only the current position and velocity of the obstacles but also their predicted future movements and potential interactions with the agent's trajectory. To address the challenges posed by dynamical obstacles, researchers have developed various planning techniques and algorithms that incorporate dynamic obstacle information into the decision-making process. In this paper, several existing approaches will be presented to study how the topic has been resolved.

One of the possible approaches that can be used concerns planning based on observation, through the use of so-called safety intervals. A safety interval is a collision-free period for a specific robot configuration. The planner exploits this observation and constructs a search space in order to traverse different states with the aim of avoiding moving obstacles, as seen in [1].

Another approach presents a strategy that is to frame motion planning as a chance-constrained optimization problem. Chance-constraints allow for a probabilistic formulation that guarantees a certain level of safety. The algorithm leverages this concept by incorporating chance-constraints into the motion planning framework. By modeling the uncertainties using probabilistic techniques, the planning technique estimates the future motion of the obstacles and computes the chance-constraints accordingly. To efficiently solve the chance-constrained problem, the algorithm employs various techniques such as stochastic optimization, convex optimization, or sampling-based methods. [2].

Instead, we have a different solution with regard generating the path planning scheme to achieve the goal using a genetic search algorithm. In this case what we are going to consider

are different paths that we can take, chosen on the basis of a natural, hence biological, selection criterion that allows us to select the best choice on the basis of a process of "natural selection" among the possible candidates present, as cited in [3].

The last approach involves contact driving and thus in urban environments. Urban environments present unique difficulties for autonomous systems due to the presence of various dynamic obstacles. Driving in urban environments requires interaction with other vehicles. This iteration can involve both communication between the vehicles themselves and the placement of sensors on them that can detect the dynamic movement of obstacles that may be other vehicles, pedestrians, or bicyclists. In fact, this environment can change dynamically, and we need systems that can detect these changes and find a real-time solution to solve the problem. [4]

Overall, planning with dynamical obstacles represents a challenging yet crucial research area for achieving safe and efficient navigation in dynamic environments. The development of effective planning algorithms and strategies will play a pivotal role in enabling autonomous systems to operate seamlessly in complex and ever-changing real-world scenarios. By considering the dynamic nature of obstacles, these planning techniques aim to enhance the robustness, adaptability, and intelligence of autonomous agents, paving the way for the widespread deployment of autonomous systems in various domains.

2 Preliminaries

In order to accomplish various task, robots need to be able to predict where these dynamic obstacles will be moving in the near future. They need to plan short paths to their goals that do not endanger objects. The prediction must be made quickly because the robot may have incorrectly predicted the trajectory of the obstacle or the environment can change and therefore we have a reformulation of the actions to be carried out.

As mentioned above, to solve the problem in question, several approaches have been developed. In the following section, we will see in detail the methods used by the various works. For what concern [1], for simplicity, it is considered, in theoretical discussion, both the environment and obstacles as static objects. In addiction to this, the configuration of the robot is defined as the set of non-temporal variables that describe the state of the robot, as a position, header, joint angle. Therefore, the state of the robot is defined by its configuration and an independent variable which is the safe interval that identifies the state and the planner uses the states to decide where to move to avoid the collision with a dynamic object.

As for the [3], genetic algorithms are very efficient at finding an optimum path in very large workspaces. As in the previous work, also in this case planning schemes with GAs (Genetic Algorithms) that includes obstacle avoidance are off-line path planning methods under static environments. In the proposed path planning method, a genetic searching algorithm is used in generating via-points after finding the objects by the vision system and it's used a simple cost function which consists of distance and safety measure for short and safe path generation to the goal. This function is called **fitness function**. The algorithm starts by initializing a population of potential paths, represented as individuals in the genetic algorithm. Each path encodes a sequence of waypoints or actions for the autonomous system to follow. The population is then subjected to a series of operations inspired by genetics, including selection, crossover, and mutation. During the selection process, individuals with higher fitness, are more likely to be chosen for reproduction. The crossover operation combines the genetic material of two selected individuals to create offspring with a mixture of their characteristics. In the context of path planning, this involves exchanging segments of paths between parents to generate new paths that inherit favorable traits. To introduce diversity and exploration, the mutation operation randomly modifies certain aspects of the paths in the population. This allows for the exploration of alternative solutions and prevents premature convergence to suboptimal paths. After each generation, the fitness of the new population is evaluated, and the selection, crossover, and

mutation processes are repeated. This iterative cycle continues until a termination condition is met, such as a maximum number of generations or reaching a satisfactory solution.

