POLITECNICO DI TORINO

Master Thesis in Mechatronics Engineering

A ROS-based motion planning for socially-aware navigation

Relatori
prof. Alessandro Rizzo
dott. Stefano Primatesta

Studente
Daniele Fidelio
matricola: 245397

Academic Year 2019-2020

Abstract

In the last years, autonomous mobile robots are increasingly present in our daily life and, in the next future, they will be increasingly present in our house, offices and in public environments. Today, it is quite common to see a robot cleaner. But, in the future we will have a complete coexistence between humans and robotic platforms.

The presence of robots in populated areas could cause some issues. The main one is the co-presence of humans and robots in the same environment. The robot should consider the presence of humans, respecting social rules and providing a safe and comfortable motion, to achieve human acceptance and trust. The robot should imitate humans, providing a human-like motion. In this way the humans will accept the presence of the robot, considering the robot like a one of them.

The research field of studying how to move a robot evaluating the presence of humans is called socially-aware navigation.

The goal of this work is to develop an algorithm to compute a socially-aware trajectory for mobile robots. In particular, the trajectory must be accepted by humans. To do that, the robot should have a human-like behavior. Generally, humans move with a collision-free trajectory, observing other movements and anticipating other people, to avoid possible further collisions.

Our aim is to *"anthropomorphize"* the robot behavior. In the literature, many studies have already published on this topic. Many approaches try to solve this problem focusing only on the robot, without considering humans as social entities. In this work, we consider humans on the populated environment.

Many common motion planning approaches provide a reactive behavior. The robot sees the obstacle and, then, tries to avoid it just reacting on the obstacle motion. However, humans use a predictive approach to move in crowded environments. Humans observe the obstacles and other humans and avoid them by predicting their motion.

In order to solve the socially-aware navigation problem, we use a game theory approach. For instance, assuming the chess game scenario, we try to predict the further moves of other player in order to decide the next move. Similarly, we apply this idea in our algorithm.

The proposed approach is based on the implementation of a game theory method. We choose a particular type of game. This game has the following features:

1. *Dynamics.* The game evolves during the time.
2. *Non-cooperative.* Each player thinks only to its own goal.
3. *Non-zero sum.* The sum of all gains and losses, of all players, is not strictly equal to 0.

Using this solution, each player moves toward the goal and interacting with other players. In order to find an optimal action, we use the concept of Nash equilibrium. This concept is applied to reach a feasible solution between all the players.

Moreover, the proposed approach considers the presence of physical objects to avoid. In this way, the robot plays with other players, but avoiding at the same time, obstacles present on the environment. To do that, we define a cost function to evaluate the cost of actions and a cost map of the environment, to define the probability of collision with obstacles. We assume that the robot knows its pose in the environment and, then, it continuously computes the action to reach the goal by computing the cost of actions, to chose the best one for itself and other neighbor players.

The algorithm will be validated using real-world surveillance videos, often used on this field. In this way, we can verify the quality of algorithm. We check if the trajectories computed by algorithm are feasible and human-like. In the first part of development, we checked how robot manages the movement in some particular environments. After this first step, we tested the algorithm with the video surveillance. Here, we put the robot inside the environment that will move toward a specified goal, among other players.

Contents

List of Figures

Chapter 1

Introduction

In 1495, Leonardo da Vinci realized the first Humanoid robot. Reading the notes of Leonardo, the mechanic knight realized was able to stand up, move its arms, head and jaw, thanks to a complex system of pulley.

In the eighties, the idea of smart robots starts to become true. This idea creates an industrial revolution. In these years, in fact, many robots are developed to help humans in heavy and repetitive work. The japanese manufacturers redesign their own production lines in order to introduce robots.

Over the years, robots become more and more present. This spread are helped to the introduction of *AI*, artificial intelligence, onto robots. This evolution allows robots to do a enormous leaps forward. Many bounds have been overcome and nowadays robots are present in many fields.

In the future, the robots can hold limitless possibilities. The evolution of AI and the rapid implementation on robots leads to create independent and smarter devices. Soon, humans and robot will be able to works closely and realize complex jobs.

Nowadays, robots are present in many field, in order to improve the final result, to reduce time and final cost. The main application are:

1. *Industrial Robots.* These robots are useful to replace humans in simple and repetitive task.

2. *Autonomous Mobile Robots.* These robots can be remotely controlled and perform task that are impossible or very dangerous by humans, as pipe inspection or bombs deactivation

3. *Educational Robots.* These robots are quite inexpensive and are deployed in schools to improve the quality of learning.

4. *Humanoid Robots.* These kind of robots is used many fields. For example, these robots are used to reproduce mechanics walking or some particular aspect of humans. This ability allows robots to interact easily with humans.

1.1 People behaviour

Today, the interaction between people and robots is becoming very common. During the twentieth century, the world has seen a widespread of robots and artificial intelligence to solve problems in other ways unsolvable for humans or that would need an enormous loss of lifes. Nowadays, instead, robots are far more common in all places, such as homes, hospitals etc, helping people in everyday. This spread highlighted an issue that was unthinkable before. How do people react in front of a so wide presence of robot? How do people interact with them? What is an acceptable robot behavior for a human being? Which should be the rules a robot must follow to be human-safe?

This challenge creates coming goals that engineers must solve to create robots that are acceptable for an everyday use, with human beings. The navigation issue will be the main problem to solve. In fact, we interact everyday with other people and in future we expect our feeling towards this interaction and robots movements to be similar with that we experience with common people. The main aspect to improve, in order to increase the quality of collaboration between people and robots are:

1. Human-likeness motion: the human beings use a smooth trajectory to reach the goal, robot must do the same movement.

2. Safe motion: human beings usually avoid both inter-human collisions and abnormal movement and behaviors that could worry the others.

3. Effective and reliable motion: the robot could reach the goal according to its own characteristics and physical principles.

4. Interaction awareness: humans can predict were people would probably go, and knowing it, they modify their own trajectory.

If the previous requirements are respected, the cooperation between people and robots will became easy to understand and intuitive, avoiding strange situation.

1.1.1 Human inclination to anthropomorphize

Human beings always try to attribute human aspects to a biological agents such as wind, sun, water, supernatural, like gods, witch, wizard and so on [1]. This behavior is very common in many fields, also in computer science. If the robot is perceived as an human, it would be able to have social influence or moral care. considering the anthropomorphization as the main goal in robot designing the major efforts are in this direction.

With evolution and toward centuries and millennia, humans have developed a strong sense of interpersonal cooperation in order to facilitate everyday tasks [2]. Nowadays, with robots entering in everyday life, a similar behavior of cooperation and interaction should be developed towards them. The appearance of robot must be similar to a simple transposition of humans. The autonomous movement is the most important aspect to develop in order to avoid interference with human mind.

1.1.2 A solution

To find a possible solution of socially aware robot navigation, we need to analyze in deep the human behavior and state of mind.Using all information coming from psychology and cognitive science [3]. In fact, as previously said, while till now complex robots have worked only in controlled spaces such as laboratories and under extremely controlled and strict conditions, our aim is to adapt them to social and complex life, being able to move in the real world and in unpredictable conditions.

1.1.3 Human spatial interaction

Inter-human interactions when moving in the same space are the first point that should be analyzed. When people move in public and private spaces their trajectories frequently intersects each other, obviously this intersections should be solved to avoid collisions. While the trajectories changes can appear completely casuals in truth they always follow complex and methodical schemes. To correctly interact with us also robots should follow these schemes when moving in human populated spaces [4]. To do that, we can use the wide knowledge available in literature. In fact, this aspect of human being has been widely studied in psicology. We can take information from the work of Aiello [5] and Burgoon [6]. In these works, the authors explain how humans reacts when something ore someone occupies the space around them. In particular, they described 4 different ranges of distances in which the behavior strongly differs:

1. Intimate: this range is less than 45 cm from the item in analysis. The interaction in this space implicate a physical contact.
2. Personal: the range is between 45 cm and 120 cm. These distances are normally present between friends and family components when interacting in everyday life. Similarly, this range is used in well organized social interaction, like queue.
3. Social: the range is between 1.2 m and 3.5 m. This range is used normally to have interaction with unknown people and it is the distance that separates people in public space, like shops, beaches, etc.
4. Public: more than 3.5 m. We don't have interaction with other people in this range. It is the distance between unknown people.

1.1.4 Socially-aware navigation requirement

If inter-human distances and robot-human distances represent a starting point in social interactions also social-awareness has an important role. With the term "social-awareness" we refer to robot abilities to interact with a crowd without disturb it. Similarly, when we refer to robots navigation we talk of socially-aware navigation referring to move and interact with crowd seeming human. According a survey done by Kruse et al.(2013) [7], normal interactions are based on three different aspects that make them socially accepted, these aspects must be guaranteed also by robots:

1. Confort: the absence of stress coming from the interaction between human and robots.

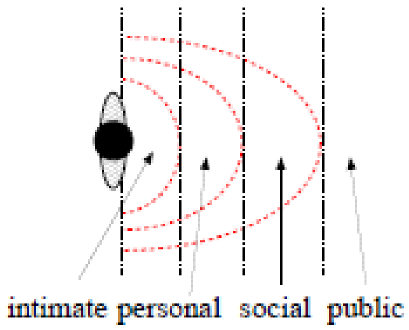


Figure 1.1. The space saw by human being

2. Naturalness: the low-level behavior between human and robot must be similar.

3. Sociability: the capability of robot to learn high-level cultural convention

To accomplish these requirements, we need to understand and predict conditions in which the robot can create discomfort. To sum up, robots must respect the following aspects:

1. Respect personal space. The distance between people is very important. If a person is too near to another, this situation can generate peeve. To avoid that, we need to respect social distances according to the situation. The robot must evaluate the distance from a human being, in order to avoid complaint and fear.

2. Respect activity space. Robot should recognize and avoid the zone that human could occupy to perform some actions. This bond is hard to reach because the robot needs to understand what the human is planning to do and calculate the space that he should avoid.

3. Respect group of agents zone. When interacting people occupy a different kind of space, called O-space. This space has a O shape and is variable. In fact, it depends on the number of people, their orientation, and speed of action and movement. Studies reveal that usually people interact while positioned on circles [8].

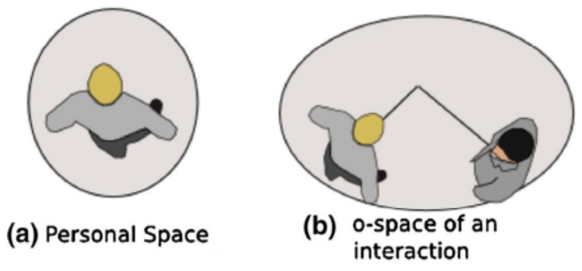


Figure 1.2. Personal and O-shape space

1.2 Thesis explaination

The goal of the thesis is the development of a method for a socially-ware navigation. This solution is chosen in order to realize robots with a human-like behavior. The starting point is the issue of navigation and we choose this solution thanks to its completeness. The work is based on a methodology proposed in [1], a thesis project developed by a colleague at the "Complex Systems Lab". Specifically, in this thesis the socially-aware navigation proposed in [1] is implemented with the ROS (Robot Operating System) framework.

After analizing the behaviour that our robot should have to be socially accepted we tried to translate this knowledge into a code that respects the bonds analyzed.

To develop a suitable code, the main concepts of game theory was studied and analyzed. This branch of science is very useful because a lot of models can be used to perform the correct behavior that should be given to a robot. In particular, in the game theory we have players that are allowed to do everything they need to reach a goal. But the main characteristic is the independence of each player. At the same time, every player plans its moves taking apart the other players' possible decision. This is the powerful aspect of game theory, that will be useful in the developing of code.

The development of code follows the *microscopic approach*. This solution permit to predict the movement of each other player, without considering the general movement of players. Instead, if it was chosen the *macroscopic approach*, the general movement of players was analyzed.

Also, the game was selected as no-cooperative. Each player decide its own movement.

Other features of the game were the dynamic and non-zero-sum solution. The main goal of the game was to avoid the collision between player, predicting how the players will move in future.

To chose which path are more feasible then other, the *Nash equilibrium* was used. The game ends only when the Nash equilibrium was reach, i.e. this equilibrium measure the best response for all players to the problem presented. The algorithm developed improve the standard approaches. In fact, usually the motion planning was developed taking in consideration only to avoid collisions with people and object. This algorithm, instead, can be considered *predictive*. Every player adapts its strategies considering the other players decision. We also implement a human-like way to avoid obstacle. According with the work presented in article [9], the robot must be interact with the obstacle. In this algorithm we didn't take in consideration the *activity space* because we consider the robot operating in an urban space, without any specific activities. Summarizing, the goal of this thesis is:

1. Improve existing motion planing algorithm to make it acceptable by humans.
2. Develop a ROS software implementing the method proposed in [cit Giada] to be implemented on a real robotic platform.

Chapter 2

State of Art

2.1 Robot navigation

The navigation is a main aspect of robots. Today, how the robot moves itself in the world is the target of many researches. Making this aspect correct and acceptable for our way of thinking is the core of further implementation of robotic system. The main characteristics of robot navigation are:

1. *Localization*: to estimate a position for the robot, according to a reference system, in order to localize the robot inside the map.

2. *Mapping*: the Robot analyzes the environment and it creates a map, in order to recognize its own position in the operational space.

3. *Planning*: the Robot elaborates a sequence of poses, in order to reach the goal with consecutive feasible positions.

4. *Motion Control*: once that the goal are chosen, the ability of robot to move itself, according to the chosen path, to reach the goal.

The robots have many sensors installed to know what kind of environment surrounds them. They use these devices to sense the environment. In this way, they can use an active localization:

The *Active localization* consists in the ability of the robot of using the data coming from the propioceptive sensors to elaborate the odometry. It also combines these data with the data coming from exteroceptive sensors, thanks to some kind of probabilistic filters. Usually, we have two kinds of filters: parametric and no-parametric. The more important parametric filter is the Kalman filter, with extended or unscented versions. The Particle filter, instead, is the more important no-parametric filter, based on the Montecarlo Localization.

For what concern the mapping process, we can use two kinds of approaches:

1. *Landmarks*: they build a stochastic map where we can find a probabilistic description of environment and obstacles.

2. *Occupation grids*: create a map divided in sectors. Each sector has an occupation probability.

When the robot decides to move, it usually uses the *SLAM*, Simultaneous Localization and Mapping. This solution solves *concurrently* the issue of Localization and mapping.

For this thesis, the localization issue is bypassed. For simplicity, the robot will know the map before the game, and that map will have static obstacles. Knowing it the robot will apply the *SLAM* algorithm to localize itself in the environment. Also, it will recognize the obstacle and where they will be. After this preliminary stage, the developed algorithm will be launched.

2.2 Motion Controller

When we deal with the path planning, we need to consider the necessity of avoiding obstacles. In fact, the robot must avoid the obstacle in order to reach the goal. The robot will use the sensors to localize the obstacle and calculate the movement to reach the target, knowing where the obstacle is. Doing that, the robot creates a path in a *local* manner. According with the work done by Khatib [11], the problem could be solved using a time-varying potential field. The space that surrounds the robot will be modified in order to create an attractive force to the goal and a repulsive force to the obstacle. Another example in literature is the solution found by Boreinstein et al. [12]. In this case, the researcher use a *Vector Field Histogram algorithm (VFH)* to compute a polar obstacle histogram. In this way, the algorythm studies each histogram and chooses the more acceptable to a given threshold. The biggest problem of this technique is the possible oscillation between positions in a world full of bonds. This issue can generate a non-smooth path. To improve this algorithm, researchers developed two evolutions of that. The *VFH+* [13] and the *VFH** [14].

1. *VFH+*: we have two phases. Firstly, we read the angle and distance from the obstacle using the sensors. In this way, we can build a polar histogram where we can find the obstacle. After that the algorithm chooses an histogram, solving the oscillating behavior of the robot characteristics of the VFH algorithm. VFH+ inserts a variable cost according to the steering direction and chooses the direction with the minimal cost.

2. *VFH**: is an enhanced version of VFH+, obtained applying heuristic functions.

Until now, we consider only the *local* view of the space. This approach represents a limitation for the optimal solution. The optimal solution for the local view couldn't be the optimal for the goal solution. To solve this issue, some researchers found a way to link the *local* and *global* planning. An example is the Elastic Bond approach developed by Quinlan and Khatib [15]. This algorithm changes the parameter of global planning in real-time according with the internal and external forces available in the field. Thanks to that we can create a smooth path to reach the goal.

2.3 Motion Planning models

2.3.1 Predictive model

For the predictive model we have some examples in literature. One of the more interesting was Paris et al. [21] work. They create a velocity-based model that improves the Fiorini and Shiller [22] model. In the *Fiorini and shiller* model we have a model that avoid collision knowing the current positions and speeds of the robot and the obstacles. This model has an issue, the oscillating behavior of robot. The *Paris'* technique solves the issue of speed oscillation of player. This problem comes out due to the lack of knowledge and prediction of external world. With this solution the subject is not pushed away from the other entity, but it finds a free way to reach the goal through the crowd. The parameters evaluated by the model are:

1. All possible directions.
2. A restricted set of speeds.

These parameters are evaluated simultaneously.

After that model knows these data, it looks for all possible collisions between player and other pedestrians. It calculates the path that reduces cost, and applies it for the successive instants.

Another work that improves the previous model is Karamouzas et al. [23]. This model is based on Paris et al. [21]. With this different approach, the aim is to reduce the computational cost. More in particular, the cost reduction is focused when the algorithm elaborate the path to avoid proximate collisions.

The continuous improvement of this idea, leads to a new model developed by Warren [24]. In this case, pedestrians compute their path considering at the same time 3 frameworks:

1. The closest framework.
2. An intermediate framework.
3. The furthest framework.

In the closest framework, the pedestrian repulses every person, because this framework correspond with the personal Hall space [25]. The intermediate area is where the algorithm calculates the speed and direction in order to avoid the collision with other pedestrians. The furthest area, instead is where all pedestrians can move freely.

In a continuous model, the margin of error increases similarly with the horizon of events. This issue is well-known and some researcher tried to find a solution, that was reached by Trautman et Al. [26]. In this work, the crowd is studied continuously, as time passes, and pedestrian reactions have a Gaussian distribution. The development starts from the *Freezing Robot Problem*, where we have players that are stuck because they cannot perform a successful movement. It can happen if the scenario is too crowded. In this condition, players remain stuck or move randomly without any sense. To solve this problem, the robot computes also the data coming from the external environment in a Gaussian process, and try to find the solution.

This technique has some advantages. The most important is the ability to represent the human navigation in a realistic way. This technique is more feasible and can store a lot of information respect to a reactive model. The disadvantages are the computational cost and the uncertain when we have a crowded scenario.

2.3.2 Reactive model

The reactive model is another model that can be useful to represent human crowds. This model has been developed comparing the human crowd with physical particles, more in particular the attractive and repulsive forces between particles have been compared to those used by the crowd to evolve [27].

In this model we have two kinds of forces:

1. *Attractive forces*: allow pedestrians to move towards the goal.

2. *Repulsive forces*: allow pedestrians to avoid collision.

The Hoeller et al. approach [31] goes towards this direction. their hypothesis was that humans move similarly to particles in physics. Humans are subjected to repulsive and attractive forces that change their trajectory. The combination of these forces generates the path that human follow to reach their own goal. To model this behavior one main problem must be overcome: the goal of people is unknown.

To solve this issue, the idea was to create a possible area that people apparently want to reach. Pedestrians are considered only *passive*, and the external forces manage their movement. This is a big limitation, because Hoeller did not manage the interaction between humans, that was the main problem when trying to reproduce a correct human-like motion. In fact, human beings are *active* and check thee environment to move, constantly. This aspect was completely deleted with the Hoeller's model.

Other works solved the problem using the same idea of particles. Considering the idea of two kind of forces, repulsive and attractive, both depend from the distance, similarly to particles. An issue is that in the local point of view, the movements are chaotic and do not represent a human-like behavior. To solve the problem of local movement some authors [28] try to lock the behavior of the players. They impose that players could go only straight on the current direction and at a fixed speed.

Another approach is the *Cellular Automation* [29]. This model is based on a grid motion decision, it represents in a discrete way the scenario around the human being. In this way the human being has a finite number of choices to manage the environment. Once that we have a specific number of choices, we can apply a probability distribution to create a predictive model that will manage the movement of human in the near future.

After this new concept, Schadschneider [30] improves it introducing the idea of *Floor Field*. This Floor Field is another grid that follow exactly the *Cellular Automation* aspects. This second grid can be dynamic, evolve during the time, or constant, its remain always the same during the time. It is used to improve the *Cellular Automation (CA)*. In fact, the CA is a grid independent by the time, instead the Floor Field is dependent and can be used to better implement the point of interest on the map. Also, the Floor Field can manage the evolution of grid due to the interaction between pedestrian, and improve the computation.

2.3.3 Learning model

This kind of approach is different from the ones already explained. The previous model was focused on pedestrian and how they decide to move and behave when they find a problem. In the *Learning model*, instead, the near-future decision is estimated considering the observed behavior of each player. We also include the observation of environment and how it changed during the past. This idea is based on *machine learning*. This new branch became more and more popular due to its ability to reproduce and adapt behavior of players if something unexpected happens. It has a ductile approach that can adapt and manage the unexpected situations. Also, we can play in advantage. We can educate the player before the beginning, i.e. the majority of computational cost can be done a priori. This process request a lot of data, but the a priori possibility solves the problem.