As for [4], on the other hand, as already mentioned, we have several steps to deal with, and to do so we first need a structural treatment of the composition of urban environments, which have complex roads, traffic, various obstacles that can move, and other dynamic objects that need to be identified. These objects, in particular, involve variable behaviors that cannot always be predicted with great accuracy, and so their detection is of paramount importance to our work. To track these dynamic objects, we need special instrumentation that can detect changes in the surrounding environment, and it is essential that the components of this instrumentation communicate with each other to exchange information. In particular, LiDAR (Light Detection and Ranging) type instrumentation capable of tracking the movement of dynamic objects in real time is used in [4] [6]. As mentioned, prediction, through prediction techniques, of the trajectories of obstacles to predict their future path is also of paramount importance. Finally, navigation in the urban environment is subject to various constraints, such as traffic rules, pedestrian right-of-way, road infrastructure, and legal and socially acceptable navigation behaviors. These described steps are of fundamental importance to the discussion of the related algorithm.

3 Problem Statement

After having given a general point of view of the topic, seeing the initial considerations made by the different approaches, in this section we will see specifically the formulation of the problem. In particular, we will go into more detail regarding some algorithms, while for others there will be a more general explanation. As for the [1], as mentioned in the previous section, we define a safe interval as a contiguous period of time for a configuration, during which there is no collision. On the other hand, a collision interval is a contiguous period of time for a configuration, where each timestep in this period is in collision with a dynamic obstacle. Each configuration has a timeline that's an ordered list of intervals, in which there are safe and collision intervals. In addition, each save interval will necessarily be followed by a collision interval, and the same is true for the latter.

Instead, in the approach used by [3] we have that the mobile robot is guided by a supervisor which consists of a vision system and path planner that can be external or locate on the robot. This system helps the mobile robot to reach the desired position avoiding the moving obstacles by providing a safe and short path to the goal. As already mentioned previously, the main problem is to have a dynamic environment in which we must necessarily have real-time identification of the moving objects and generation of reasonably short and safe dynamic path to the goal.

At this point, both as regards [1] and [3], suppose that there is another system that traces the dynamic obstacles in the environment and involves their future trajectories. Through this, we are able to build a dynamic path planning scheme based on a genetic search algorithm for fast generation of via points that the robot can follow to arrive to the goal position. In particular, the attention is pose on the reduction of search space for generation of safe path.

Finally, can be considered a further approach, as seen in [4], in which the problem can be divided into 3 main discussion points: detection of dynamic obstacles, prediction of future motion of dynamic obstacles and how to avoid dynamic obstacles. Regarding the first point, what we make use of is sensors, as it is one of the elements that can provide us with support for obstacle detection. Obviously, we have to purposely use sensors with different characteristics and then combine the results obtained. As for the prediction of dynamic obstacles, we always tend to consider them as static. There are some approaches, such as the one mentioned in the last section [2], that use probabilistic notions to predict the robot's course but even these rely on the idea that the robot does not change its direction of movement because we would need too much detailed information that is impossible to achieve since a degree of error may always be present.

4 Algorithms

In the following section, we will go in more detail as regards the algorithms used to solve the problem. In particular, more details will be inserted as regards algorithms relating to [1], [2], [4].

4.1 SIPP Algorithm

In particular, as seen in [1], it is assumed for the first time that there is another system that traces the dynamic obstacles in the environment and predict their future trajectories. There is a list of dynamic obstacles, in which each obstacle has a radius and a trajectory. A trajectory is a list of points, in which each point has state variables, specifying its configuration and time. The points that define the trajectory are orderly following the time, so it is possible to understand how the obstacle in the future is possible from these. In addition, for simplicity it is assumed that each obstacle only travels a trajectory and that this is static, therefore does not change dynamically. In fact, the algorithm that is used is the A^* but, as seen in [1], the function is modified to obtain the nodes that are successor to the current node. Precisely, in the following work we will not report the algorithm A^* as it is already known, but we will present a variation in one of its external functions to find the successor node. In particular, an $M(s)$ function takes into account all the possible moves that can be made from a node s. These moves will also have a time to be performed, so once the configuration is selected, we place ourselves in the interval of time i-th in which we do not have collisions and select the successor node (i.e. the next configuration) with the slightest quantity of time to arrive in this new configuration and that, at time in which arrive in this new configuration, it is in the time interval in which it has no collisions, in particular is in the safe interval. If, on the other hand, our safe interval ends and subsequently begins the safe interval of another configuration, there is a danger that during the move we can meet an obstacle, then we eliminate the successor node protagonist of this situation from the possible ones, setting the time variable $t = \infty$, how we can be seen from the figure taken from [1]. Again, the variable i indicates the various safe intervals inside the same configuration. So, for example, configuration A has two safe intervals.

In this case, the robot is located at configuration B and can move one cell per time-step in the others configurations. In Figure 1, can be shown the graph during the expansion of the robot's current state, S_{B0}. The other state in configuration B is not presented because you cannot transition between safe intervals within the same configuration because the robot would have to wait the collision interval before go in another safe interval. In this case, the robot could move

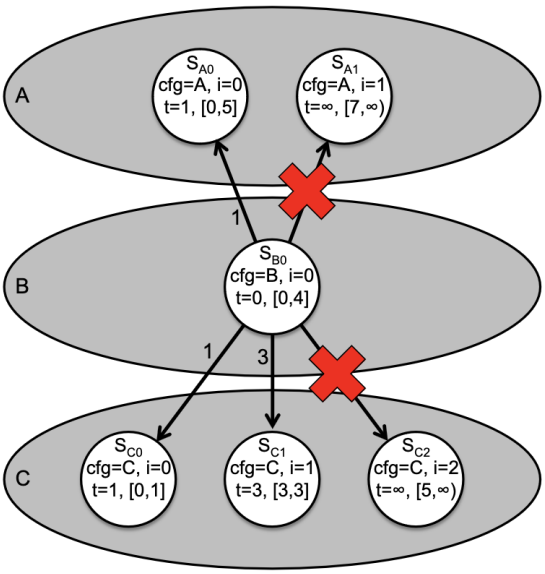

Figure 1: SIPP expansion

immediately to A and arrive in safe interval 0, or the other successor in A is at safe interval 1, however this interval starts at time 7 and we can only safely wait in S_{B0} until time 4, after which we would get hit by the upward moving dynamic obstacle. So, as said before, this transition is invalid. The time (t) for S_{A0} is 1 because it takes one time-step to move one cell, while the $t = \infty$ for S_{A1} because there exist no valid path. For the downward motion to configuration C there are three possible successors. First, the robot could move down immediately to S_{C0} in one time-step. Another successor is S_{C1}. To arrive at time 3, the robot would have to wait 2 time-steps before moving to C because 1 time-step is for arriving at that configuration. In addition, the other successor is S_{C2} but, as seen before, this transition is invalid because the safe interval in the departure state ends at time 4 and that relating to the state S_{C2} begins at time 5, then in that waiting time-step there may be collision with an obstacle.

Below, we can see the algorithm used by A^* to find the successors:

```
def getSuccessors(s):
    successors = [];
for each movement in M(s):
    configuration = getConfiguration(s)
    time_movement = getTimeToExecute(movement)
```

```
    start_time = time(s) + time_movement
    end_time = endTime(interval(s)) + time_movement
    for each safe_interval in configuration:
        if getStartTime(safe_interval) > end_time or getEndTime(safe_interval) < star
        continue
        timeToReach = getTimeToReachConfiguration(i)
        if t is None:
            continue
        s_ = getConfiguration(safe_interval, t, configuration)
        successors.insert(s_)
return successors
```

Now we can see in more detail an example showing how the algorithm works. The following differs from what we showed in the figure 1 in that we preferred to show in a simpler and more explanatory way how the planner works.