Bennewitz et al. [32] created a model that can be used to manage the trajectory path of players in a crowded environment. Their model take as input a various set of trajectories. Also, they insert some point for each player, where the player can stop and remain for a specific amount of time. These places are called *Resting Place*. This model observes how the environment and player evolve during the time and learn which regulate the movement. After that, it individuates the group of trajectory that each player can follow in the near-future. The algorithm chooses which motion pattern are feasible using *Expectation-Maximization* algorithm.

An improvement of this model came with Foka et al. [33]. They applied a neural network to elaborate speed and trajectories of human beings. This approach improves the prediction of non-linear behavior. In this way, we can have a better prevision of near-future instances.

Other improvements follow the same concept. A better implementation of neural network. In particular, the *Recurrent Neural Network (RNN)* was a step further in the ability of prediction of future human motion.

A limitation of this model was that it does not consider how other people move in the proximity of the analyzed player. An improvement came with the idea of Alahi et al. [34]. They proposed the implementation of *Long short-term memory (LSTM)* applied to a social purpose. They insert a common sense rule that each player must follow to create its own path. Moreover, this must be applied inside a crowded scenario and must share the space between the players. Remind that, this algorithm evaluates how pedestrian can be move in the future, but don't analyses how these players can be interact each other. The output of this algorithm is a path, that the authors call *Average Behavior*.

An improvement of this algorithm came with Gupta et al.. They overcome the limit of previous algorithm introducing the *Generative Adversarial Networks*. This solution permits to improve the previous algorithm without not so high computation cost. The *GAN* is a learning method where two neural networks are in challenge inside the same framework.

The *GAN* solution seen previously was improved with the addition of *LSTM* model by Liang et al. [36]. Here we have the prediction of future, thanks to the data coming from various videos. The main advantages are exactly the data coming from the reality. Thanks to it, the path chosen by player are "real" and permit to player to seem human and manage a crowded scenario like an human being.

The learning approach permit to the algorithm to evaluate also the knowledge coming from experience. The experience gives the possibility to recreate a path acceptable or followed in the past. Also, in the interaction with other people, the player already knows some cases and how it behaves on them. The only issue is how generalize it. If we change the scenario or we introduce a new player, the player must be retrained to behave in the correct way. This issue create a loss of time and has a computational cost that decrease the efficiency of the algorithm.

2.3.4 Game theory model

Here we speak about the last type of model analyzed. Nowadays, the game theory is becoming more and more present in the computer science. The main reason is the ability to reproduce the behavior that players have in the real world. The game theory can reproduce rational decision. Also, it can take in consideration the presence or not of other players. It can cooperate with the other player or go alone. The application fields of this theory are very wide. Mainly, game theory find his fortune with the *Artificial Intelligence* field.

The Game theory, instead, has only few application on the human motion planning. In literature are present only some examples and the information are very limited. The more valuable work was done by Hoogendoorn er al. [37]. This study develops the idea to apply the game theory to the pedestrian navigation in a particular scenario. The idea is to realize a motion planning that reproduce an human-like movement. To do that, the researchers use a differential game. Here, we have the players that try to reach the goal using the optimal path, this computation returns a feedback and algorithm tries to minimize the cost, including the data coming from the path of other players. The solution is an optimal solution of the problem because we have the minimum cost. The problem is the lack of an equilibrium during the game.

The game theory was combined with the *Cellular Automata* [38]. With this implementation, the researchers Mesmer and Boebaum [39] develop a model that can be implemented in the human navigation. In particular, they enhanced the game theory introducing the speed of human being and obstacle avoidance. This method is useful to simulate the emergency. In particular, it is used to create a correct emergency plan to evacuate building and so on.

Turnwald et al. [40] study how humans interact each other during the navigation. This study analyzes this aspect in a microscopic way. First of all, the researchers choose a non-cooperative game, then they apply 5 different cost functions and see the results. To receive a better solution, they applied this algorithm to a real experiment with two persons. They found that the most suitable cost function depends on the lengths of the path done. Also, they success to validate the application of *Nash Equilibrium* in a human motion planning algorithm. In fact, the result of their work shows that the path chosen, i.e. the optimal solution, correspond to the path that generate the Nash Equilibrium.

Some other models are developed to represent how the humans react in a crowded scenario. One of these is the Roy et al. [41]. Here, the authors develop an algorithm that shows how two players avoid the collision between them. They applied the *Fokker-Plank Nash* game. We have a differential game on a particular scenario. This game tries to minimize the cost function of each player. This cost function depends on the collision of the players and the capability of them to avoid it.

All the methods showed are implemented with a low number of players. In fact, if the number of players increases, the computation cost became bigger and bigger. If this cost becomes too big to manage, the algorithm cannot find the solution and becomes impossible to manage. To manage it, the solution was find in the *Main Field Game*. Here the algorithm neglects the players that don't have impact with the system, decreasing the complexity and the computational cost.

Chapter 3

Game Theory and ROS environment

We choose to apply the Game theory in the thesis development. The game theory is chosen because is very useful and powerful to create a human motion model. In particular, the application of this solution in this field is quite new and we are free to apply it how we consider it appropriate. At the end, the game theory is included in a ROS code that allows to implement game theory concepts in a physical device.

3.1 History

Game theory is a branch of applied mathematics used to find the optimal strategy in uncertainly situations or with incomplete information. This theory models the situations of every day and finds a solution.

While the academic world formally established that the game theory was born in the 1950s, the game theory-like insight can be seen in centuries old history.

Ideas of game theory appear many times during the history. From the war solutions of Sun Tzu to the Charles Darwin discoveries. The basis of primal game theory can be seen in three specific works

1. *Researches into the Mathematical Principles of the Theory of Wealth* by Augustin Cournout.
2. *Mathematical Psychics* by Francis Ysidro Edgeworth.
3. *Algebre et calcul des probabilites* by Emile Borel

Each of this works introduces the basics ideas of the future game theory. They attempt to use math and computation to predict and map the human behavior.

3.2 General concept

Nowadays, the game theory is a wide used discipline. We can find the application of game theory in many fields like biology, computer science, economy, politics, etc. Game theory is a way to find model to reproduce common behavior, i.e. it creates a mathematical model able to reproduce the strategy behind a rational decision.

The concept of game theory was developed by Von Neumann in the 1928. Von Neumann creates a theory to solve a particular game, the zero-sum cooperative. After some years, he improves the game theory [16]. But the real revolution on this field happens when John Nash develops his theory. In particular, in 1950 John Nash develops the *John Nash equilibrium* applied in the non-cooperative game [17].

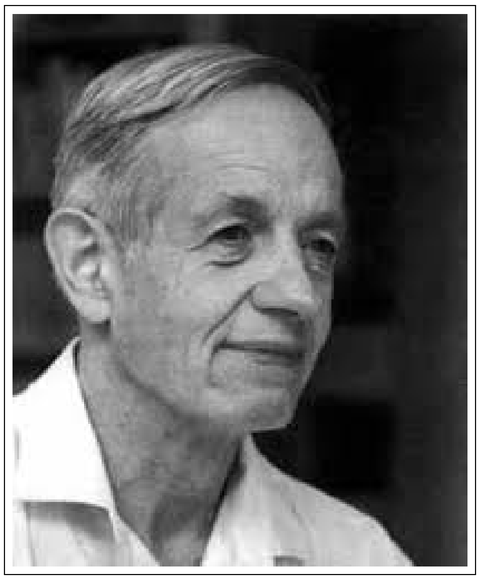


Figure 3.1. John Nash

3.3 Keywords

The following legenda explains the more common words used in the game theory.

1. *Players*: in a game, the members of it are called players.
2. *Actions*: well-known move that each player can use to reach the goal.
3. *Stages*: the number of steps that create the whole game.
4. *States*: here the game stores the data of each player in that particular stage.
5. *Strategy*: the state of mind that each player follows during the game to reach the goal.
6. *Cost Function*: each player has a variable that leads the choice of it.
7. *Rational Behavior*: The player tries to minimize its cost function.

3.4 Game Types

Games are various and exist many types of them. Games are collected accordingly to the features available. For simplicity, we enroll only the games that are linked with the thesis scope and could be used to represent the interaction between people and crowded scenario [18].

1. *Non-cooperative or cooperative* Each player works individually and tries to maximize its own profit. Players could cooperate, only if the cooperation give some advantages to the players themselves. When the game reaches the situation where all players maximize their profit, the *Equilibrium point* was set. In the *Non-cooperative* game the *equilibrium point* is the *Nash Equilibrium*. Instead, in the cooperative game, the success of the group or coalition is the goal of each player and it comes before its own.

2. *Zero-Sum or Non-Zero-Sum* In a zero-sum game, the sum of loss and gain of all players are always 0. In particular, if some player gains something another player loss the same quantity. Instead, in the Non-zero-sum game, the sum is not constant and equal to 0. If some player losses something it isn't recovered from the others.

3. *Static and Dynamic games* In a *static games* we don't have changes during the time. Due to this solution, the moves of all players are taken *at the beginning*. All the players analyze the scenario, chose the best path to follow and play. Instead, when we play a *dynamic game* each player takes decision *sequentially*. For each instance, the players recalculate their paths. In this solution we need to indicate how information each player can have about the current and previous state of other players.

4. *Perfect information game* In this kind of game, each player knows exactly the action of the other players. Moreover, the player knows the previous, current and future state of the game.

5. *Finite game* The game has a prefixed number of stages and players, i.e. the actions of players are a limited number.

3.5 Nash equilibrium

Nash develops a theory that is in contradistinction with the Von Neumann and Morgenstein theory [19]. In the Nash theory, we have absence of dependence between player. The concept behind this theory is the *equilibrium point*. This equilibrium is a game strategy that allows to player to find a a trade-off between all self interest. I.e a strategy respects the Nash Equilibrium if there aren't players that can do better their goal changing strategies. Each player thinks that the other players will use the strategy chosen and if it can receives an advantage changing its own strategy.

The response, that correspond to a Nash equilibrium, is: each player prefers to don't switch its strategy. The Nash equilibrium is the best response in that situation and in that conditions.

Sometimes, the Nash equilibrium may appear irrational from an external point of view and it is normal because it is not necessary *Pareto optimal* [20].

From the mathematical point of view, *Nash Equilibrium* is:

$$s_i^{j*} = (s_1^{j*}, ..., s_N^{j*}), \quad N \in \mathbb{R} \tag{3.1}$$

where i indicates the players and j the stage of the game.

If the following N inequalities are verified, then the N strategies satisfy the Nash equilibrium

$$\begin{gathered}
J_1(s_1^{j^*}, s_2^{j^*}, ..., s_N^{j^*}) \leq J_1(s_1^{j}, s_2^{j^*}, ..., s_N^{j^*}) \\
J_2(s_1^{j^*}, s_2^{j^*}, ..., s_N^{j^*}) \leq J_2(s_1^{j^*}, s_2^{j}, ..., s_N^{j^*}) \\
\vdots \\
J_N(s_1^{j^*}, s_2^{j^*}, ..., s_N^{j^*}) \leq J_N(s_1^{j^*}, s_2^{j^*}, ..., s_N^{j})
\end{gathered}$$

where J_i elements are the cost function related to *i-th* player.

3.6 Examples of Game Theory

The game theory is used to analyzed some *"games"*. These games are explained in the below chapter.

3.6.1 The Prisoner's Dilemma

This example is the most famous analyzed by the game theory. Here we have two prisoners arrested for the same crime. The prosecutor doesn't have any evidence able to convict the two criminals to confess.

In order to reach the confession, the prisoners are isolated alone. Then the officials question each prisoner. The two prisoners haven't a way to communicate.

Each prisoner receives four deals:

1. Both prisoners confess: each prisoner receives five years prison

2. Prisoner 1 confesses, prisoner 2 not: prisoner 1 receives three years, prisoner 2 instead nine years.

3. Prisoner 2 confesses, prisoner 1 not: prisoner 1 receives ten years, prisoner 2 instead two year.

4. Either of them confess: both receive two years.

The most favorable strategy is to avoid confession. But, the two prisoners haven't any idea of the other's strategy. For this reason the most likely solution is that the two prisoners confess. The *Nash Equilibrium* suggests that the player do the best for them individually, but the worst solution for them collectively.

3.6.2 Dictator Game

In this game, we have two players. The player A must decide how to divide a cash price with the other player. The player A takes this decision without interference from player B. The result of people behaviour is:

1. 50% of people keep all the price to themselves.
2. 45% of people divide the price but the player B receives a smaller quantity.
3. 5% of people divide the price equally.

3.6.3 Volunteer's Dilemma

In this game, someone must make a job for the common good. The worst possible result is that nobody decides to do that.

For example, we have a rampant fraud in a company. Some junior employees know it, but don't tell it to top management because they could be fired. Also, being labeled as a spy may also have repercussions.

If nobody volunteers, the fraud leads the company to the bankruptcy and everyone loss their jobs.

3.6.4 The Centipede Game

This game is an extensive-form game in the game theory. Here we have two players and a pot.

The players have alternatively the chance to get the bigger part of a increasing money stash. If the player passes the stash, the latter can take the price increased slightly.

The game ends when one player takes the stash. The player that take the stash receives the bigger quantity and the other the smaller portion.

3.7 ROS environment

When programming a robot, two of the most complex aspects are the development and writing of a suitable code for robot motion planning and execution with people, object or other robots. In fact, an almost infinite numbers of configurations are available and a simple difference in the configuration of the robot could mean a completely different software inside and, as a consequence, a different behavior in the space [10]. To reach this goal, many frameworks have been crated by researchers all over the world, in order to manage the complexity of robots and to make easier their prototyping. The final result is an environment where you can manage the robot.

3.7.1 Design Goals

The Robotic Operating System (ROS) is only one of the frameworks available for robotics. ROS was developed with the diffusion of service robots that would be diffused on a large scale; for this reason it has a very general architecture that can be suitable for different conditions.

The goals of ROS are:

1. *Peer-To-Peer*: the ROS idea is a universe of different hosts connected together thanks to a peer-to-peer topology. ROS uses this solution due to the heterogeneous type of members. In fact, a framework based on a central server can realize the same things, but isn't able to manage different type of robots.

2. *Multi-lingual*: nowadays we have many types of languages, that can be used to write a code. Each programmer uses the language that he prefers. We can find many different reasons behind the chosen language, such as the syntax, easy-debugging, the run-time efficiency. In order to reach the widest number of users, the ROS was developed to be independent, for this reason it is defined as language-neutral. ROS fully support 4 programming languages: C++, Python, Octave and LISP

3. *Tools-based*: ROS design is based on a micro-kernel that helps us manage ROS complexity. Thanks to micro-kernel we have a lot of tools that can be used to build and run different ROS components. The micro-kernel is more versatile than a monolithic Kernel.

4. *Thin*: ROS has a modular structure. The ROS allows us to use standalone libraries without dependencies. In this way, we can write simple codes, using a lot of already available libraries, decreasing the complexity of code development. This solution makes code debugging easier. In fact, we can test only the portion that create issue. With this solution, we don't need to write a code only to manage different functions of common libraries.

5. *Free and Open-Source*: the developer of ROS decided that it must be publicly available. This solution is very smart, because it makes the debug of all level software easier. This is very important and convenient when we have many hardware and software developed and analyzed in parallel. ROS uses also the BSD license. This kind of license allows users to develop both commercial and no-commercial projects.

3.7.2 RVIZ

We use *RVIZ* a powerful tool of a ROS. This tool is a graphic visualizer developed for the ROS. RVIZ mean, in fact, ROS Visualizer. Thanks to this useful tool, we can show the result of the code developed. It is used to draw trajectories and costmap of our test. To do that, we implement inside the code, the instructions to start and use RVIZ.

3.8 ROS Concepts

3.8.1 Master

Master has the role to provide the names and to register all the nodes inside the ROS system. It checks which are publishers or subscribers to specific services or topics. Master permits to nodes to locate themselves inside the ROS system. Once that the nodes are located, they can communicate each other.

In the following figure we show the working process of master. In this example we have two nodes, a View-image node and a Camera node. At the beginning, camera node notifies to master that it wants to acquire an image. The topic used is called *"images"*.

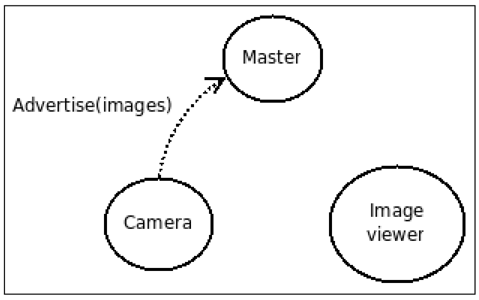


Figure 3.2. Images available.

The Camera node publishes the *"images"*, but we haven't another node that subscribes that topic. The image is still blocked. Then the View-image node asks to master if there are images available.

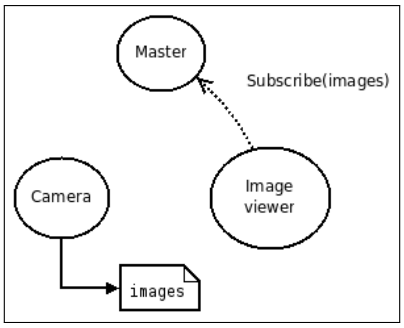


Figure 3.3. Images published.

Now, the topic *"images"* has either publisher or subscriber. The master manages the two nodes and the image transfer starts.

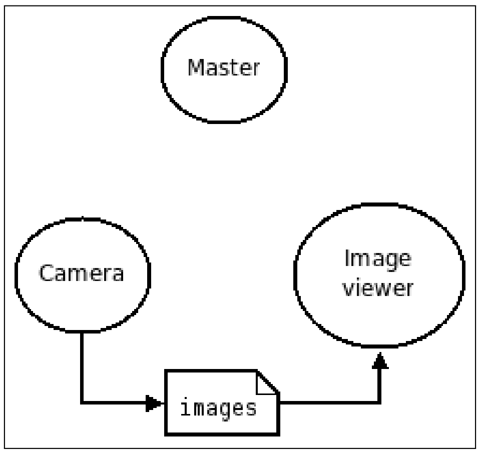


Figure 3.4. Images acquired.

3.8.2 Nodes

A *Node* in ROS is a process that develops computation. ROS uses many node together, in order to combined them together. Once that the nodes are linked in a graph, they use topics, RPC services and Parameter server to communicate each other.

Usually, a robot control system has many nodes inside. Each node has some specific role and interact with the other to reaches the goal. For example, one node controls the drive wheels, one node manages localization, one node creates path planning and so on.

The ROS system, based on the idea of nodes, creates an affordable system. In fact, if some node failed, the issue remains limited to that node. This system has a complexity reduced respect to a monolithic system.

All nodes are represented inside a graph. In this graph, the node is called with a *Graph Resource Name*, a unique name that identifies that node. Also, each node has a *Node Type* according with the function developed on it.

The nodes are written using specific libraries of ROS, called *roscpp* and *rospy*, according with the language chosen to write the node.

3.8.3 Topics

Topics are the virtual buses where the Nodes send and receive messages. Topics have a specific semantics in order to publish and subscribe data. Thank to these semantics, the Topics are able to distinguish the production of information from their destruction.

Usually, nodes don't know from which the information are coming or which other node tries to communicate with them. For this reason, the nodes, that want to know which other node sends the information, *subscribe* to the pertinent topic. In the opposite direction, the node *publish* to the pertinent topic. Obviously, we can have more than one node that subscribe or publish a topic.

Topics represent a unidirectional communication. If we need that nodes perform a remote call, we need to use a *service.*

3.8.4 Services

Services are used when we need to respond to RPC request/reply action. In this situation, the services send a couple of messages, one for the question and one for the answer. This utility is more efficient than the publish and subscribe model that we see before with Topics.

The services use a supplier ROS node that offers a string name. The client node calls the service sending a messages and waiting the response.

Services use srv files to instantiate themselves. These files are compiled inside the code thanks to a ROS library.

3.8.5 Messages

The nodes communicate each other using messages. These messages are published inside the topics.

Messages are simple data structure composed by specific fields. They supported standard primitive types and they can include array and structure as the struct in C.

Nodes can receive and send messages to answer to ROS service call.
Messages are sent in a *msg files.* These files are saved inside a *msg subdirectory* inside the packages.

Chapter 4

Code Development

In this chapter the code developed and implemented is explained in details. Specifically, the code consist in two main elements: the core code and the edge one.

The *core* code is the main element, because it implements the *socially-aware movement* for the robot. On the contrary, the *edge* software is used to interface the navigation environment with the core part of the code and to create a dynamic flow of data, able to reproduce a real-time scenario where the robot behaves in a human-like manner.

4.1 Cost Function solution

This thesis is based on a previous work, developed by Giada Galati [1] where she found and validated an algorithm that was able to move robot in a human-like manner.

The algorithm is based on the implementation of a cost function. This cost function is used to select the successive poses used by the robot. Thanks to these poses, the robot is able to move itself in a human-like manner.

For this reason, each point that player can reach has a cost. This cost is computed following the below formula 4.1:

$$\begin{aligned} Cost =& FatherCost + CostmapCost * WeightCostmap + \\ &+ NewPositionCost * PenalityAngle * WeightPosition \end{aligned} \tag{4.1}$$

Each formula's member has a specific role in the whole computation of the cost. Now we will explain the meaning fo each parameter:

1. *FatherCost*: the new point, also called daughter point, is generated from a previous one, called father point. The father point is obtained at previous time. This value is used to give at the daughter point the cost of the father position. In this way, at the end of the computation, the cost of last point is equal to the sum of all previous. I.e. it is the cost of the path.

2. *CostmapCost*: This value is read from the costmap loaded at the beginning. This value represents how much cost this position on that map. In our case if the place is free, we don't have a cost. Instead, if we have an obstacle or we are near of it, the cost is different.

3. *WeightCostmap*: we insert this multiplicative factor to tune in the right way the *CostmapCost*.