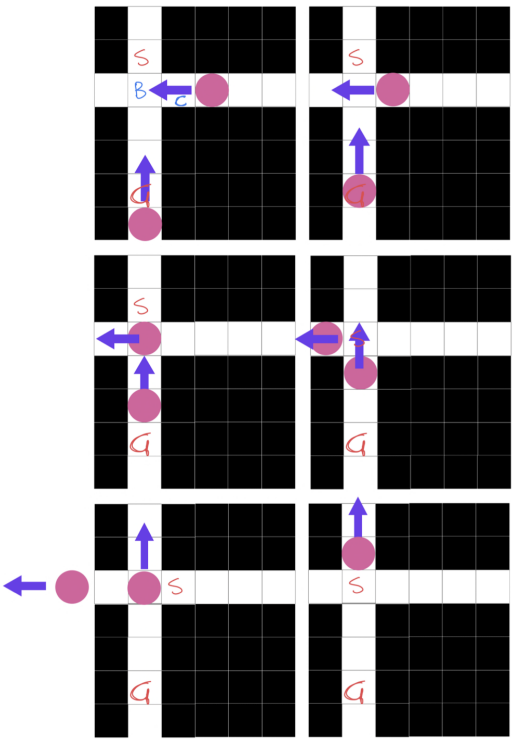

Figure 2: SIPP example

First, we show the environment and the actions taken by the planner; moreover, the robot starts its progress from the position marked 'S' and ends its journey in the cell marked 'G'. We present for simplicity two dynamic obstacles, colored purple, moving from right to left and from bottom to top. These obstacles move on a non-square grid. The robot and dynamic objects move one time-step. The cell where the starting state is has 4 safe intervals and therefore we have the possibility to move to 4 states (represented by the configuration of the robot and the save interval as mentioned earlier). In this case, we can make two choices: either generate a B0 state in the first safe interval because we do not collide with any dynamic object, but then we would have to go back (to the S0 state) since we have a dynamic object that passes. The other alternative would be to wait for 2 time-steps in the S0 state until time t = 2, where we would then move to the C0 state (passing through the B1 state at time t = 3 and arrive in C0 at time t=4). As we can see from the figure, the alternatives that have a yellow mark are the obsolete ones since, by staying waiting in a certain state, we risk colliding with an object. This is because, as explained earlier, we move to the new configuration in the shortest possible time, and therefore, if we identify that to arrive at a given consideration we must take two time-steps, we will ensure that we are in the final configuration at time t = 2, thus putting ourselves "on waiting" for one time-step and performing the action in the other time-step available. If, for example, I can arrive in the configuration having a save interval at time t = 3, this means that I have to wait 2 time-steps and then move to the desired configuration.In fact, given this, going to the C0 configuration has a cost of 2 compared to stopping at the B1 configuration which has a cost of 3. Having said that, we now have the situation where the dynamic object that has to move from the bottom to the top is an obstacle for us and this is shown in Figure 4. Once we are stationary in configuration C0, we can move to the third safe interval of the cell to our left (configuration B2) since there is no longer any dynamic obstacle that has to pass. From here on, we can proceed to our goal. Below we also show the graph of the processed situation, where the choices that were made by the planner are highlighted in blue color.

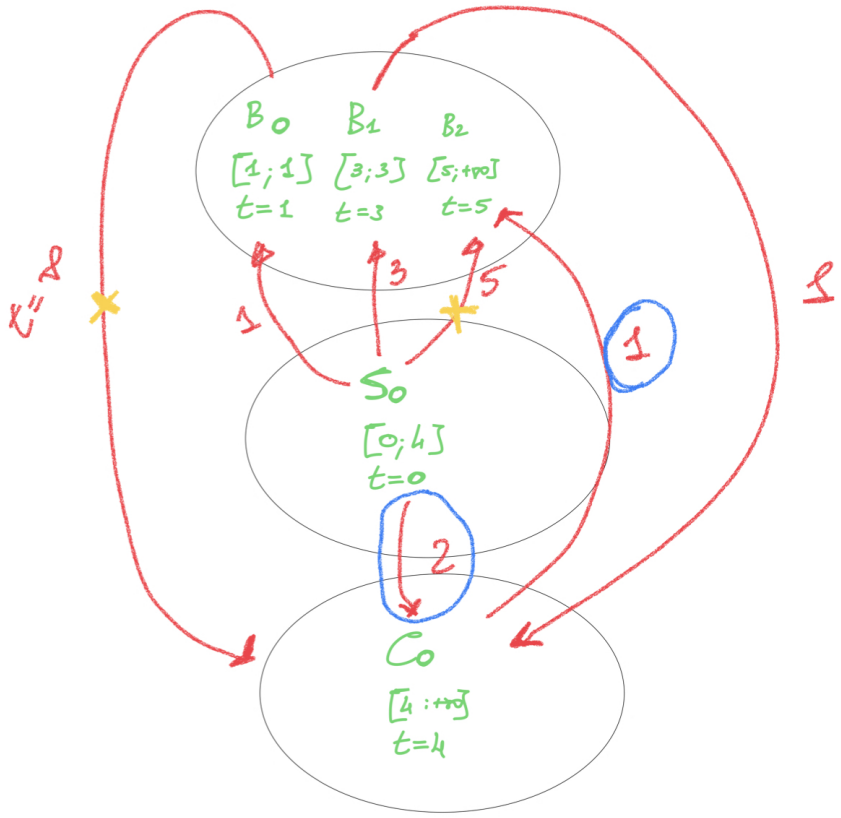

Figure 3: SIPP graph

One of the notable advantages of the SIPP algorithm is its ability to handle complex dynamic scenarios. It can efficiently plan paths in real-time, even in highly congested environments with multiple moving obstacles and complex motion patterns. Additionally, the algorithm provides guarantees of safety and optimality, making it a reliable choice for many robotics applications.

4.2 Genetic Algorithm

As mentioned above, the starting point is to identify the movement of obstacles. This also applies to the case described in [3], where this knowledge of the position of the final point is added. Genetic algorithms (GAs) are research and optimization algorithms based on the mechanics of natural selection in biological systems. It implements a strategy of "survival of physical fitness" within a population of potential competing solutions. Each solution is represented by a set of

parameters codified in binary strings, which are called chromosomes. The fitness function in a genetic algorithm evaluates the quality or suitability of an individual (a potential solution) within the population. The goal of the fitness function is to assign a numerical value to each individual, indicating how well it performs in solving the problem at hand. The individuals with higher fitness values are more likely to be selected for reproduction and contribute their genetic material to future generations. A simple algorithm having three basic operations is used to cause an evolutionary exchange and these three operations are selection, crossover, and mutation. This allow a selection based on "survival of the fittest", and exploration of previously unvisited parts of the search domain, crossover and mutation. The generation given by the evolution of the population of potential solutions results in a convergence towards the "best" possible solution. In particular, can be seen various step to perform the algorithm:

- Problem Setting: Define a representation for your paths or trajectories. It could be a sequence of waypoints, a set of control parameters, or any other format that captures the necessary information to describe the paths.

- Fitness Evaluation: Determine a fitness function that quantifies the quality of a path in terms of its ability to reach the goal while avoiding obstacles. The fitness function should take into account factors such as path length, clearance from obstacles, smoothness, and other domain-specific criteria.

- Initial Population: Generate an initial population of candidate paths. These paths can be randomly generated or based on some heuristics. The population should consist of a set of potential solutions that will evolve over time.

- Genetic Operators: Define genetic operators, such as crossover and mutation, to manipulate and create new paths based on the existing population. Crossover combines genetic information from two parent paths to produce mixed paths, while mutation introduces small random changes in a path. These operators mimic the natural genetic processes of reproduction and mutation.

- Selection: Perform selection to choose paths from the current population for the next generation based on their fitness values. Higher-fitness paths are more likely to be selected, but some level of diversity should also be maintained to prevent premature convergence to suboptimal solutions.