4. *NewPositionCost*: This cost is dependent from the position itself. In fact, it depends on the distance between the positions of this point and the goal. The algorithm measures the distance between these two points, and convert it in a value. In this way, a point nearer to the goal cost less than another furthest.

5. *PenalityAngle*: This multiplicative factor is inserted to increase the cost of those path that deflects from the straight. This solution is chosen to introduce the human-like movement. In fact, humans prefer to move straight on and this factor works exactly to introduce a penalty if the robot chooses to move differently. Also, this weight increases accordingly with the angle amplitude.

6. *WeightPosition*: we insert this multiplicative factor to tune in the right way the *NewPositionCost*.

4.2 Code

Now, we try to explain all of features of the two main parts of code.

1. *Core Code*

   The Core Code is the most important part of the software. It Implements all the assumptions and the logic previously described to create a socially-aware movement. Here, we create a code that develops a motion for the robot, acceptable for the human being.

   In order to compute the optimal and a socially-aware motion, we calculate all the possible paths that all players of the game will follow. The goal is to find the optimal paths with the lower cost. After this step, the code checks if we have intersections between the computed by all players. If not, the code controls if the other players change their paths, knowing the new path of the player analyzed. This process goes on until the paths chosen of all player remain the same for two consecutive iterations. Once that the code reaches this status, it means that we have the *Nash Equilibrium*. The *Nash Equilibrium* between players is the goal of the core part of the code.

2. *Edge Code*

   The Edge Code is the software developed around the Core Code. It has the role to make more dynamic the internal code. It reads the values of players positions from a file. This file can be generate by the sensor of robot or, like in our case, analyzing a video surveillance. Once that the code has these information, it imports the data regarding the robot. The information used are:

$$RobotData = \sum_{n=0}^{numRobot} Robot[X(t), Y(t), \Theta(t), t], \quad n \in \mathbb{R} \tag{4.2}$$

   Where:

   (a) *X(t)* is the x coordinate at that time instant.

   (b) *Y(t)* is the y coordinate at that time instant.

   (c) *θ(t)* is the angle at that time instant.

   (d) *t* is time instant.

   The time instant is used inside the code to manage all the features. The most important is the robot time instant, i.e. when the robot appear on the scenario. It establish the starting point of the software. Knowing the robot time instant, the code starts to analyze which players are present on scenario at that instant. Now, it send the information to *Core Code* that elaborates the paths and so on. The game ends when the robot goal is reached or if the time instant reaches the last value stored in the data coming from the video surveillance .

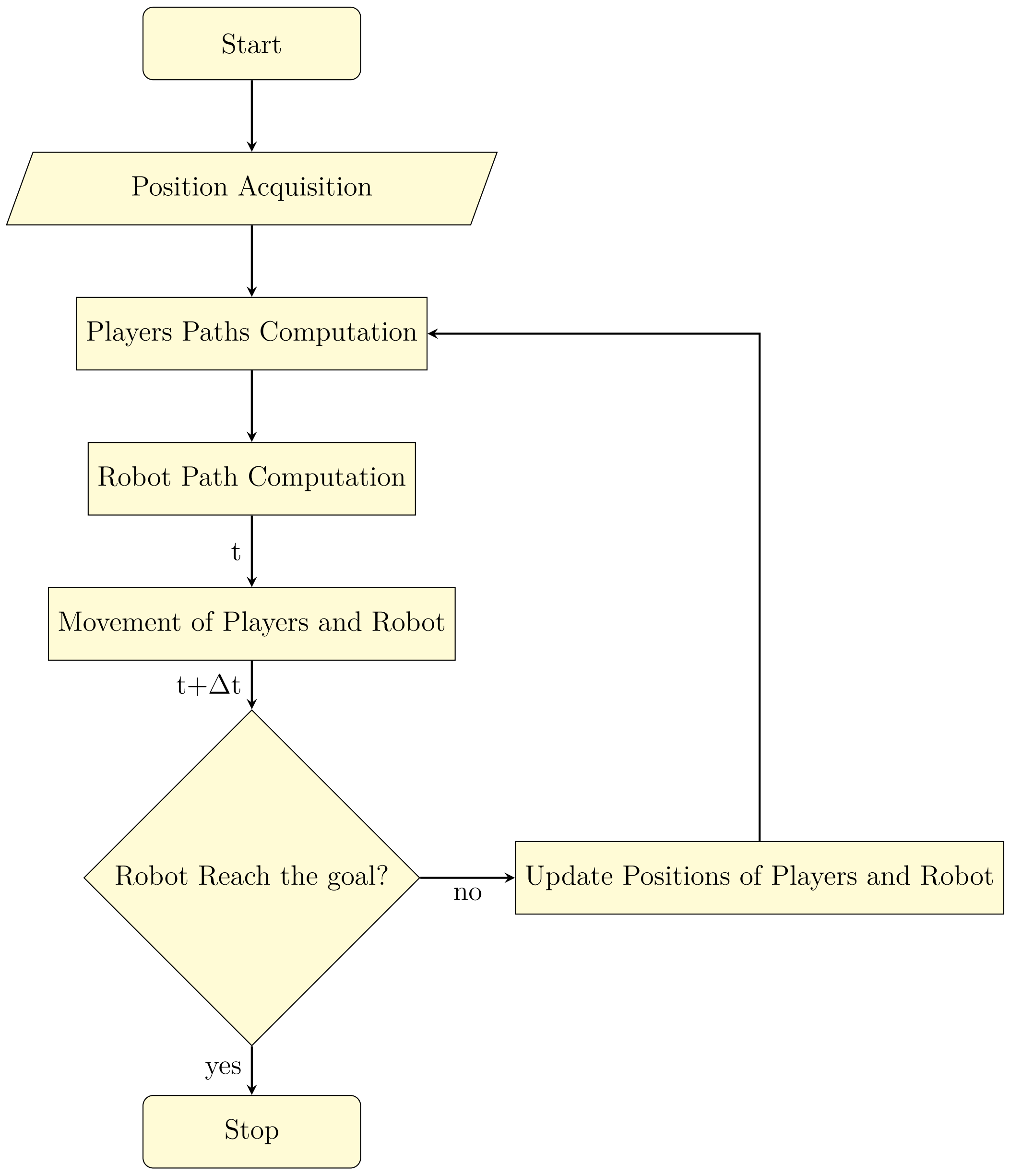
   Now we can represent the paths using specific tool of ROS. In particular, we use *RVIZ*, a visualizer of ROS, to see the path in 3D vision. In this way we can appreciate the result of the software.

RVIZ uploads every time the new poses of all present players. After that, the code increases the time instant by one and repeat all the process, until the *Core Code* return that robot reached the goal. If it receives that information, it stop the process and the code ends.

4.3 Flow Chart

The thesis work follows a specific development. To better understand how the code works, we create a flow chart. The following flow chart explains the developing process of the thesis.

Each block will be explain in the following pages.


Start
Position Acquisition
Players Paths Computation
Robot Path Computation
t
Movement of Players and Robot
t+Δt
Robot Reach the goal?
no
Update Positions of Players and Robot
yes
Stop

4.3.1 Pose Acquisition

In this stage, the code acquires information from the external environment. It finds the pose of each player in the space and the relative pose of the robot.

This data comes as result of process, that analyses environment with camera and sensors present on the robot, or from an external database. In our case, we use the data coming from a surveillance video .

These data are read by the code and stored inside it, in order to be elaborate in the future processes. These data include: the pose of each player, x and y coordinates, time instant and angle orientation. These information will be used by the robot to move itself in the correct way.

In our case the data are saved in a file that software reads at the beginning. In this way the code knows how many players are present on the scenario. Also, it knows where players will be and when them start to play.

The code creates C structures where information are stored and it tunes the process to manage each player.

4.3.2 Players Paths Computation

From the data acquired in the previous stage, we check which players are present in that instant on the scenario.

The algorithm uses the positions at the time *t and t-* Δt and the orientation to compute the paths of each player. Knowing this data, the code estimates the future poses of each player. The code use the hypothesis that all the players go straight on. In this way, the code computes the future pose from *t* until *t+3* time instant.

Using this information, the code compute three successive fictitious steps. In this way, for each player we have a path composed by four steps, one known and three fictitious.

4.3.3 Robot Path Computation

The computation of the path followed by the robot differs from the estimation of the path of other players. In fact, we know exactly the position of robot, X and Y coordinate, and the coordinate of goal. We impose that the velocity of the robot is constant, in order to simulate a movement similar to an human being. I.e. the steps of robot path are constant. With this assumption, the robot computes four steps in order to reach the goal.

Number of steps

The robot computes how many steps it needs to reach the goal. To do that, it computes the distance between robot position and goal and divides it for the step wide.

$$NumSteps = \frac{GoalPosition - RobotPosition}{LengthStep} \tag{4.3}$$

Where:

1. *NumSteps* is the number of steps that robot needs to reach the goal.
2. *GoalPosition* is the absolute position of the robot goal.
3. *RobotPosition* is the current position of robot.
4. *LengthStep* is the wide of step that robot can perform.

The code uploads the result of the previous equation and shows it. After that, it computes again how many steps must be created to reach the goal. We can find 4 different situations:

1. $$NumSteps > 4 \tag{4.4}$$
   The code computes 4 steps and the Robot still doesn't reach the goal.
2. $$NumSteps = 4 \tag{4.5}$$
   The code computes 4 steps and the Robot reaches exactly the goal.
3. $$NumSteps < 4 \tag{4.6}$$
   The code rounds down the number of steps and uploads the numbers of steps computed, according with the previous result.
4. $$NumSteps < 1 \tag{4.7}$$
   The code doesn't compute a further step because the robot is too close to the goal. The distance between robot and goal is less then a step and a further movement can ruined the process.

Number of choices

Now the code elaborates the poses that the robot can occupy during the movement. For each step, the robot can chose 7 options in order to reproduce the real movement of an human being.

The choices are the following:

1. Go straight with a *Wp1* as weighting factor.
2. Turn left by 30° with a *Wp2* as weighting factor.
3. Turn left by 60° with a *Wp3* as weighting factor.
4. Turn left by 90° with a *Wp4* as weighting factor.
5. Turn right by 30° with a *Wp5* as weighting factor.
6. Turn right by 60° with a *Wp6* as weighting factor.
7. Turn right by 90° with a *Wp7* as weighting factor.

Each choice has a different cost, according with the *weighting factor*. The cost depends on how much the movement can be unusual for a human being. In fact, we prefer to move straight or in a smooth manner. If we need to avoid a player or an obstacle, we prefer to use a low steering angle and plan it with some advance. For this reason the cost of a direction change is lower when the angle is smaller.

The following figure shows the possible choices with the related weighting factors.

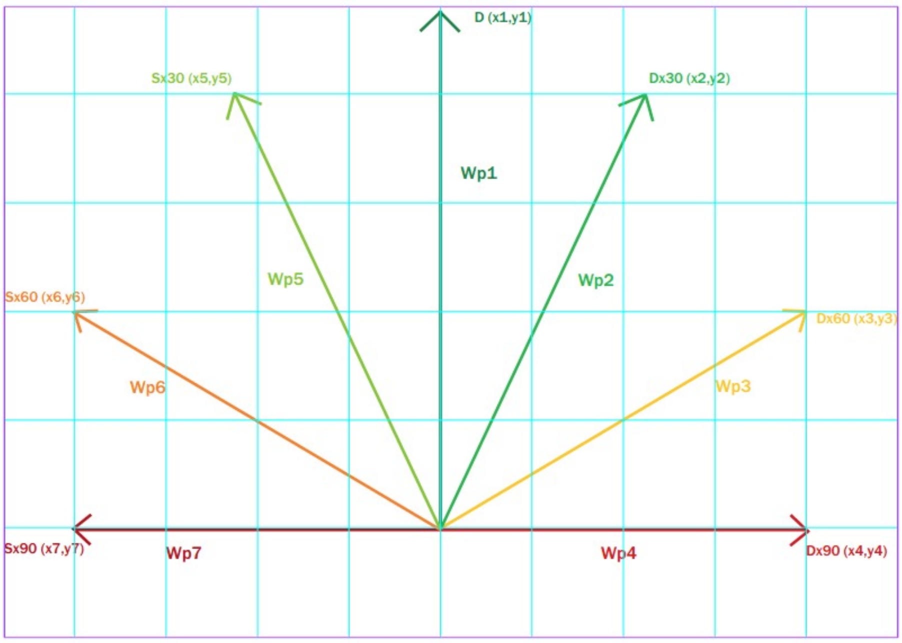


Figure 4.1. Choices with the weighting factor

Number of possible position

Now, the algorithm computes each positions that player can occupy in the next time instant. To do that the code use the following equation:

$$NumPosition = \sum_{n=0}^{numSteps+1} Choices^n, \quad n \in \mathbb{R} \tag{4.8}$$

Thanks to this equation we have all the possible position that player can use. The members of equation are:

1. *NumPosition*: This number represents all the position that player can occupy.
2. *numSteps*: the number of steps that each player will do accordingly with the numbers that algorithm found before.
3. *Choices*: the number of choices that each player has to move in the scenario. In our case the number is equal to 7.

Now the robot computes a large number of possible paths, computing the effective cost of each path. The algorithm searches for the path with the minimum cost. Now it checks this path with the players paths and verifies if there is a collision with trajectories of other players. If the algorithm doesn't find intersection, it chooses this path, instead if it finds an intersection, it labels the path as *not usable* and checks the other path accordingly with increasing cost and repeats it until it finds an usable one. When we have the path, the algorithm returns the pose that robot will occupy.

4.3.4 Movement of Players and Robot

Once that all the paths are available, the code decides how to move the robot. Here we apply the *Nash Equilibrium.*

The code finds the optimal paths for all the players and robot. It continues until all the players don't change their paths for at least two iterations. After that, the poses of all characters are available. The code sends to the *RVIZ*, a tool of ROS, the poses that members will occupy at the next step. Then we can see how the movement evolve.

After this process, we see that the paths of all players available and robot increase by one step. Then the code ends this phase and moves on.

4.3.5 Robot reach the goal or not

Now the code checks if the robot, with the path increased, reached the goal or not. To do that, the code controls if the distance between goal and robot is more or less than a fixed step.

The step was always the same computed before. After this check, we can have two responses:

1. Yes, the robot reached the goal. It means that the distance between goal and robot is *less* than a step and the code ends with success.
2. No, the robot didn't yet reach the goal. It means that the distance between goal and robot is *more* than the fixed step and the code continues to elaborate.

4.3.6 Update Positions of Players and Robot

This stage is reachable only if the robot didn't reach the goal. Now it is necessary to update the data used by the algorithm, using the data coming from the result of the previous step. In fact, now the algorithm needs to upload the old starting positions, because robot and players move.

The new poses are saved as a new initial pose, useful for the further iterations. Now the code has the new, players and robots, positions. With these updated data, the code could manage again the process without any particular issue.

Chapter 5

Validation and Test

The proposed algorithm was tested evaluating the *Core Code* individually and the whole proposed framework with also the so-called *Edge Code*. With the same idea, the test have been done following the same process.

Firstly, we show the results of the test performed with only the *Core Code*. At this stage, we adapt the code to manage 4 players to provide a simple simulation scenario. We want to verify if the algorithm is correctly implemented and if it works correctly. The turning point is the correct application of *Nash Equilibrium*. To check it, we include inside the code the possibility to enable and disable the *Nash Equilibrium*.

After this first test, we check the effectiveness of the code that elaborates external data input. Hence, we check if the input data received from external are read correctly by the code. In order to verify it, we test the algorithm with a group of 3 players and a robot, where the robot position can be defined arbitrary to test the code with different conditions.

5.1 Test 1: fictitious players

This test is useful to check the effectiveness of the main part of the algorithm.

A malfunction of the main part of the algorithm, could compromised all the subsequent developments. Moreover, this test is essential to find any problem and error, because could be difficult or impossible to find it after this step. To check the code properly, we perform two different kinds of test.

1. *Obstacle Avoidance*: we check if the player is able to avoid a fixed obstacle.
2. *Players Avoidance*: we check if the player succeed to avoid the other players, by applying the proposed method.

Now we will explain the examples and the results obtained.

5.1.1 Obstacle Avoidance

In this paragraph we report the first test. The player must avoid the obstacle that intersects his trajectory. To do that, we insert a customized costmap inside the code. The following image 5.1 shows the map used for the test.

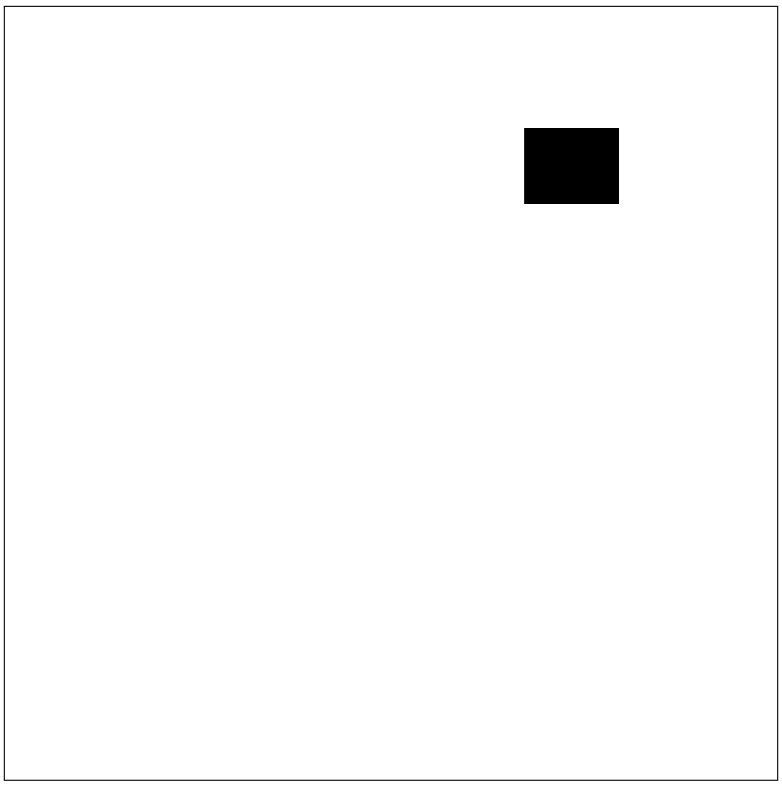


Figure 5.1. The Costmap created for the test.

This *costmap* is used by ROS to calculate the cost to insert to each possible position. The costmap returns a value depending on the spatial position occupies by the player. In particular:

1. If the player is on the free space, the costmap assigns a cost equal to zero.
2. If the position of player is near to the obstacle , we have a proportional cost depending on the distance between player and obstacle.
3. If the position of player is inside the obstacle, we have the maximum cost.

This cost can be assigned with a finite value, if we want that player takes in consideration the position or an infinite value if we want that player excludes it.

Figure 5.2 shows the test performed. We have a player that must avoid obstacle to reach the goal. To avoid it, the algorithm chooses a path reaching the goal and avoiding the obstacle. In particular, the player chooses the following poses:

1. *First movement*: the player goes right with 30° degree.
2. *Second movement*: the player goes right with 30° degree.
3. *Third movement*: the player goes straight on.
4. *Fourth movement*: the player goes left with 30° degree.

In this way, the player avoids the obstacle and goes as close as possible to the goal.

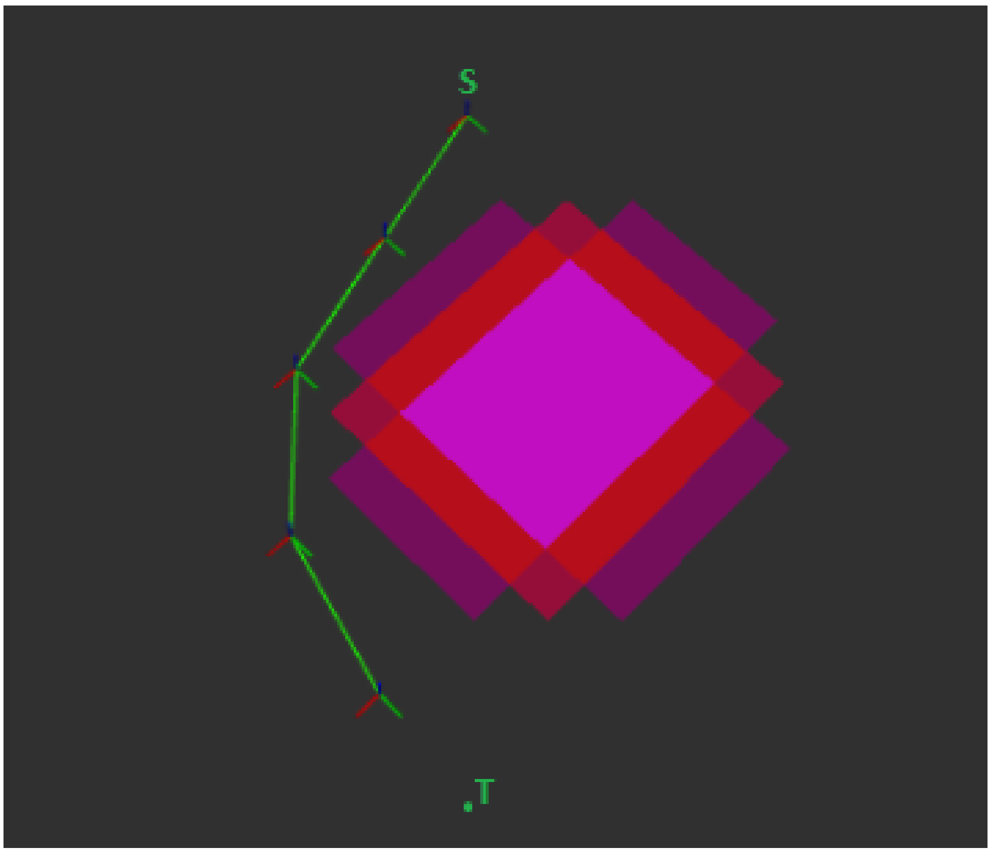


Figure 5.2. The path followed by the player to avoid the obstacle

To be sure that the code works properly, we perform another test with, this time, the *Robot* and other three players. In particular, we insert the robot goal position inside the obstacle, and we see how the code react to this challenging scenario. The Robot is placed in the position A(8,8) and the goal is in G(13,10) in the map reference frame.