- Recombination and Mutation: Apply crossover and mutation operators to the selected paths to create a new generation of paths. The offspring paths inherit genetic information from the parent paths, allowing the algorithm to explore different combinations of path segments.
- Evaluate Fitness: Evaluate the fitness of the new paths in the population using the fitness function defined in step 2.
- Termination Condition: Determine a termination condition for the genetic algorithm, such as a maximum number of generations or reaching a satisfactory fitness threshold.
- Iterate: Repeat steps 5 to 8 until the termination condition is met.

As said before, the path is composed of intermediate points that are sets of parameters codified in binary strings. In this case, the coding size is of fundamental importance because, through this, we determine the cost of the overall calculation for a certain fitness function and therefore there are different possible ways to reduce the size of these strings. Assuming that the initial positions of the robot and the objective are known, the length of the binary string can be abbreviated by projecting the two -dimensional data to the single-dimensional ones because the robot, of departure, is represented in two dimensions in the coordinates (X, Y).

As we can see from the figure, we have that the G nodes are positioned at the same distance on the line that connects the initial and final position that must travel the robot. The set of points G becomes the research space for each intermediate point of the route that must perform the robot. To reduce the size of the data in each research space and accelerate the speed of the genetic research algorithm, a reallocation of the starting points on each via-point is dynamically implemented.

In the figure 4, the intersection between the G_i parallel straight lines and the line that starts from the initial point to the end point is called *node* and all points on the parallel straight lines are called **knot**; So, starting from the initial point, we can choose several knot as starting points associated with each node.

In addition, the fitness function associated with each string (or parameter) is calculated taking into account the distance from the final point (and therefore from the goal) and from the distance from the obstacle.

It increases when the distance from the final point is small (short path) and the distance from

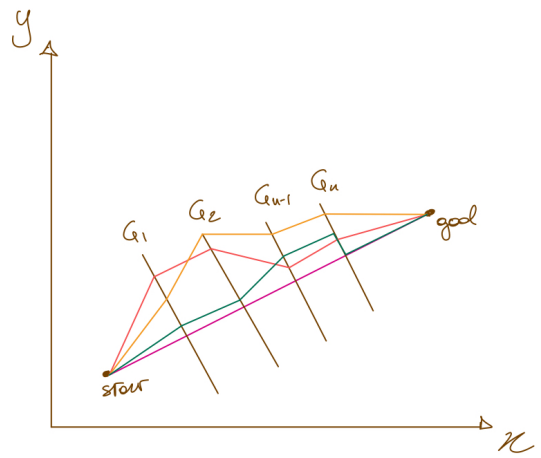

Figure 4: GAs coding structure

the obstacle is wide. It is calculated in this way:

$$fitness\_value = \alpha \cdot \sum_{i=1}^{n} \frac{L_i}{l_i} + \beta \cdot \sum_{i=1}^{n} d\_min \tag{1}$$

where [3]

- l_i: distance between the knot point and goal
- L_i: distance between the node point and goal
- d_min : the minimum distance between the knot point and obstacles
- α, β: the weighting factors.

The fitness function is composed by two components: the first one is the rate of distance of node position to that of knot position both from the goal position for short path consideration. This is equivalent to minimizing the deviation of the robot path from the global minimum path of straight line from the start to goal position. The second is the minimum distance of robot to the obstacles for safe path consideration. In addition, the weighting factors α and β can be adjusted to obtain the optimal via-points for short and safe path to the goal while avoiding the

moving obstacles. Here, can be seen a pseudo code of an implementation for GAs, including the most important functions that can be used:

```
    function GeneticAlgorithm():
    population = InitializePopulation()
    generation = 0
    while TerminationConditionNotMet():
        fitnessValues = EvaluateFitness(population)
        parents = Selection(population, fitnessValues)
        offspring = Recombination(parents)
        mutatedOffspring = Mutation(offspring)
        newPopulation = Combine(population, mutatedOffspring)
        population = newPopulation
        generation = generation + 1
    bestPath = GetBestPath(population, fitnessValues)
    return bestPath

function InitializePopulation():
    population = []
    for i = 1 to populationSize:
        path = GenerateRandomPath()
        population.append(path)
    return population

function EvaluateFitness(population):
    fitnessValues = []
    for path in population:
        fitness = CalculateFitness(path)
        fitnessValues.append(fitness)
    return fitnessValues

function Selection(population, fitnessValues):
    parents = []
    for i = 1 to numberOfParents:
```

```
        parent = BestSelectionWithHigherFitnessValue(population, fitnessValues)
        parents.append(parent)
    return parents

function Recombination(parents):
    offspring = []
    for i = 1 to numberOfOffspring:
        parent1 = RandomlySelectParent(parents)
        parent2 = RandomlySelectParent(parents)
        child = Crossover(parent1, parent2)
        offspring.append(child)
    return offspring
```

```
function Crossover(parent1, parent2):
    child = CombineGeneticInformation(parent1, parent2)
    return child

function TerminationConditionNotMet():
    return generation < maxGenerations
```

4.3 Planning in Urban Environment

Instead, regarding the development of algorithms for planning in the presence of obstacles in the urban environment, as mentioned earlier, as a first step we have to create a model of the dynamic objects. As we see from the figure present in [4], we can have both a model in the form of a box, which contains different information about the shape of the vehicle, and we can have a point model. Both are used depending on the characteristics of the sensor. As for the box model, it resembles the model of a bicycle, while the point model is identified by its coordinates in two dimensions. Obviously, using prefixed models can solve the complexity of the problem, but models similar to real ones can also be used. Of fundamental importance is the role of processing data from sensors. These output features that must be validated in some way in order to interpret their meaning, which must then be validated to make sure that the information given is correct. This validation can be done by comparison with the actual map of the obstacles and the road, so that the information from the sensors can be interpreted and the dynamic objects can be printed out in point or box form. Having these data available, we can create a system that differentiates objects that move from objects that we assume to be stationary; if an object has its own speed, we assume that it moves while if we observe that the obstacle is stationary for a given time interval, we assume that it is stationary and therefore can be treated as a static object.

The following code will be a pseudo code of a possible obstacle detection implementation using the information discussed previously.

```
function detectObstacles(sensorData):
obstacles = []
obstacles_moving = []
obstacles_static = []
for each sensorReading in sensorData:
    if isObstacleDetected(sensorReading):
        obstacle = extractObstacleInformation(sensorReading)
        obstacles.append(obstacle)
        if isObstacleMoving(obstacle)
        obstacles_moving.append(obstacles)
        else
        obstacles_static.append(obstacles)
return obstacles
```

Instead, as regards the prediction of the movement of obstacles, the process itself is already complicated as everything is based on probabilistic assumptions. What could be useful instead is to consider the probability of a certain action by a vehicle considering the structure of the road; in fact, if we are on a straight road, it is much more probable that the moving obstacle continues along the road, while if we are close to a crossroads, the probability is limited to the roads present in that crossroads. Therefore, to carry out an implementation discussion of what has just been said, we need the position and speed of each obstacle that are given to us by the sensors based on the box model and the point model respectively. Having this, as previously mentioned, we are going to make a projection of the information on the model of the road, also considering the lanes into which the road is divided. With this, we calculate in which lane the obstacle is present, or if it is centering within a lane, for example after entering from an acceleration lane. In addition, we also consider the speed trend, as we know that if we have a stop line or an intersection, then we can predict that the speed of the obstacle under consideration will decrease over time until it reaches zero. A different matter is made if we consider that during the journey we encounter parking lots or curves, as in this case the structure of the road is different. In particular, it could be considered that in curves and in parking lots, the speed is lowered and therefore one can react in such a way as to avoid collisions.