The robot avoids the obstacle and tries to reach the goal. It stops its path near the goal position.

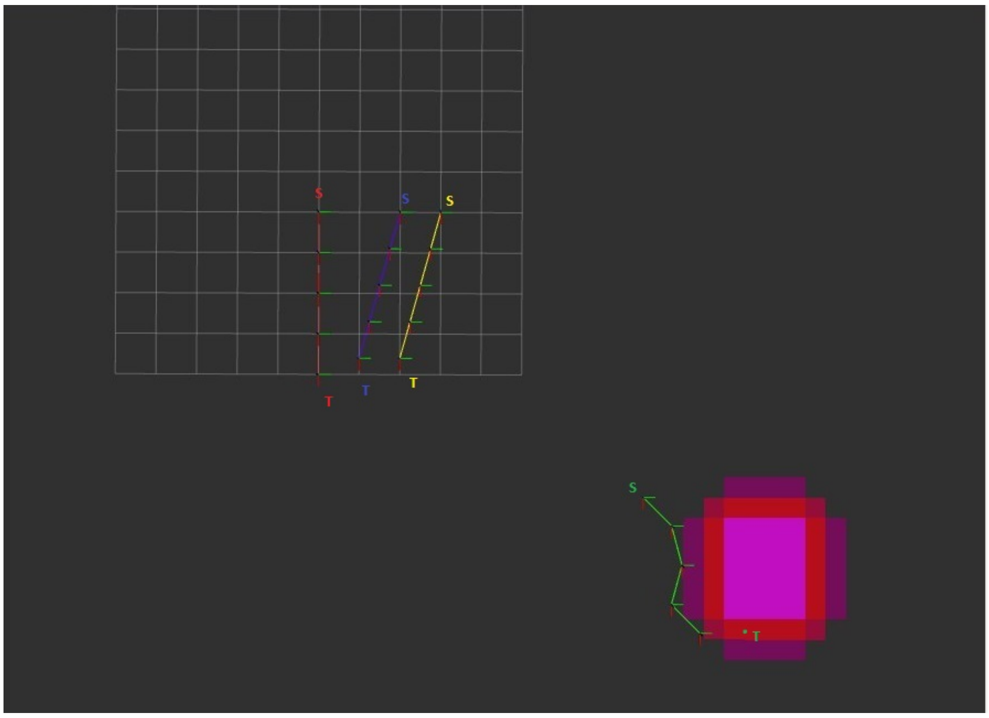


Figure 5.3. The path followed by the robot to avoid the obstacle

5.1.2 Player Avoidance

In this section, we perform tests to verify the ability of the algorithm to create paths without intersections with other players.

First Test

In the following figures, we have 3 players that have intersections. In figure 5.4 is reported a simple elaboration to avoid the collision between players without the implementation of *Nash Equilibrium*. Here we have:

1. *Player1*, in blue, steers on the right with 30° degree angle to avoid collision with *Player2*, and *Player3*, according to the hypotheses that all players go straight on.
2. *Player2*, in green, steers on the right with 30° degree angle to avoid collision with *Player1*, according to the hypotheses that all players go straight on.
3. *Player3*, in red, steers on the right with 30° degree angle to avoid collision with *Player1*, according to the hypotheses that all players go straight on.

This solution is the starting point for our algorithm. The solution found in this case is acceptable but it is not the optimal. In fact, all the players change their own direction, but in this way they waste time and increase the costs. Moreover, players reach the final poses with a certain distance from the original goal.

The two deviations and the distance from the goal increase the cost function.

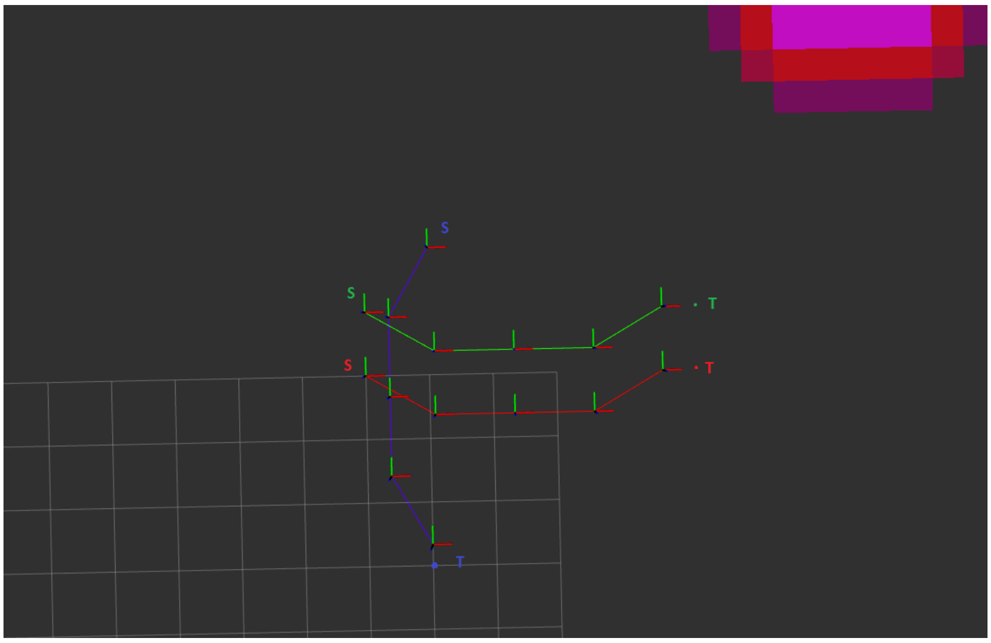


Figure 5.4. Players paths without Nash Equilibrium

Now, instead, we implemented in the code the *Nash Equilibrium*. On figure 5.5, we have the result of the whole algorithm. This setup goes in deep respect to the previous example. Once that the algorithm finds the previous solution, it calculates again the possible path. This time, it uses the result obtained in the previous computation, instead to use the hypotheses that all players go straight on. In this way, it finds the optimal solution. The iterations continue until for two subsequent results all players choose the same paths. The differences respect the test done before are:

1. *Player1*, in blue, goes straight on
2. *Player2*, in green, steers on the right with 30° degree angle to avoid collision with *Player1*, due to the decision of *Player1* to go straight on.
3. *Player3*, in red, goes straight on.

Now the solution is optimal, because only one player decided to change its path in order to avoid collisions. In this way, the cost functions of other players remain the lowest possible.

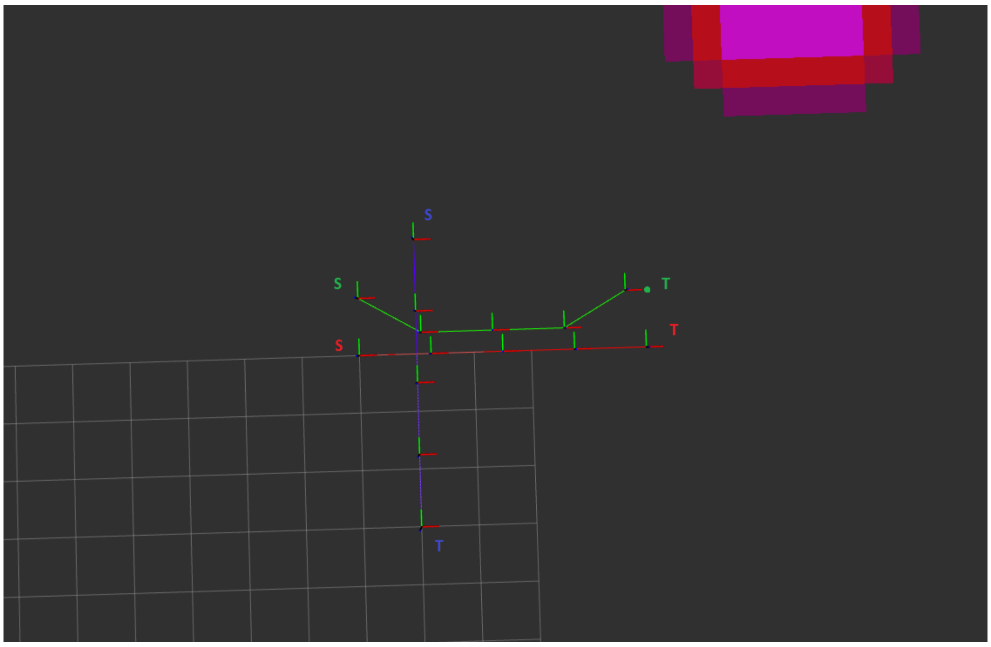


Figure 5.5. Players paths with Nash Equilibrium

Second Test

Another test is performed to verify the avoidance between players. In this test the robot intersects only one player. The other two players are inserted to see how the algorithm manages it. In fact, these two players have free path and the algorithm, if it works correctly, should send it to goal directly. Figure 5.6 shows the obtained results.

1. *Robot*, in green, steers on the right with 60° degree angle to avoid collision with *Player1*, according to the hypotheses that *Player1* goes straight. After this movement, it steers to the left with 30° degree angle to go toward the goal in position G(4,3).

2. *Player1*, in yellow, goes straight on from A0(1,1) to B1(4,4). The player doesn't make deviation because the robot already avoids it.

3. *Player2*, in blue, goes straight on from B0(1,0) to B1(4,0).

4. *Player3*, in red, goes straight on from C0(0,1) to C1(0,4).

This solution is found without the implementation of Nash game.

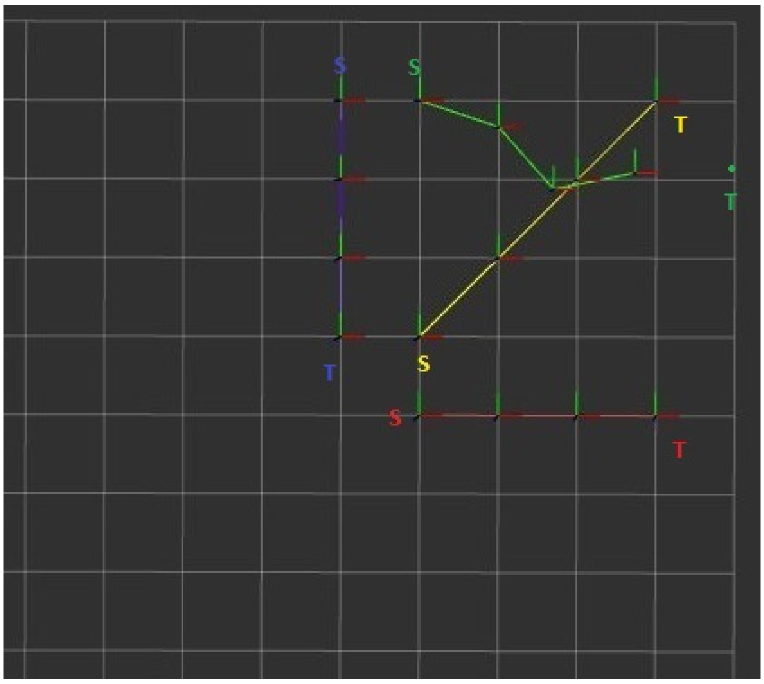


Figure 5.6. Second Test without Nash Equilibrium

Now, we repeat the same test but enabling the Nash game, in order to reach the *Nash Equilibrium*. The following figure 5.7 shows the result obtained.

In this scenario, the result is different. In fact, we have a different interaction between the *Robot* and the *Player1*. Now we have the solution that satisfies the *Nash Equilibrium* and has the lowest cost function.

1. *Robot*, in green, steers on the left with 30° degree angle to avoid collision with *Player1*.
2. *Player1*, in yellow, goes straight on from A0(1,1) to B1(4,4). The player turns right by 30° degree angle to avoid the *Robot* path.
3. *Player2*, in blue, remains the same: it goes straight on from B0(1,0) to B1(4,0).
4. *Player3*, in red, remains the same: it goes straight on from C0(0,1) to C1(0,4).

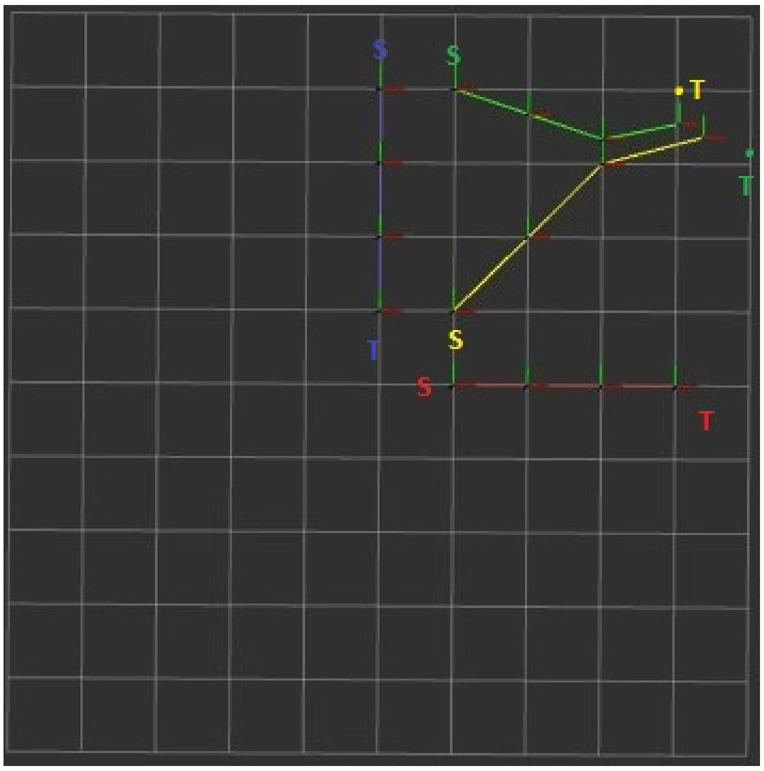


Figure 5.7. Second Test with Nash Equilibrium

Third Test

A third test is done to check how the algorithm reacts to a robot that tries to cut the path of others two players. The starting point of *Robot*, *Player1* and *Player2* are set in order to have an intersection in two different instants. Moreover, here we insert 5 poses in order to increase the computational cost.

The following figure 5.8 shows the result obtained with the algorithm without *Nash Equilibrium*.

1. *Robot*, in green, steers on the right with 60° degree angle to avoid collision with *Player1*. After that, it turns left with 30° angle to avoid the *Player2*. Then it goes straight and at the end it turns left with 30° degree angle to get close to the goal.

2. *Player1*, in yellow, goes straight on from A0(1,1.3) to A1(4.6,2).

3. *Player2*, in blue, goes straight on from B0(1,2) to B1(4.6,1).

4. *Player3*, in red, goes straight on from C0(1,0) to C1(5,0).

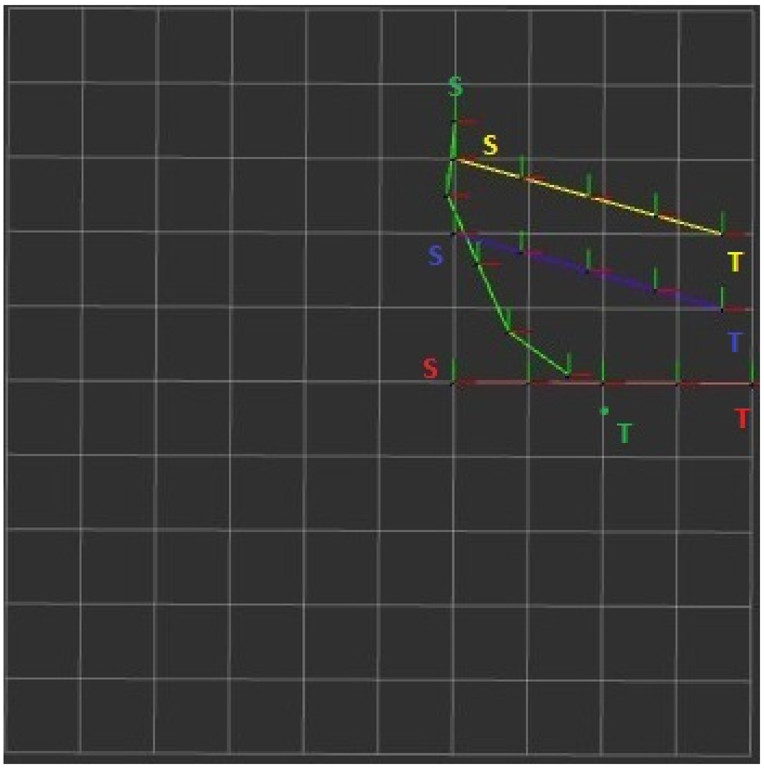


Figure 5.8. Third Test without Nash Equilibrium

In the following figure 5.9 we apply the concept of the *Nash Equilibrium* and see the result.

Here, we can observe that the result of algorithm doesn't change. In fact, also in this case, the Robot chooses to avoid the player using the same path. This mean that the *Nash Equilibrium* is already reached in the previous case. This could be possible if the assumption, that all players go straight on, results also the *optimal* solution for *Nash game*.

This optimal solution doesn't change with the implementation of *Nash game* because the others solutions aren't advantageous respect to the already found.

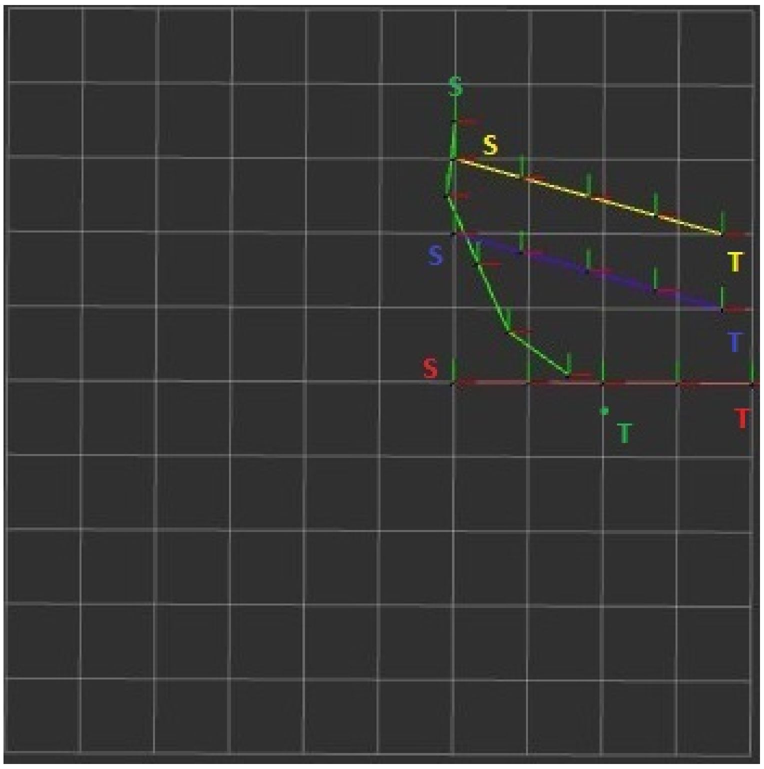


Figure 5.9. Third Test with Nash Equilibrium

5.1.3 Test result

These test have been made to verify the correctness and effectiveness of the algorithm. Moreover, we perform different kinds of test to check if the algorithm is stable and to find always a feasible result.

First of all, the main goal is to check the ability to avoid collisions between players. we find that the algorithm is able to do that and to define the correct poses to players and robot.

Also, the algorithm is able to avoid an obstacle inserted with a cost-map. During the test done, the algorithm computes the correct poses to avoid the obstacle. The robot, following these data, reaches a position near the goal without collides with the obstacle.

5.2 Test 2: real-world surveillance video

This test has been done to demonstrate the effectiveness of the algorithm implemented on a real world scenario. To do that, we use a video used during the testing phase on software of this kind. The video is elaborated and the data of players are extrapolated in a file. In this way, we can send it to the algorithm and manage it. Moreover, we design the costmap that represents the scenario of video to be used by the algorithm.

5.2.1 Video

Figure 5.10 shows the video used. This video is chosen to evaluate many algorithm due to its completeness and useful properties. In fact, we have pedestrians, groups of them and also fixed obstacles.

The algorithm takes information from this video and elaborates it. The fixed obstacles are loaded into the costmap. Instead, the poses of pedestrians are elaborated and extrapolated in a input file read by the code and, then, managed as players of the game theoric approach.