The following code will be a pseudo code of a possible obstacle motion prediction implementation using the information discussed previously.

```
function predictObstacleMotion(obstacles, laneData):
predictedObstacles = []
for each obstacle in obstacles:
    predictedObstacle = obstacle
    predictedObstacle.position = predictPosition(obstacle)
    predictedObstacle.velocity = predictVelocity(obstacle)
    if getObstaclesAtIntersection(obstacles):
        predictedObstacle.lane = selectLaneAtIntersection(obstacle, laneData)
    else:
        predictedObstacle.lane = assignLane(predictedObstacle.position, laneData)
    predictedObstacles.append(predictedObstacle)
return predictedObstacles

function selectLaneAtIntersection(obstacle, laneData):
    obstacleLanes = getLanes(obstacle, laneData)
    laneProbabilities = calculateLaneProbabilities(obstacleLanes)
    laneWithHigherProb = chooseLaneWithHigherProbability(laneProbabilities)
    return laneWithHigherProb
```

Finally, we have the last step which is the discussion of dynamic obstacle avoidance. In this phase, we have the generation of several candidate trajectories and our algorithm will have the task of selecting these trajectories generated by the [5] on the basis of a hierarchical procedure. Therefore, given a trajectory for our vehicle and a presumed trajectory for the dynamic object, what we do as a first step is to build a box that surrounds the space occupied by the trajectory; this box is taken, as seen in [4], in a pessimistic way as we don't enlarge the box more than necessary. After that, what we do is check if, in following the trajectories, the two trajectories intersect: if they do, we have to keep looking for another trajectory.

The next step, once we have chosen the two trajectories, is to build a circular box around the vehicle and around the robot and check if they intersect. If they do not intersect all the way, then they can be chosen, otherwise a polygon is constructed around each one; if the latter also intersect, we start from scratch otherwise it can be considered the same as a candidate trajectory. Below, we show the implementation code for dynamic obstacle avoidance.

```
function avoidObstacles(currentPath, predictedObstacles):
```

```
safePath= currentPath
for each obstacle in predictedObstacles:
    if obstacle.isInPath(currentPath):
        alternativePath = determineAlternativeLanes(currentPath, obstacle)
        safePath = chooseSafePath(alternativePath)
        break
return safePath
```

5 Conclusion

Overall, planning with dynamic obstacles is a vital area of research and development in autonomous systems. Considering the presence and movement of obstacles, these systems can navigate effectively and avoid collisions, ensuring safety and efficiency.

In this context, various algorithms and techniques have been developed to address the challenges posed by dynamic obstacles. These algorithms often incorporate sensor data, prediction models, and lattice algorithms to generate feasible, collision-free trajectories.

A commonly used approach is integrating sensing systems to track obstacles, such as sensors. By continuously analyzing obstacle data, these algorithms identify and track dynamic obstacles in the environment. Predictive models then estimate the future movement of these obstacles, providing valuable information for planning and decision-making processes. Also, including lane information in dynamic obstacle planning adds another layer of complexity. Finally, we have also seen how algorithms based on biological action, ie genetic algorithms, make a significant contribution to research, by reducing the space of solutions, to find the optimal solution by following biological processes.

As technology advances, machine learning and artificial intelligence techniques are also being leveraged to improve dynamic obstacle planning. These techniques allow systems to learn from experience and adapt their behavior based on observed obstacles and environmental conditions. Machine learning models can improve prediction accuracy, improve obstacle classification, and optimize trajectory generation.

References

[1] Mike Phillip and Maxim Likhachev, SIPP: Safe Interval Path Planning for Dynamic Environments, 2011.

[2] Manuel Castillo-Lopez; Philippe Ludivig; Seyed Amin Sajadi-Alamdari;Jose Luis Sanchez-LopezMiguel;A. Olivares-Mendez and Holger Voos, A Real-Time Approach for Chance-Constrained Motion Planning With Dynamic Obstacles, 2020.

[3] Woong-Gie Han; Seung-Min Baek and Tae-Yong Kuc, Genetic Algorithm Based Path Planning and Dynamic Obstacle Avoidance of Mobile Robots, 1997.

[4] Dave Ferguson; Michael Darms;Chris Urmson and Sascha Kolski, Detection, Prediction, and Avoidance of Dynamic Obstacles in Urban Environments

[5] T. Howard and A. Kelly, "Optimal rough terrain trajectory genera- tion for wheeled mobile robots," International Journal of Robotics Research, vol. 26, no. 2, pp. 141–166, 2007.

[6] LIDAR, Wikipedia.