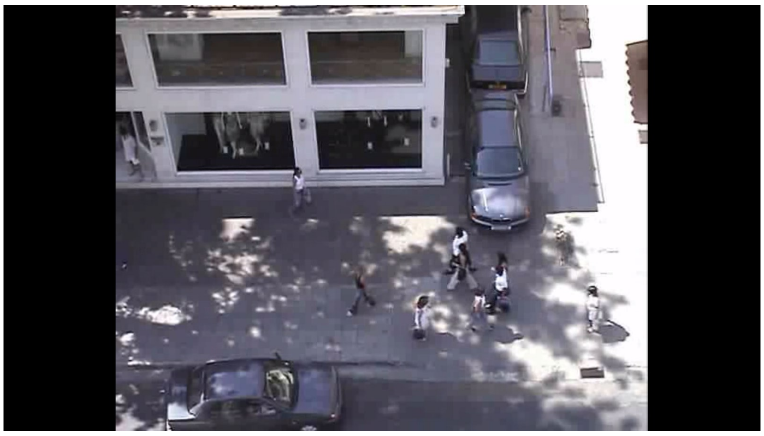


Figure 5.10. Example of scenario represented in the surveillance video

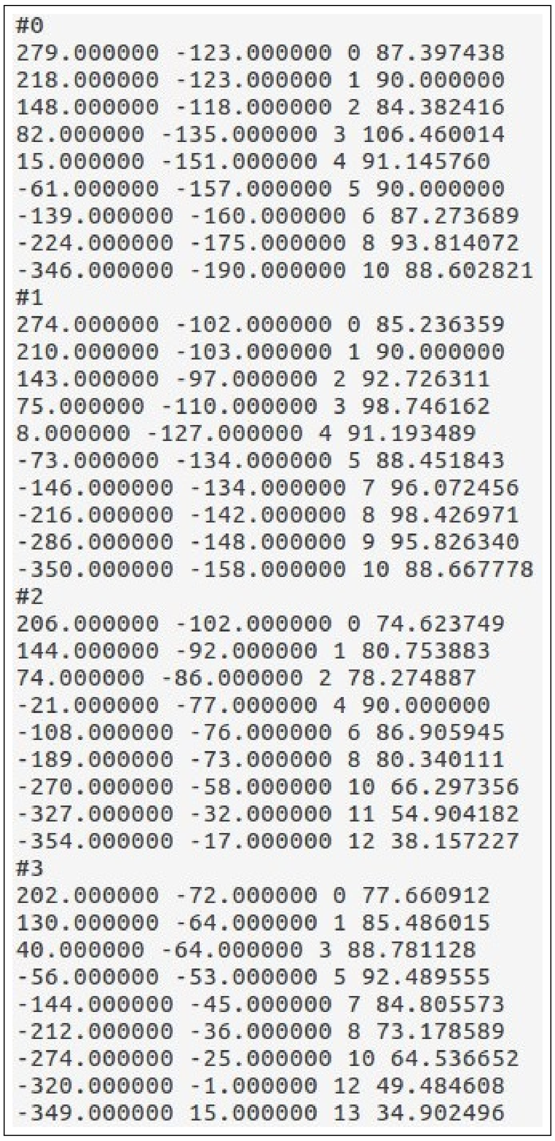
5.2.2 Extrapolated data from video

The video is analyzed. The pedestrians poses and, then, the paths are extracted and saved in a file. This file can be read by the algorithm. In particular, in this file we have groups of data. Each group is a spline curve that represent the path of that specific player in that period of time. Figure 5.11 shows an example of data saved inside the file. Specifically, we have the following information:

1. *Number of Spline*: The number after the # character indicates the sequence number of spline. The code elaborates this number as a player.

2. *First row*: it indicates the X coordinates.

3. *Second row*: it indicates the Y coordinates.
4. *Third row*: it indicates the time instant.
5. *Fourth row*: it indicates the orientation of the player.



```
#0
279.000000 -123.000000 0 87.397438
218.000000 -123.000000 1 90.000000
148.000000 -118.000000 2 84.382416
82.000000 -135.000000 3 106.460014
15.000000 -151.000000 4 91.145760
-61.000000 -157.000000 5 90.000000
-139.000000 -160.000000 6 87.273689
-224.000000 -175.000000 8 93.814072
-346.000000 -190.000000 10 88.602821
#1
274.000000 -102.000000 0 85.236359
210.000000 -103.000000 1 90.000000
143.000000 -97.000000 2 92.726311
75.000000 -110.000000 3 98.746162
8.000000 -127.000000 4 91.193489
-73.000000 -134.000000 5 88.451843
-146.000000 -134.000000 7 96.072456
-216.000000 -142.000000 8 98.426971
-286.000000 -148.000000 9 95.826340
-350.000000 -158.000000 10 88.667778
#2
206.000000 -102.000000 0 74.623749
144.000000 -92.000000 1 80.753883
74.000000 -86.000000 2 78.274887
-21.000000 -77.000000 4 90.000000
-108.000000 -76.000000 6 86.905945
-189.000000 -73.000000 8 80.340111
-270.000000 -58.000000 10 66.297356
-327.000000 -32.000000 11 54.904182
-354.000000 -17.000000 12 38.157227
#3
202.000000 -72.000000 0 77.660912
130.000000 -64.000000 1 85.486015
40.000000 -64.000000 3 88.781128
-56.000000 -53.000000 5 92.489555
-144.000000 -45.000000 7 84.805573
-212.000000 -36.000000 8 73.178589
-274.000000 -25.000000 10 64.536652
-320.000000 -1.000000 12 49.484608
-349.000000 15.000000 13 34.902496
```

Figure 5.11. Data extracted from video analysis

5.2.3 Costmap

The following figure 5.12 shows the costmap created from the video. This image was loaded inside the ROS code as costmap. In this way, we can avoid the obstacle simply reading the costmap. In fact, inside the cost function we have the value read from the costmap.

Each cell of costmap has a different value, depending on the presence or not of obstacle. This value is added to the cost function of each path. In this way, the player tries to find another path with the lowest cost and, then, avoiding the obstacle.

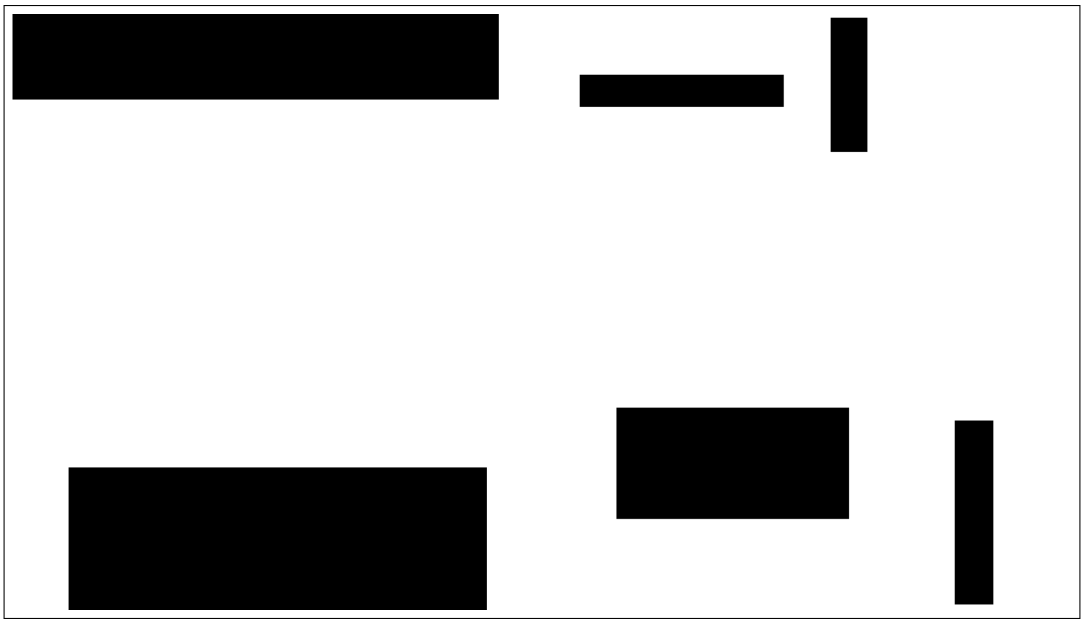


Figure 5.12. The Costmap used for the scenario with the surveillance video of Figure 5.10

5.2.4 Test

The test performed is explained in the following paragraphs.

In this test, we read the data of three spline from the file. In this way we have three players that will be displayed in the test. Also, we have the complete costmap. The following legenda helps us to understand the figures:

1. *Black spaces*: the free space that is present on the costmap
2. *Purple space*: the obstacles that are present on the costmap.
3. *Light purple space*: these areas, that surround the obstacles, are in the inflation radius. This technique to inflate the dimensions of obstacles is used to avoid obstacle maintaining a safety distance.
4. *Green line*: the robot path.
5. *Red line*: the first player path.
6. *Blue line*: the second player path.
7. *Yellow line*: the third player path.

We perform a simulation on the real world scenario. In the following pages we will explain each single step performed by the algorithm. Each step corresponds to a single iteration of the algorithm. We perform 8 sequential steps.

First Step

At the beginning, the robot moves towards the goal. During the computation the robot doesn't intersect any other player. For this reason, the compute path chosen by the robot goes straight.

The algorithm computes also the future positions of the other three players. In particular, it returns that, in the next steps we could have intersection between the second player and third. To avoid these possible collisions, we have to find a feasible solution. At the end, the algorithm finds that game reached the *Nash Equilibrium*.

Once that we have reached the *Nash equilibrium*, the algorithm publish the trajectory on the ROS Visualizator (RVIZ).

The robot position is obtained by the algorithm to obtain the *Nash equilibrium*. On the other hand, players positions, drawn in RVIZ, are not computed by the algorithm, but they follow the trajectory defined in the surveillance video. In this way, the algorithm computes how the robot and players can interact, but the players positions are always the real ones coming from the real-world scenario.

Figure 5.13 shows the result obtained in the first step.

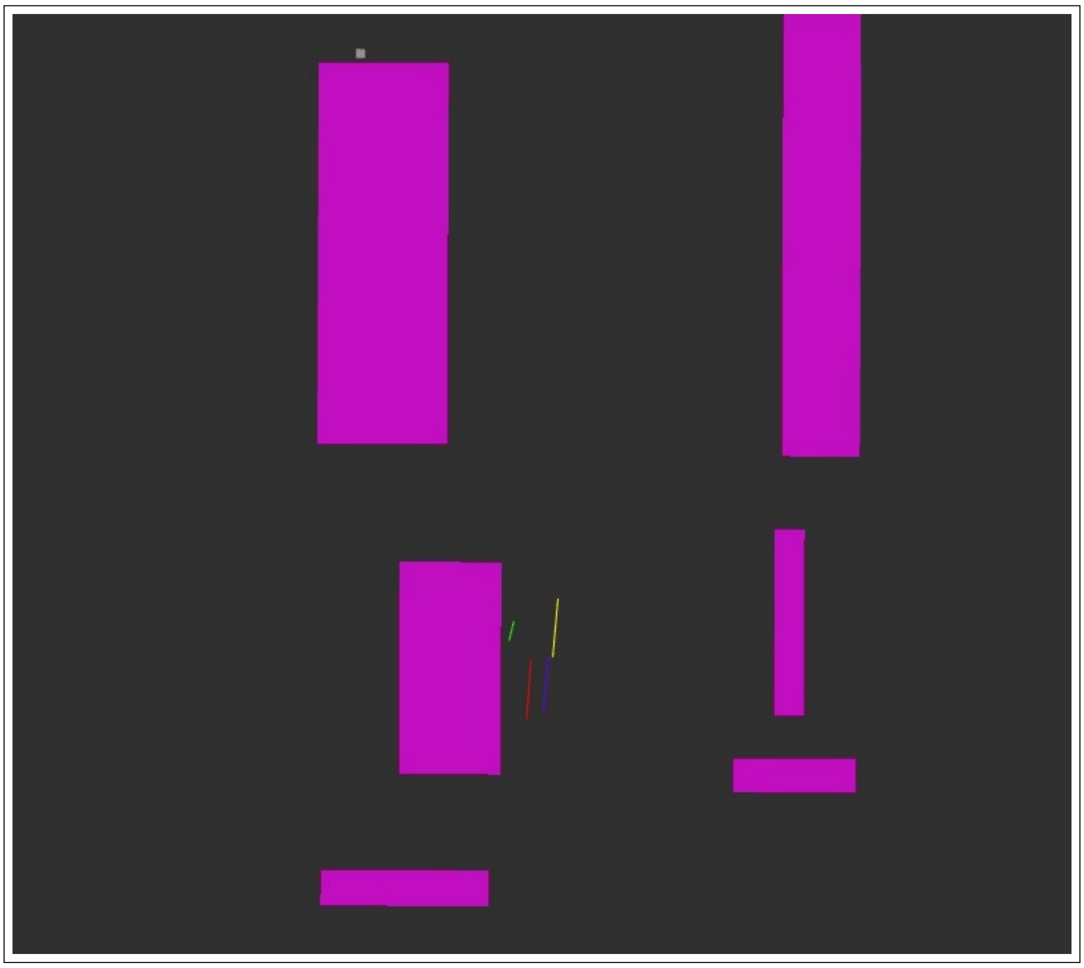
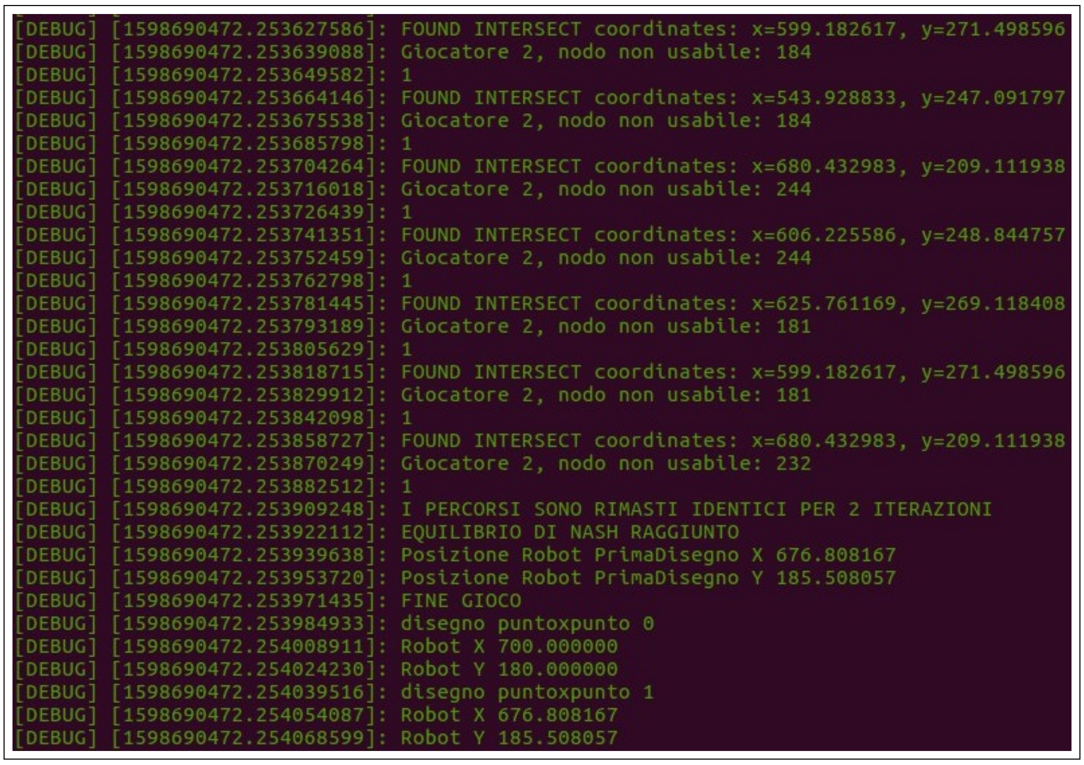


Figure 5.13. First step of movement

Instead, figure 5.14 shows the terminal during the execution of the algorithm to find the robot path. Only a few part the the stream on the terminal is reported.

Once that the algorithm excludes the path unusable, it checks if for two iterations all the players and robot choose the same paths. If yes, we have reached *Nash Equilibrium* and we can conclude the game and, then, draw the result on RVIZ.



```
[DEBUG] [1598690472.253627586]: FOUND INTERSECT coordinates: x=599.182617, y=271.498596
[DEBUG] [1598690472.253639088]: Giocatore 2, nodo non usabile: 184
[DEBUG] [1598690472.253649582]: 1
[DEBUG] [1598690472.253664146]: FOUND INTERSECT coordinates: x=543.928833, y=247.091797
[DEBUG] [1598690472.253675538]: Giocatore 2, nodo non usabile: 184
[DEBUG] [1598690472.253685798]: 1
[DEBUG] [1598690472.253704264]: FOUND INTERSECT coordinates: x=680.432983, y=209.111938
[DEBUG] [1598690472.253716018]: Giocatore 2, nodo non usabile: 244
[DEBUG] [1598690472.253726439]: 1
[DEBUG] [1598690472.253741351]: FOUND INTERSECT coordinates: x=606.225586, y=248.844757
[DEBUG] [1598690472.253752459]: Giocatore 2, nodo non usabile: 244
[DEBUG] [1598690472.253762798]: 1
[DEBUG] [1598690472.253781445]: FOUND INTERSECT coordinates: x=625.761169, y=269.118408
[DEBUG] [1598690472.253793189]: Giocatore 2, nodo non usabile: 181
[DEBUG] [1598690472.253805629]: 1
[DEBUG] [1598690472.253818715]: FOUND INTERSECT coordinates: x=599.182617, y=271.498596
[DEBUG] [1598690472.253829912]: Giocatore 2, nodo non usabile: 181
[DEBUG] [1598690472.253842098]: 1
[DEBUG] [1598690472.253858727]: FOUND INTERSECT coordinates: x=680.432983, y=209.111938
[DEBUG] [1598690472.253870249]: Giocatore 2, nodo non usabile: 232
[DEBUG] [1598690472.253882512]: 1
[DEBUG] [1598690472.253909248]: I PERCORSI SONO RIMASTI IDENTICI PER 2 ITERAZIONI
[DEBUG] [1598690472.253922112]: EQUILIBRIO DI NASH RAGGIUNTO
[DEBUG] [1598690472.253939638]: Posizione Robot PrimaDisegno X 676.808167
[DEBUG] [1598690472.253953720]: Posizione Robot PrimaDisegno Y 185.508057
[DEBUG] [1598690472.253971435]: FINE GIOCO
[DEBUG] [1598690472.253984933]: disegno puntoxpunto 0
[DEBUG] [1598690472.254008911]: Robot X 700.000000
[DEBUG] [1598690472.254024230]: Robot Y 180.000000
[DEBUG] [1598690472.254039516]: disegno puntoxpunto 1
[DEBUG] [1598690472.254054087]: Robot X 676.808167
[DEBUG] [1598690472.254068599]: Robot Y 185.508057
```

Figure 5.14. Stream of the execution of the algorithm during the first step

Second Step

The algorithm reloads the new poses of players, reading them from the input file, and the new poses of robot, obtained as result of last iteration.

In this second step, the robot intersects the path of first player.

The code finds a new path to avoid the collision. To do it, the code analyses new reachable poses, according to the increasing cost. At the end, if the game reached the *Nash Equilibrium*, the code shows the result in the RVIZ environment. This result is showed in figure 5.15.

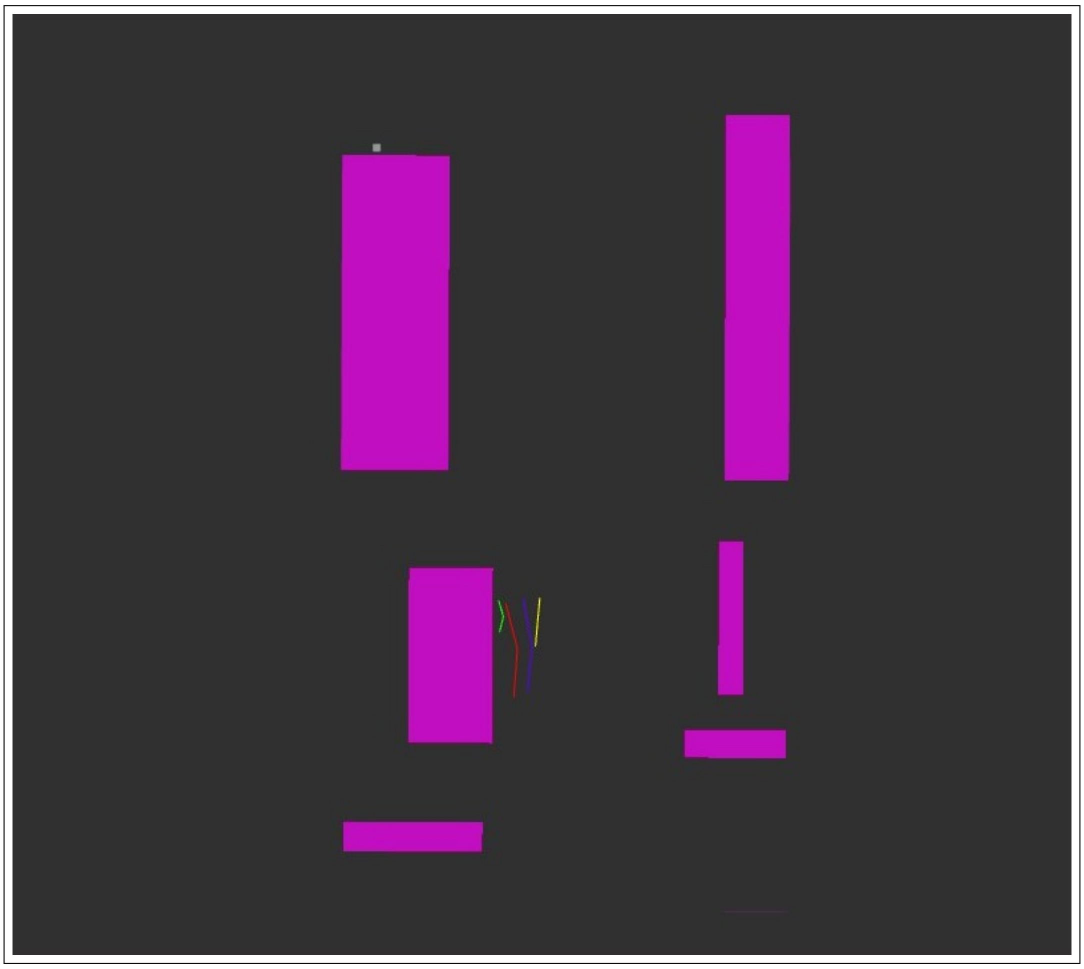
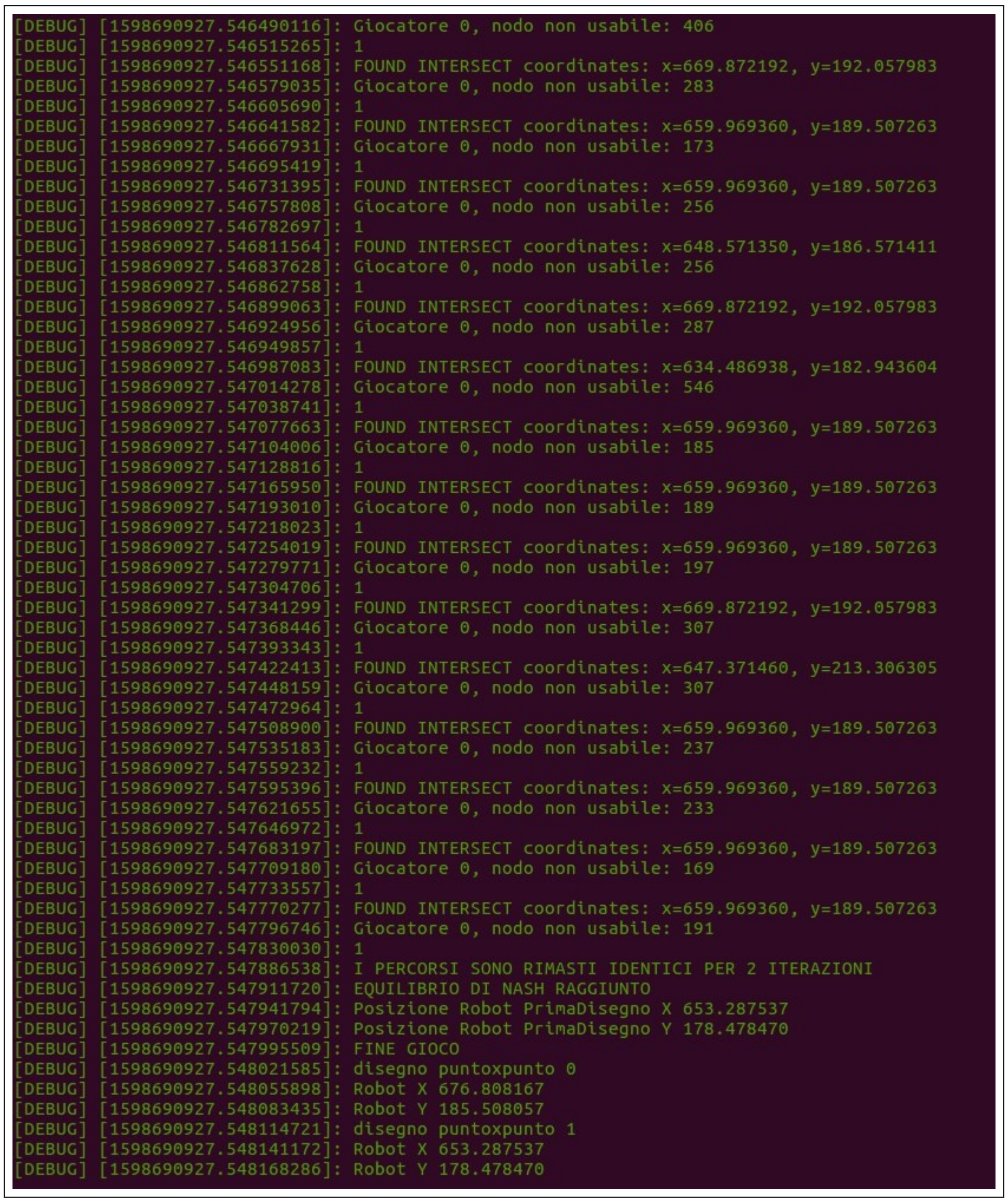


Figure 5.15. Second step of movement

Here, we have the result of computation. In figure 5.16, we have the terminal during the last part of the process.

In the upper part of image, the code applies the *Nash Game* to reach the equilibrium. The code excludes the paths that creates intersection. At the end, the code sees that for two iterations, the paths chosen are the same. The equilibrium is reached.

Now the positions are uploaded and the process goes above to the next step.



```
[DEBUG] [1598690927.546490116]: Giocatore 0, nodo non usabile: 406
[DEBUG] [1598690927.546515265]: 1
[DEBUG] [1598690927.546551168]: FOUND INTERSECT coordinates: x=669.872192, y=192.057983
[DEBUG] [1598690927.546579035]: Giocatore 0, nodo non usabile: 283
[DEBUG] [1598690927.546605690]: 1
[DEBUG] [1598690927.546641582]: FOUND INTERSECT coordinates: x=659.969360, y=189.507263
[DEBUG] [1598690927.546667931]: Giocatore 0, nodo non usabile: 173
[DEBUG] [1598690927.546695419]: 1
[DEBUG] [1598690927.546731395]: FOUND INTERSECT coordinates: x=659.969360, y=189.507263
[DEBUG] [1598690927.546757808]: Giocatore 0, nodo non usabile: 256
[DEBUG] [1598690927.546782697]: 1
[DEBUG] [1598690927.546811564]: FOUND INTERSECT coordinates: x=648.571350, y=186.571411
[DEBUG] [1598690927.546837628]: Giocatore 0, nodo non usabile: 256
[DEBUG] [1598690927.546862758]: 1
[DEBUG] [1598690927.546899063]: FOUND INTERSECT coordinates: x=669.872192, y=192.057983
[DEBUG] [1598690927.546924956]: Giocatore 0, nodo non usabile: 287
[DEBUG] [1598690927.546949857]: 1
[DEBUG] [1598690927.546987083]: FOUND INTERSECT coordinates: x=634.486938, y=182.943604
[DEBUG] [1598690927.547014278]: Giocatore 0, nodo non usabile: 546
[DEBUG] [1598690927.547038741]: 1
[DEBUG] [1598690927.547077663]: FOUND INTERSECT coordinates: x=659.969360, y=189.507263
[DEBUG] [1598690927.547104006]: Giocatore 0, nodo non usabile: 185
[DEBUG] [1598690927.547128816]: 1
[DEBUG] [1598690927.547165950]: FOUND INTERSECT coordinates: x=659.969360, y=189.507263
[DEBUG] [1598690927.547193010]: Giocatore 0, nodo non usabile: 189
[DEBUG] [1598690927.547218023]: 1
[DEBUG] [1598690927.547254019]: FOUND INTERSECT coordinates: x=659.969360, y=189.507263
[DEBUG] [1598690927.547279771]: Giocatore 0, nodo non usabile: 197
[DEBUG] [1598690927.547304706]: 1
[DEBUG] [1598690927.547341299]: FOUND INTERSECT coordinates: x=669.872192, y=192.057983
[DEBUG] [1598690927.547368446]: Giocatore 0, nodo non usabile: 307
[DEBUG] [1598690927.547393343]: 1
[DEBUG] [1598690927.547422413]: FOUND INTERSECT coordinates: x=647.371460, y=213.306305
[DEBUG] [1598690927.547448159]: Giocatore 0, nodo non usabile: 307
[DEBUG] [1598690927.547472964]: 1
[DEBUG] [1598690927.547508900]: FOUND INTERSECT coordinates: x=659.969360, y=189.507263
[DEBUG] [1598690927.547535183]: Giocatore 0, nodo non usabile: 237
[DEBUG] [1598690927.547559232]: 1
[DEBUG] [1598690927.547595396]: FOUND INTERSECT coordinates: x=659.969360, y=189.507263
[DEBUG] [1598690927.547621655]: Giocatore 0, nodo non usabile: 233
[DEBUG] [1598690927.547646972]: 1
[DEBUG] [1598690927.547683197]: FOUND INTERSECT coordinates: x=659.969360, y=189.507263
[DEBUG] [1598690927.547709180]: Giocatore 0, nodo non usabile: 169
[DEBUG] [1598690927.547733557]: 1
[DEBUG] [1598690927.547770277]: FOUND INTERSECT coordinates: x=659.969360, y=189.507263
[DEBUG] [1598690927.547796746]: Giocatore 0, nodo non usabile: 191
[DEBUG] [1598690927.547830030]: 1
[DEBUG] [1598690927.547886538]: I PERCORSI SONO RIMASTI IDENTICI PER 2 ITERAZIONI
[DEBUG] [1598690927.547911720]: EQUILIBRIO DI NASH RAGGIUNTO
[DEBUG] [1598690927.547941794]: Posizione Robot PrimaDisegno X 653.287537
[DEBUG] [1598690927.547970219]: Posizione Robot PrimaDisegno Y 178.478470
[DEBUG] [1598690927.547995509]: FINE GIOCO
[DEBUG] [1598690927.548021585]: disegno puntoxpunto 0
[DEBUG] [1598690927.548055898]: Robot X 676.808167
[DEBUG] [1598690927.548083435]: Robot Y 185.508057
[DEBUG] [1598690927.548114721]: disegno puntoxpunto 1
[DEBUG] [1598690927.548141172]: Robot X 653.287537
[DEBUG] [1598690927.548168286]: Robot Y 178.478470
```

Figure 5.16. Data of algorithm for the second step

Third Step

Now the code reloads the new positions of players, reading them from the input file, and the new position of robot, obtained as result of last iteration.

In this third case, the robot intersects again the path of first player.

The code finds a new path to avoid the collision. To do it, the code explores new reachable poses, according to the increasing cost. At the end, if the game reached the *Nash Equilibrium*, the code shows the result in the RVIZ environment. This result is showed in figure 5.17.

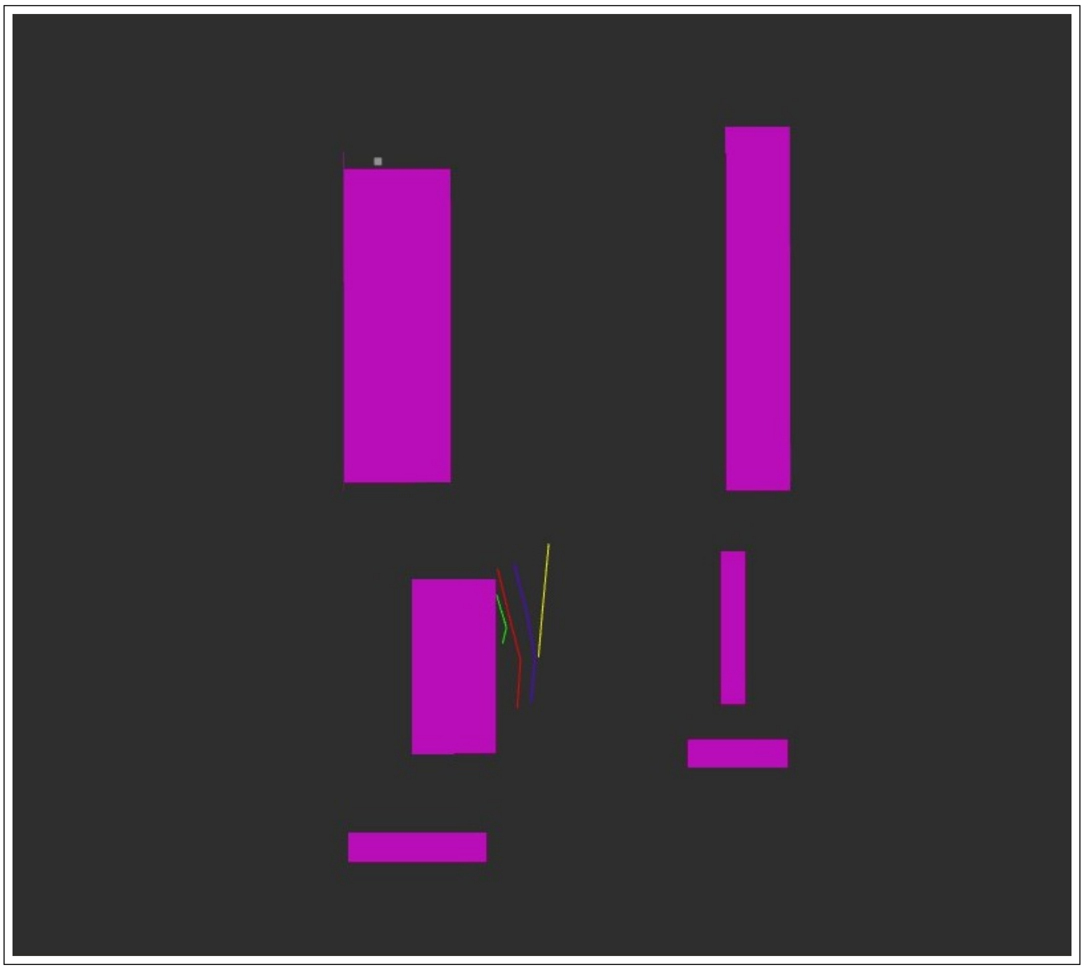
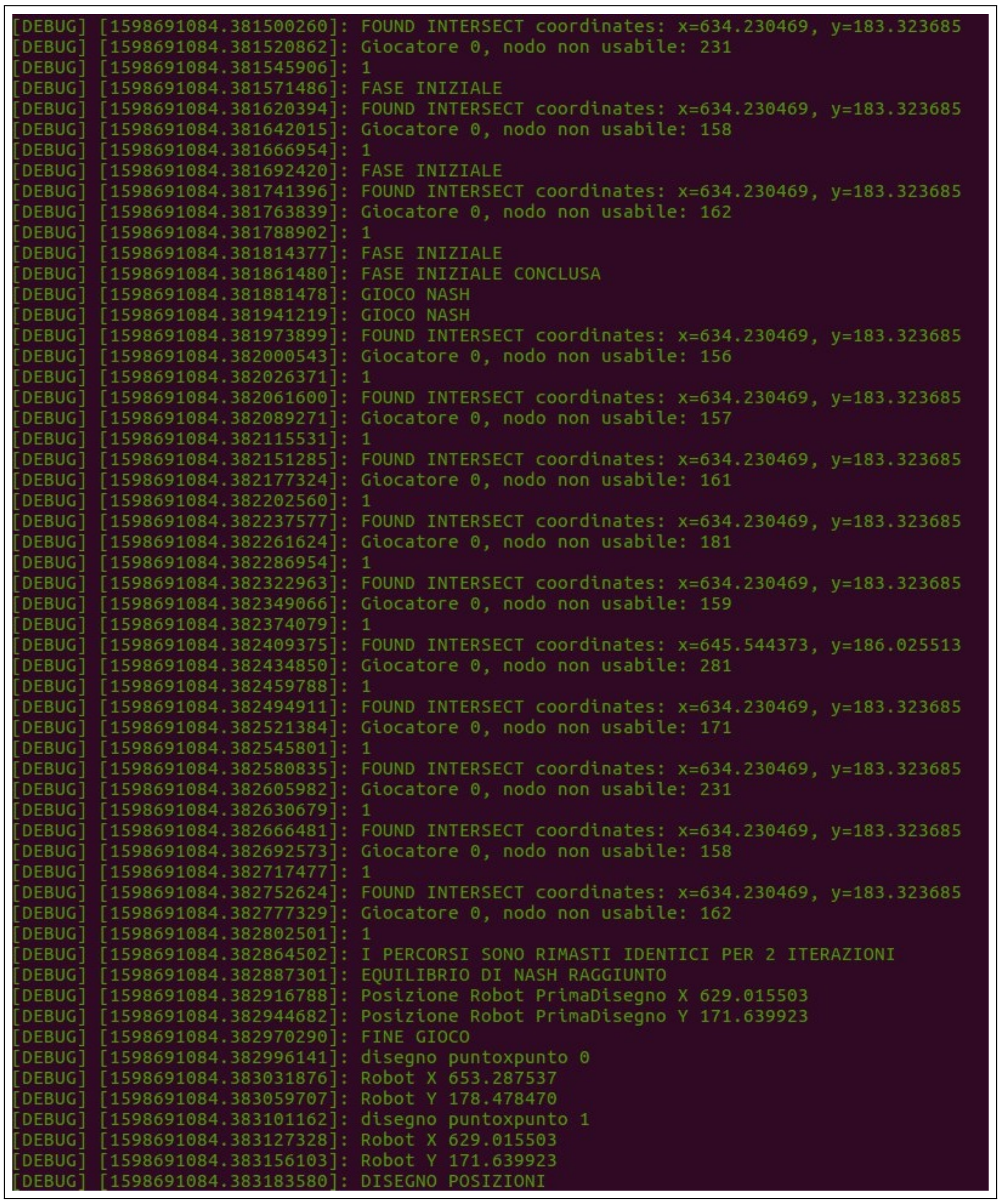


Figure 5.17. Third step of movement

Here, we have the result of computation. In figure 5.18, we have the last part of the process.

In the upper part of image, the algorithm applies the *Nash Game* to reach the equilibrium. The algorithm excludes the paths that creates intersection. At the end, the algorithm detects that for two iterations, the paths chosen are the same. The equilibrium is reached.

Now the positions are uploaded and the process goes above to the next step.



```
[DEBUG] [1598691084.381500260]: FOUND INTERSECT coordinates: x=634.230469, y=183.323685
[DEBUG] [1598691084.381520862]: Giocatore 0, nodo non usabile: 231
[DEBUG] [1598691084.381545906]: 1
[DEBUG] [1598691084.381571486]: FASE INIZIALE
[DEBUG] [1598691084.381620394]: FOUND INTERSECT coordinates: x=634.230469, y=183.323685
[DEBUG] [1598691084.381642015]: Giocatore 0, nodo non usabile: 158
[DEBUG] [1598691084.381666954]: 1
[DEBUG] [1598691084.381692420]: FASE INIZIALE
[DEBUG] [1598691084.381741396]: FOUND INTERSECT coordinates: x=634.230469, y=183.323685
[DEBUG] [1598691084.381763839]: Giocatore 0, nodo non usabile: 162
[DEBUG] [1598691084.381788902]: 1
[DEBUG] [1598691084.381814377]: FASE INIZIALE
[DEBUG] [1598691084.381861480]: FASE INIZIALE CONCLUSA
[DEBUG] [1598691084.381881478]: GIOCO NASH
[DEBUG] [1598691084.381941219]: GIOCO NASH
[DEBUG] [1598691084.381973899]: FOUND INTERSECT coordinates: x=634.230469, y=183.323685
[DEBUG] [1598691084.382000543]: Giocatore 0, nodo non usabile: 156
[DEBUG] [1598691084.382026371]: 1
[DEBUG] [1598691084.382061600]: FOUND INTERSECT coordinates: x=634.230469, y=183.323685
[DEBUG] [1598691084.382089271]: Giocatore 0, nodo non usabile: 157
[DEBUG] [1598691084.382115531]: 1
[DEBUG] [1598691084.382151285]: FOUND INTERSECT coordinates: x=634.230469, y=183.323685
[DEBUG] [1598691084.382177324]: Giocatore 0, nodo non usabile: 161
[DEBUG] [1598691084.382202560]: 1
[DEBUG] [1598691084.382237577]: FOUND INTERSECT coordinates: x=634.230469, y=183.323685
[DEBUG] [1598691084.382261624]: Giocatore 0, nodo non usabile: 181
[DEBUG] [1598691084.382286954]: 1
[DEBUG] [1598691084.382322963]: FOUND INTERSECT coordinates: x=634.230469, y=183.323685
[DEBUG] [1598691084.382349066]: Giocatore 0, nodo non usabile: 159
[DEBUG] [1598691084.382374079]: 1
[DEBUG] [1598691084.382409375]: FOUND INTERSECT coordinates: x=645.544373, y=186.025513
[DEBUG] [1598691084.382434850]: Giocatore 0, nodo non usabile: 281
[DEBUG] [1598691084.382459788]: 1
[DEBUG] [1598691084.382494911]: FOUND INTERSECT coordinates: x=634.230469, y=183.323685
[DEBUG] [1598691084.382521384]: Giocatore 0, nodo non usabile: 171
[DEBUG] [1598691084.382545801]: 1
[DEBUG] [1598691084.382580835]: FOUND INTERSECT coordinates: x=634.230469, y=183.323685
[DEBUG] [1598691084.382605982]: Giocatore 0, nodo non usabile: 231
[DEBUG] [1598691084.382630679]: 1
[DEBUG] [1598691084.382666481]: FOUND INTERSECT coordinates: x=634.230469, y=183.323685
[DEBUG] [1598691084.382692573]: Giocatore 0, nodo non usabile: 158
[DEBUG] [1598691084.382717477]: 1
[DEBUG] [1598691084.382752624]: FOUND INTERSECT coordinates: x=634.230469, y=183.323685
[DEBUG] [1598691084.382777329]: Giocatore 0, nodo non usabile: 162
[DEBUG] [1598691084.382802501]: 1
[DEBUG] [1598691084.382864502]: I PERCORSI SONO RIMASTI IDENTICI PER 2 ITERAZIONI
[DEBUG] [1598691084.382887301]: EQUILIBRIO DI NASH RAGGIUNTO
[DEBUG] [1598691084.382916788]: Posizione Robot PrimaDisegno X 629.015503
[DEBUG] [1598691084.382944682]: Posizione Robot PrimaDisegno Y 171.639923
[DEBUG] [1598691084.382970290]: FINE GIOCO
[DEBUG] [1598691084.382996141]: disegno puntoxpunto 0
[DEBUG] [1598691084.383031876]: Robot X 653.287537
[DEBUG] [1598691084.383059707]: Robot Y 178.478470
[DEBUG] [1598691084.383101162]: disegno puntoxpunto 1
[DEBUG] [1598691084.383127328]: Robot X 629.015503
[DEBUG] [1598691084.383156103]: Robot Y 171.639923
[DEBUG] [1598691084.383183580]: DISEGNO POSIZIONI
```

Figure 5.18. Data of algorithm for the third step

Fourth Step

Now the algorithm reloads the new positions of players, reading them from the input file, and the new position of robot, obtained as result of last iteration.

In this fourth case, the robot doesn't intersect any players.

The code finds the least expensive path. At the end, if the game reached the *Nash Equilibrium*, the algorithm shows the result in the RVIZ environment. This result is showed figure 5.19. This time the result comes very fast due to the lack of collisions.

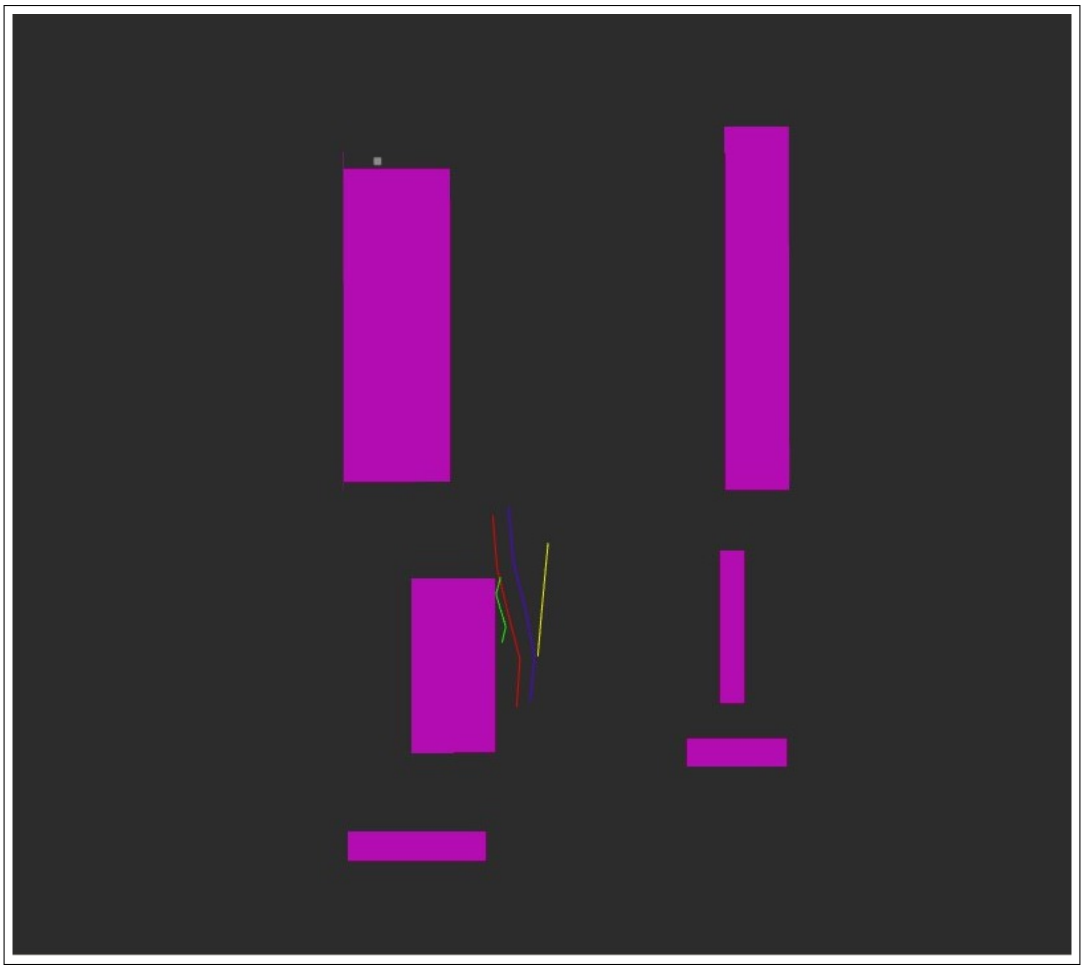
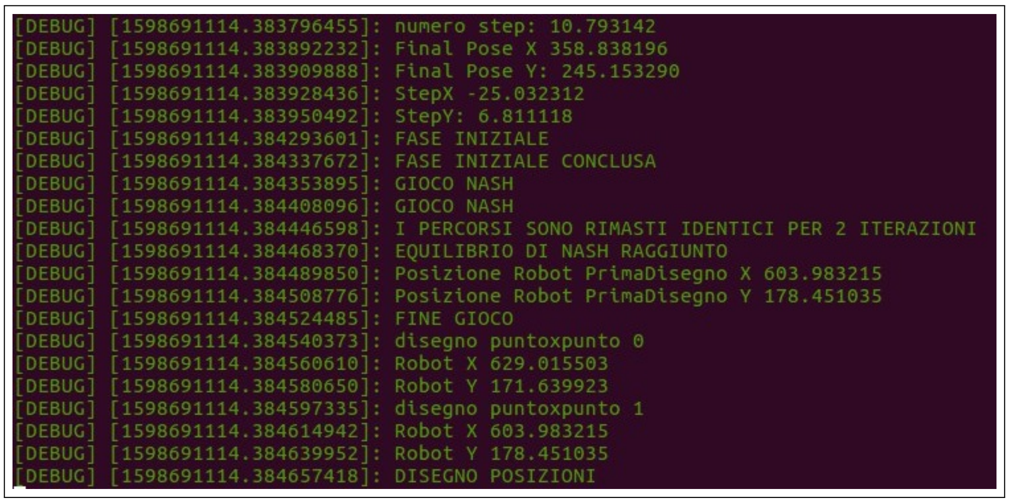


Figure 5.19. Fourth step of movement

Here we have the result of computation. In figure 5.20, we have the whole process.

In the upper part of code, we have the algorithm that finds beginning path. After this phase, called by the algorithm *"FASE INIZIALE"*, the algorithm applies the *Nash Game* to reach the equilibrium. The algorithm finds that the paths are the same for two iteration. This means that we don't have intersection between players, and the first path found is also the optimal. Then the *Nash equilibrium* is reached.

Now the positions are uploaded and the process goes above to the next step.



```
[DEBUG] [1598691114.383796455]: numero step: 10.793142
[DEBUG] [1598691114.383892232]: Final Pose X 358.838196
[DEBUG] [1598691114.383909888]: Final Pose Y: 245.153290
[DEBUG] [1598691114.383928436]: StepX -25.032312
[DEBUG] [1598691114.383950492]: StepY: 6.811118
[DEBUG] [1598691114.384293601]: FASE INIZIALE
[DEBUG] [1598691114.384337672]: FASE INIZIALE CONCLUSA
[DEBUG] [1598691114.384353895]: GIOCO NASH
[DEBUG] [1598691114.384408096]: GIOCO NASH
[DEBUG] [1598691114.384446598]: I PERCORSI SONO RIMASTI IDENTICI PER 2 ITERAZIONI
[DEBUG] [1598691114.384468370]: EQUILIBRIO DI NASH RAGGIUNTO
[DEBUG] [1598691114.384489850]: Posizione Robot PrimaDisegno X 603.983215
[DEBUG] [1598691114.384508776]: Posizione Robot PrimaDisegno Y 178.451035
[DEBUG] [1598691114.384524485]: FINE GIOCO
[DEBUG] [1598691114.384540373]: disegno puntoxpunto 0
[DEBUG] [1598691114.384560610]: Robot X 629.015503
[DEBUG] [1598691114.384580650]: Robot Y 171.639923
[DEBUG] [1598691114.384597335]: disegno puntoxpunto 1
[DEBUG] [1598691114.384614942]: Robot X 603.983215
[DEBUG] [1598691114.384639952]: Robot Y 178.451035
[DEBUG] [1598691114.384657418]: DISEGNO POSIZIONI
```

Figure 5.20. Data of algorithm for the fourth step

Fifth Step

Now the algorithm reloads the new positions of players, reading them from the input file, and the new position of robot, obtained as result of last iteration.

In this fifth case, the robot doesn't intersect any players.

The algorithm finds the least expensive path. At the end, if the game reached the *Nash Equilibrium*, the algorithm shows the result in the RVIZ environment. This result is showed in figure 5.21. This time, the result comes very fast due to the lack of collisions.

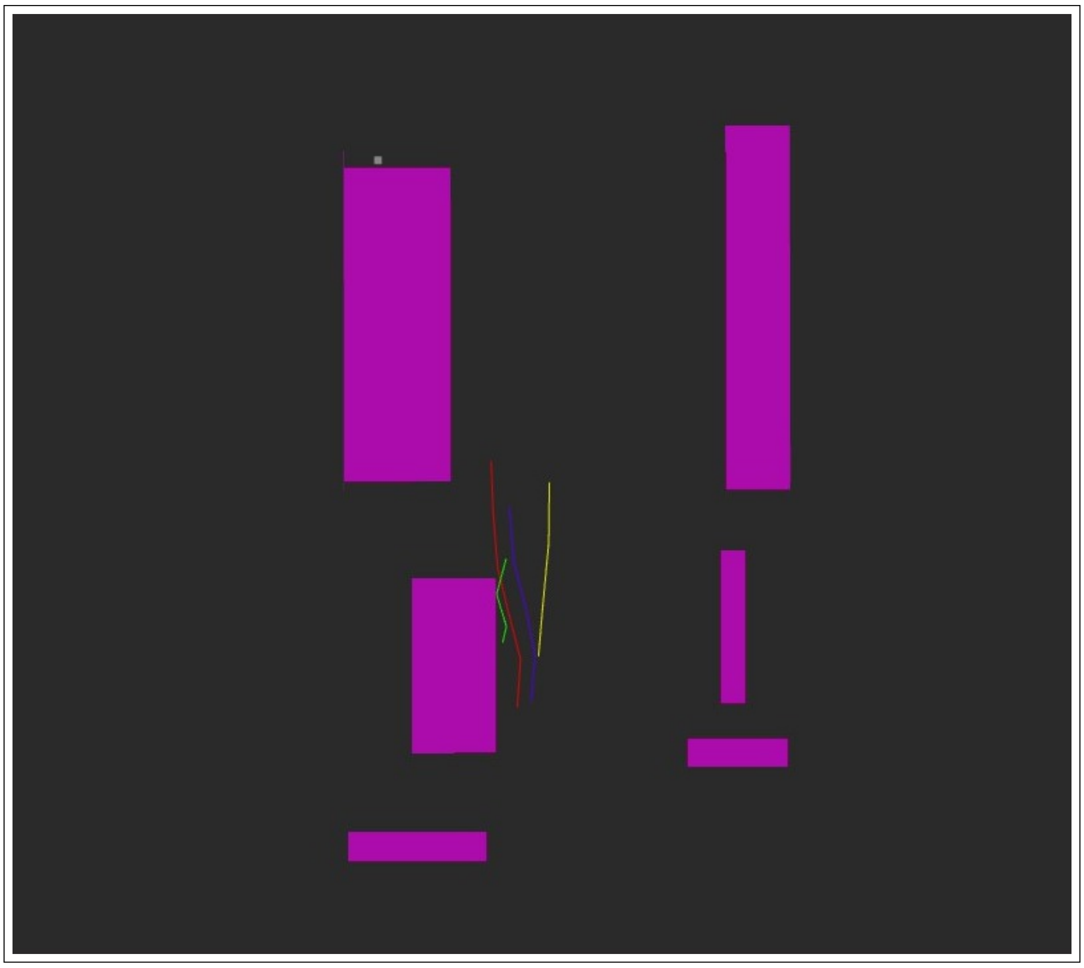
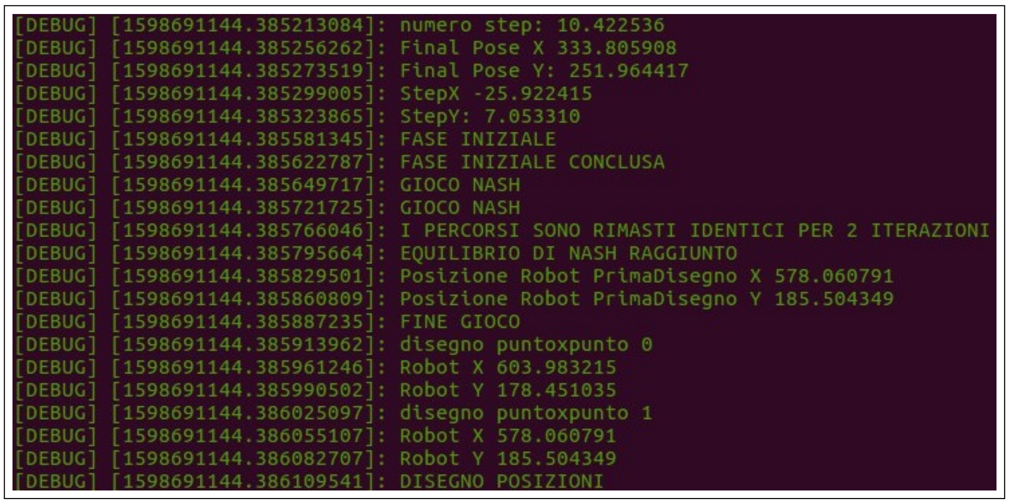


Figure 5.21. Fifth step of movement

Here we have the result of computation. In figure 5.22, we have the whole process.

In the upper part of algorithm, we have the algorithm that finds beginning path. After this phase, called by the algorithm *"FASE INIZIALE"*, the algorithm applies the *Nash Game* to reach the equilibrium. The algorithm finds that the paths are the same for two iteration. This means, that we don't have intersection between players, and the first path found are also the optimal. Then the *Nash equilibrium* is reached.

Now the positions are uploaded and the process goes above to the next step.



```
[DEBUG] [1598691144.385213084]: numero step: 10.422536
[DEBUG] [1598691144.385256262]: Final Pose X 333.805908
[DEBUG] [1598691144.385273519]: Final Pose Y: 251.964417
[DEBUG] [1598691144.385299005]: StepX -25.922415
[DEBUG] [1598691144.385323865]: StepY: 7.053310
[DEBUG] [1598691144.385581345]: FASE INIZIALE
[DEBUG] [1598691144.385622787]: FASE INIZIALE CONCLUSA
[DEBUG] [1598691144.385649717]: GIOCO NASH
[DEBUG] [1598691144.385721725]: GIOCO NASH
[DEBUG] [1598691144.385766046]: I PERCORSI SONO RIMASTI IDENTICI PER 2 ITERAZIONI
[DEBUG] [1598691144.385795664]: EQUILIBRIO DI NASH RAGGIUNTO
[DEBUG] [1598691144.385829501]: Posizione Robot PrimaDisegno X 578.060791
[DEBUG] [1598691144.385860809]: Posizione Robot PrimaDisegno Y 185.504349
[DEBUG] [1598691144.385887235]: FINE GIOCO
[DEBUG] [1598691144.385913962]: disegno puntoxpunto 0
[DEBUG] [1598691144.385961246]: Robot X 603.983215
[DEBUG] [1598691144.385990502]: Robot Y 178.451035
[DEBUG] [1598691144.386025097]: disegno puntoxpunto 1
[DEBUG] [1598691144.386055107]: Robot X 578.060791
[DEBUG] [1598691144.386082707]: Robot Y 185.504349
[DEBUG] [1598691144.386109541]: DISEGNO POSIZIONI
```

Figure 5.22. Data of algorithm for the fifth step

Sixth Step

Now the code reloads the new positions of players, reading them from the input file, and the new position of robot, obtained as result of last iteration.

In this sixth case, the robot doesn't intersect any players.

The algorithm finds the least expensive path. At the end, if the game reached the *Nash Equilibrium*, the algorithm shows the result in the RVIZ environment. This result is showed in figure 5.23. This time the result comes very fast due to the lack of collisions.

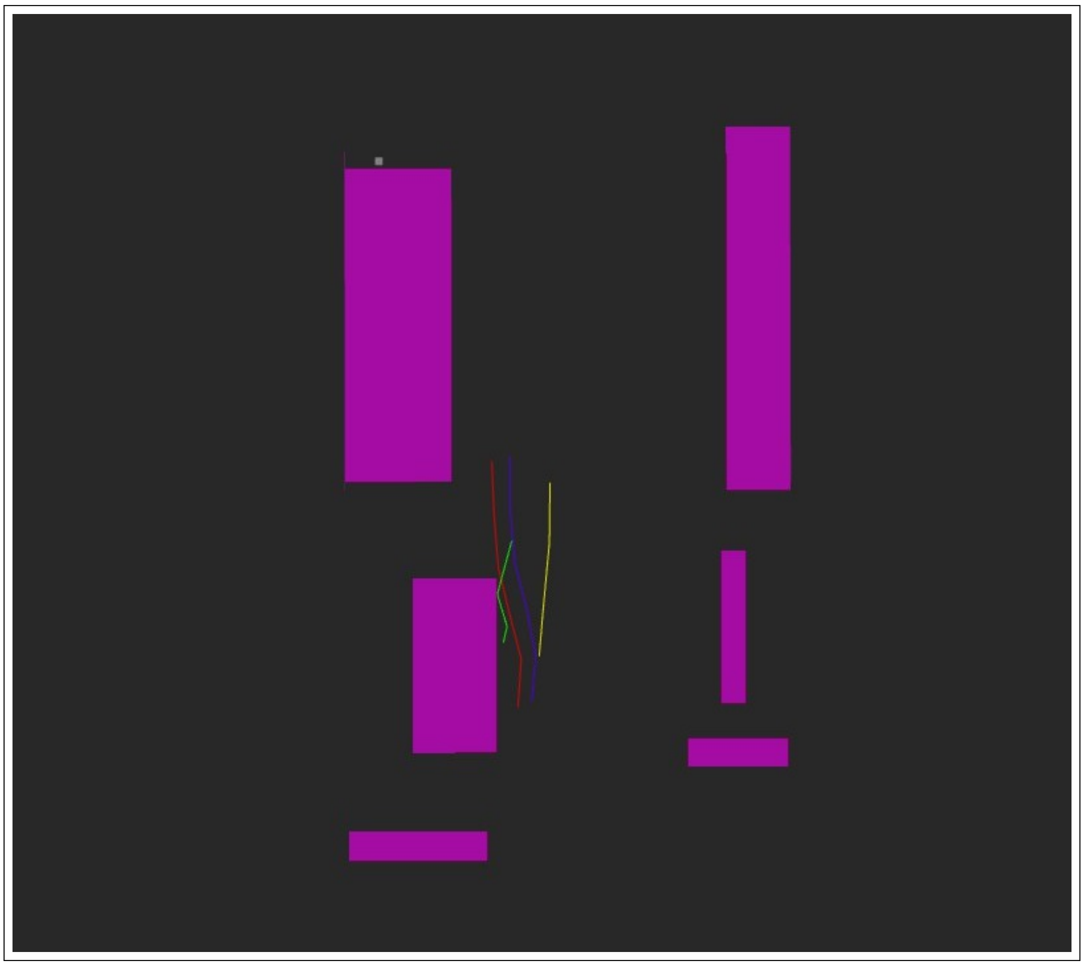
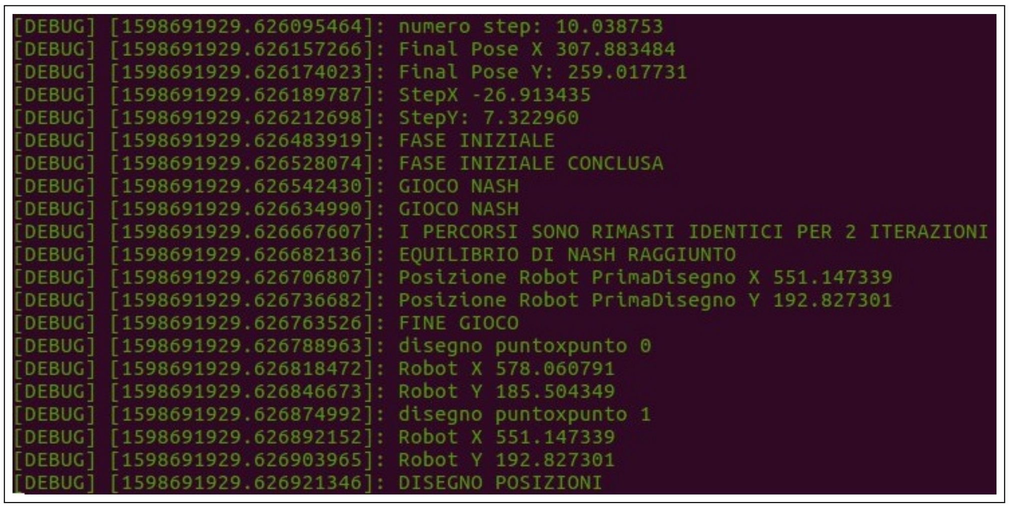


Figure 5.23. Sixth step of movement

Here we have the result of computation. In figure 5.24, we have the whole process.

In the upper part of algorithm, we have the algorithm that finds beginning path. After this phase, called by the algorithm *"FASE INIZIALE"*, the algorithm applies the *Nash Game* to reach the equilibrium. The algorithm finds that the paths are the same for two iteration. This means, that we don't have intersection between players, and the first path found are also the optimal. Then the *Nash equilibrium* is reached.

Now the positions are uploaded and the process goes above to the next step.



```
[DEBUG] [1598691929.626095464]: numero step: 10.038753
[DEBUG] [1598691929.626157266]: Final Pose X 307.883484
[DEBUG] [1598691929.626174023]: Final Pose Y: 259.017731
[DEBUG] [1598691929.626189787]: StepX -26.913435
[DEBUG] [1598691929.626212698]: StepY: 7.322960
[DEBUG] [1598691929.626483919]: FASE INIZIALE
[DEBUG] [1598691929.626528074]: FASE INIZIALE CONCLUSA
[DEBUG] [1598691929.626542430]: GIOCO NASH
[DEBUG] [1598691929.626634990]: GIOCO NASH
[DEBUG] [1598691929.626667607]: I PERCORSI SONO RIMASTI IDENTICI PER 2 ITERAZIONI
[DEBUG] [1598691929.626682136]: EQUILIBRIO DI NASH RAGGIUNTO
[DEBUG] [1598691929.626706807]: Posizione Robot PrimaDisegno X 551.147339
[DEBUG] [1598691929.626736682]: Posizione Robot PrimaDisegno Y 192.827301
[DEBUG] [1598691929.626763526]: FINE GIOCO
[DEBUG] [1598691929.626788963]: disegno puntoxpunto 0
[DEBUG] [1598691929.626818472]: Robot X 578.060791
[DEBUG] [1598691929.626846673]: Robot Y 185.504349
[DEBUG] [1598691929.626874992]: disegno puntoxpunto 1
[DEBUG] [1598691929.626892152]: Robot X 551.147339
[DEBUG] [1598691929.626903965]: Robot Y 192.827301
[DEBUG] [1598691929.626921346]: DISEGNO POSIZIONI
```

Figure 5.24. Data of algorithm for the sixth step

Seventh Step

Now the algorithm reloads the new positions of players, reading them from the input file, and the new position of robot, obtained as result of last iteration.

In this seventh case, the robot doesn't intersect any players.

The algorithm finds the least expensive path. At the end, if the game reached the *Nash Equilibrium*, the code shows the result in the RVIZ environment. This result is showed in figure 5.25. This time the result comes very fast due to the lack of collisions.

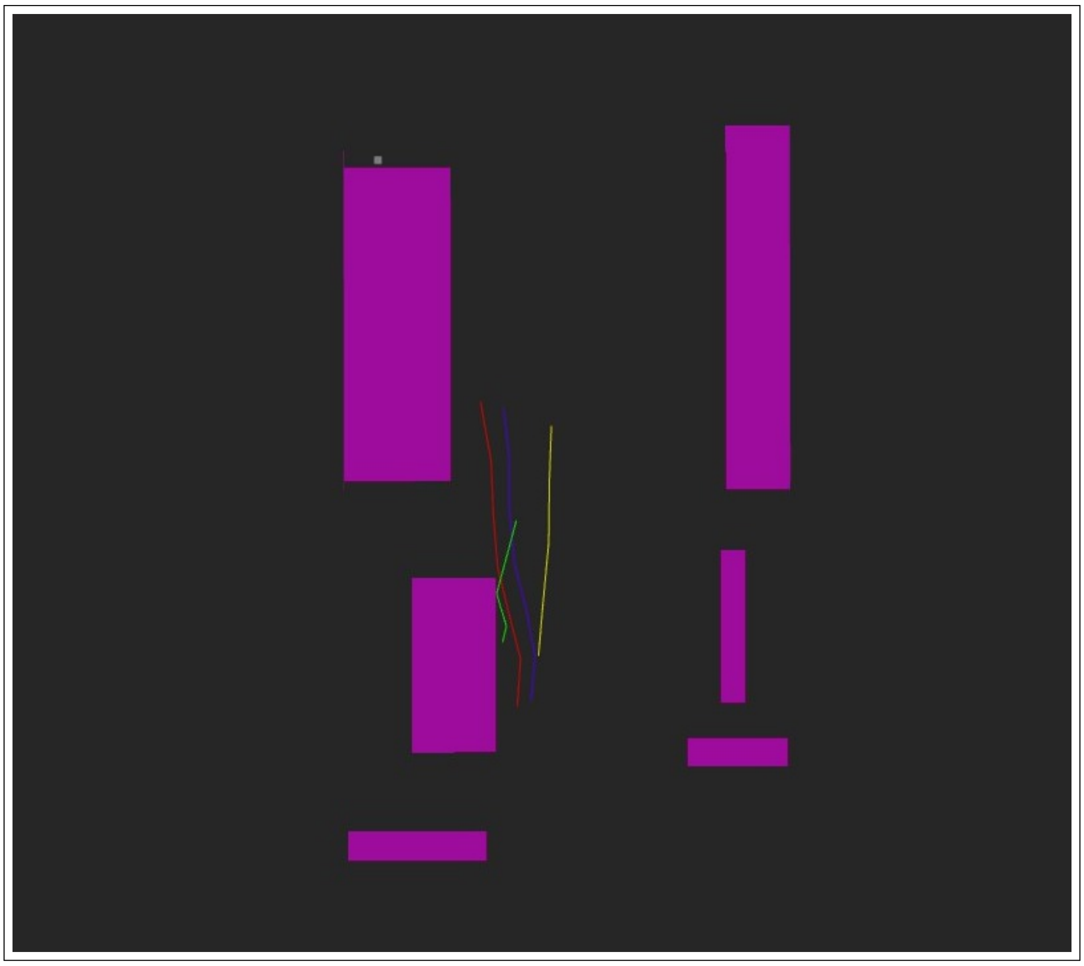
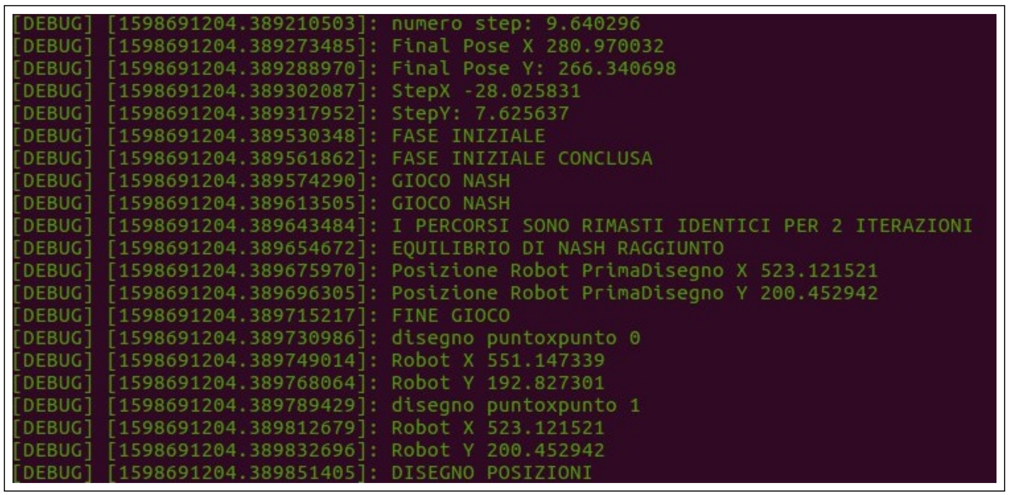


Figure 5.25. Seventh step of movement

Here we have the result of computation. In figure 5.26, we have the whole process.

In the upper part of code, we have the algorithm that finds beginning path. After this phase, called by the algorithm *"FASE INIZIALE"*, the algorithm applies the *Nash Game* to reach the equilibrium. The algorithm finds that the paths are the same for two iteration. This means, that we don't have intersection between players, and the first path found are also the optimal. Then the *Nash equilibrium* is reached.

Now the positions are uploaded and the process goes above to the next step.



```
[DEBUG] [1598691204.389210503]: numero step: 9.640296
[DEBUG] [1598691204.389273485]: Final Pose X 280.970032
[DEBUG] [1598691204.389288970]: Final Pose Y: 266.340698
[DEBUG] [1598691204.389302087]: StepX -28.025831
[DEBUG] [1598691204.389317952]: StepY: 7.625637
[DEBUG] [1598691204.389530348]: FASE INIZIALE
[DEBUG] [1598691204.389561862]: FASE INIZIALE CONCLUSA
[DEBUG] [1598691204.389574290]: GIOCO NASH
[DEBUG] [1598691204.389613505]: GIOCO NASH
[DEBUG] [1598691204.389643484]: I PERCORSI SONO RIMASTI IDENTICI PER 2 ITERAZIONI
[DEBUG] [1598691204.389654672]: EQUILIBRIO DI NASH RAGGIUNTO
[DEBUG] [1598691204.389675970]: Posizione Robot PrimaDisegno X 523.121521
[DEBUG] [1598691204.389696305]: Posizione Robot PrimaDisegno Y 200.452942
[DEBUG] [1598691204.389715217]: FINE GIOCO
[DEBUG] [1598691204.389730986]: disegno puntoxpunto 0
[DEBUG] [1598691204.389749014]: Robot X 551.147339
[DEBUG] [1598691204.389768064]: Robot Y 192.827301
[DEBUG] [1598691204.389789429]: disegno puntoxpunto 1
[DEBUG] [1598691204.389812679]: Robot X 523.121521
[DEBUG] [1598691204.389832696]: Robot Y 200.452942
[DEBUG] [1598691204.389851405]: DISEGNO POSIZIONI
```

Figure 5.26. Data of algorithm for the seventh step

Eighth Step

Now the algorithm reloads the new positions of players, reading them from the input file, and the new position of robot, obtained as result of last iteration.

In this eighth case, the robot doesn't intersect any players.

The algorithm finds the least expensive path. At the end, if the game reached the *Nash Equilibrium*, the algorithm shows the result in the RVIZ environment. This result is showed in figure 5.27. This time the result comes very fast due to the lack of collisions.

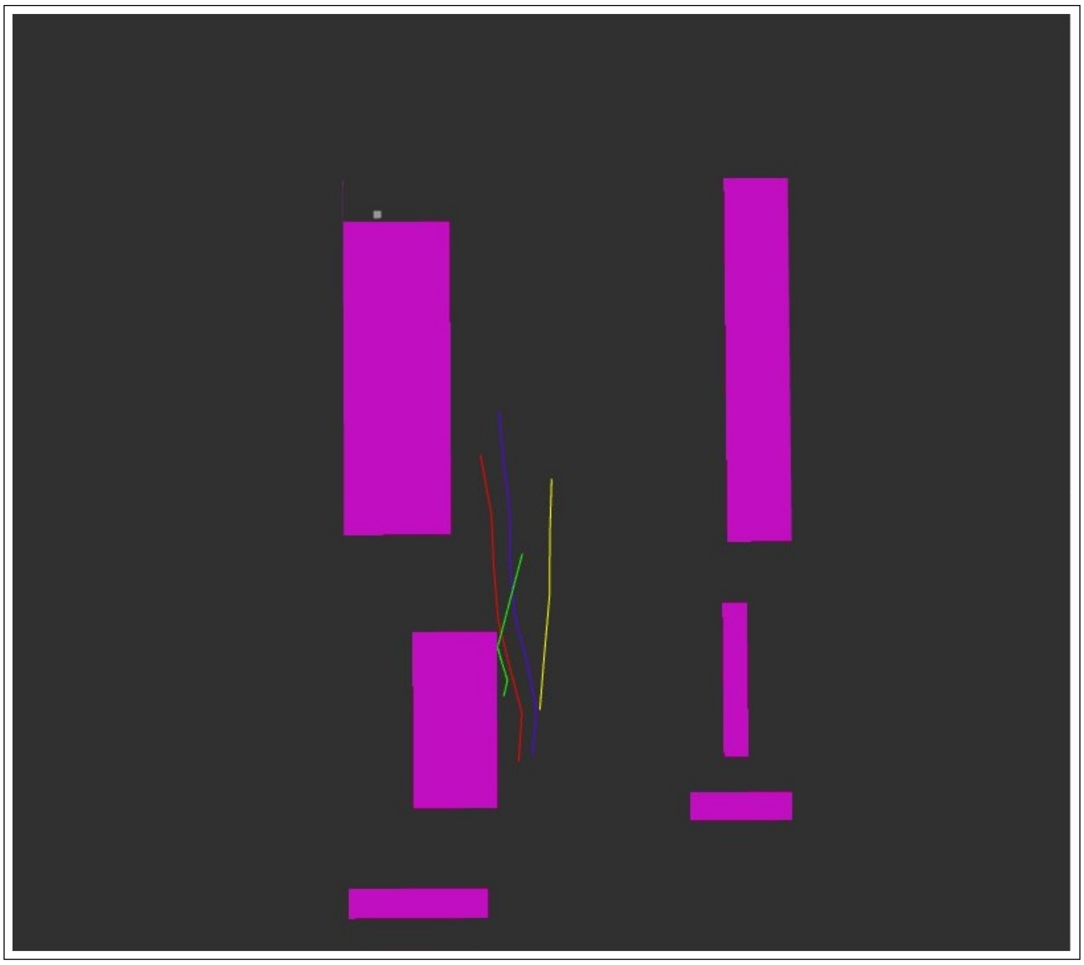
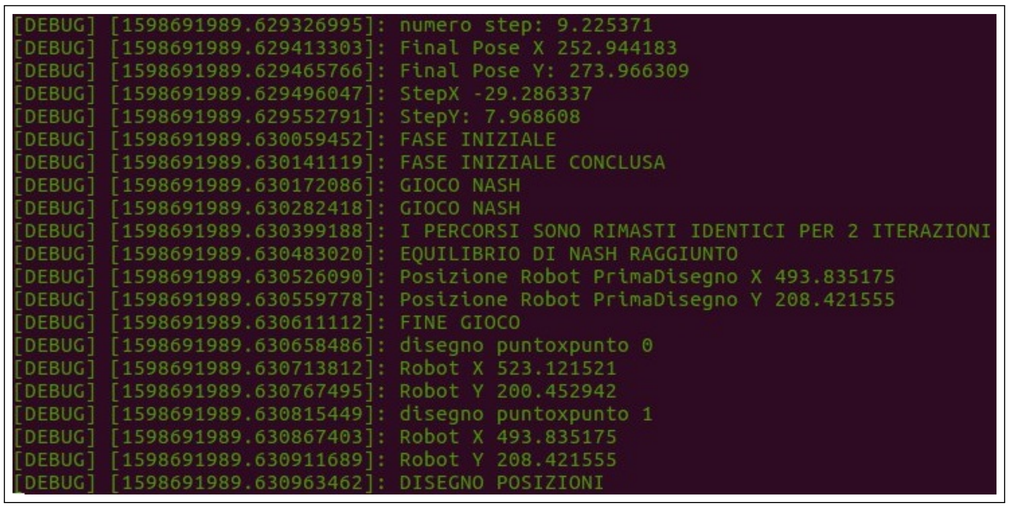


Figure 5.27. Eighth step of movement

Here, we have the result of computation. In figure 5.28, we have the whole process.

In the upper part of algorithm, we have the code that finds beginning path. After this phase, called by the algorithm *"FASE INIZIALE"*, the algorithm applies the *Nash Game* to reach the equilibrium. The algorithm finds that the paths are the same for two iteration. This means, that we don't have intersection between players, and the first path found are also the optimal. Then the *Nash equilibrium* is reached.

Now the positions are uploaded and the process goes above to the next step.



```
[DEBUG] [1598691989.629326995]: numero step: 9.225371
[DEBUG] [1598691989.629413303]: Final Pose X 252.944183
[DEBUG] [1598691989.629465766]: Final Pose Y: 273.966309
[DEBUG] [1598691989.629496047]: StepX -29.286337
[DEBUG] [1598691989.629552791]: StepY: 7.968608
[DEBUG] [1598691989.630059452]: FASE INIZIALE
[DEBUG] [1598691989.630141119]: FASE INIZIALE CONCLUSA
[DEBUG] [1598691989.630172086]: GIOCO NASH
[DEBUG] [1598691989.630282418]: GIOCO NASH
[DEBUG] [1598691989.630399188]: I PERCORSI SONO RIMASTI IDENTICI PER 2 ITERAZIONI
[DEBUG] [1598691989.630483020]: EQUILIBRIO DI NASH RAGGIUNTO
[DEBUG] [1598691989.630526090]: Posizione Robot PrimaDisegno X 493.835175
[DEBUG] [1598691989.630559778]: Posizione Robot PrimaDisegno Y 208.421555
[DEBUG] [1598691989.630611112]: FINE GIOCO
[DEBUG] [1598691989.630658486]: disegno puntoxpunto 0
[DEBUG] [1598691989.630713812]: Robot X 523.121521
[DEBUG] [1598691989.630767495]: Robot Y 200.452942
[DEBUG] [1598691989.630815449]: disegno puntoxpunto 1
[DEBUG] [1598691989.630867403]: Robot X 493.835175
[DEBUG] [1598691989.630911689]: Robot Y 208.421555
[DEBUG] [1598691989.630963462]: DISEGNO POSIZIONI
```

Figure 5.28. Data of algorithm for the eighth step

Chapter 6

Conclusions

The goal of the thesis is to create a software, that could be implemented on robots to emulate an human-like behavior. In particular, we focus the work on how the robot can interact with humans in a crowded and unknown scenario.

In order to reach this goal, we search in literature all the methods that could be useful to reach a feasible solutions. We find different approaches leading to different results. In particular, the models analyzed are:

1. *Predictive model*: the player "predicts" how the other players could move in the future. This model creates possible oscillation on the player movement and this cannot be acceptable for human-being.

2. *Reactive model*: the player "reacts" based on the motion of other players. I.e. The player changes its path and trajectory every time that it finds a possible collision. This solution creates a "rough" behavior of player. This kind of model generates a behavior far from an human-like.

3. *Learning model*: The player "learns" how to behave using the information stored in a database. This model is very useful if the robot works always in the same environment. Moreover, the database can be reloaded if the scenario change. The negative aspect is that the robot needs a lot of computation to reload the data and cannot be used in a real-time crowded scenario.

4. *Game theory model*: the player "plays" with the others gamers during the process. The Game theory is useful because can reproduce rational decision. Also, according to the kind of game chosen, it can create different solutions tuned on the requirements.

After this phase, where the literature is analyzed, the *Game Theory Model* is the choice taken in order to reach the goal of thesis. This solution is chosen due to its versatility. In fact, it could be tuned according to the requirements of the target. The features of the game chosen are:

1. *Dynamics.* The game evolves during the time.

2. *Non-cooperative.* Each player thinks only to its own goal.

3. *Non-zero sum.* The sum of all gains and losses, of all players, is not strictly equal to 0.

At the end the game, the *Nash Equilibrium* must be reach. I.e. each player, knowing the other players paths, cannot change its own for at least two iteration.

At the end, we create two codes to reach the final goal. In particular the two codes are:

1. *Core code.* knowing the data of players, this code implement the core element of the proposed algorithm, by implementing the *Nash Equilibrium* to compute the solution of the game and, then, a suitable path.

2. *Edge code.* This code includes the *Core code* to be used in specific applications, such as with a realistic scenario in this thesis. I.e. it includes the dynamic requirement chosen at the beginning.

In order to validate the codes, we perform specific tests. These test are created accordingly to the specs that the codes have to reach, according to the thesis goals. The tests performed show that the robot is able to avoid obstacle and interact with humans. Also, the robot creates smooth trajectories in order to move with a human-like motion. To do that, the robot prefers to steer with low angle if it interacts with an human or an obstacle. The test demonstrates that the codes are able to avoid the intersections between robot and players, creating usable paths without collisions.

The *Edge Code* is tested also in a realistic scenario, i.e. using a real-world surveillance video, the code is able to create robot path that is acceptable for humans.

Our work could be improved and implemented in autonomous cars and many robotic fields. In the future, this solution could be useful to manage cleaning robot, delivery robot and many others that need to interact with humans.

In future, some other improvements could be done on the algorithm to make it more complete and applicable in real scenarios. Moreover, the multi robot scenario should be considered by using the proposed game theoretic approach to compute the path of more robots.

Bibliography

[1] Giada Galati, Learning from humans to improve Socially-Aware Motion Planning. Rel. Alessandro Rizzo, Sergio Grammatico. Politecnico di Torino, Corso di laurea magistrale in Mechatronic Engineering (Ingegneria Meccatronica), 2019

[2] Damiano L, Dumouchel P. Anthropomorphism in Human-Robot Co-evolution. Front Psychol. 2018;9:468. Published 2018 Mar 26. doi:10.3389/fpsyg.2018.00468.

[3] Luber, Matthias, Spinello, Luciano, Silva, Jens, Arras, Kai, 2012/10/01, 902-907, 978-1-4673-1737-5, Socially-aware robot navigation: A learning approach, doi:10.1109/IROS.2012.6385716.

[4] Embodied social interaction for service robots in hallway environments. Elena Pacchierotti, Henrik I. Christensen, and Patric Jensfelt

[5] Aiello, J. R. (1987). Human Spatial Behaviour. In D. Stokels & I. Altman (Eds.), Handbook of Environmental Psychology. New York, NY: John Wiley & Sons.

[6] Burgoon, J., Buller, D., & Woodall, W. (1989). Nonverbal Communication: The Unspoken Dialogue. New York, NY: Harper & Row.

[7] Human-Aware Robot Navigation: A Survey, Thibault Kruse, Amit Pandey, Rachid Alami, Alexandra Kirsch, 2013, 61 (12), pp.1726-1743. hal-01684295

[8] Karamouzas, I., & Overmars, M. (2010). A Velocity-Based Approach for Simulating Human Collision Avoidance. Lecture Notes in Computer Science, 180–186. doi:10.1007/978-3-642-15892-6-19

[9] Turnwald, A., Althoff, D., Wollherr, D., & Buss, M. (2016). Understanding Human Avoidance Behavior: Interaction-Aware Decision Making Based on Game Theory. International Journal of Social Robotics, 8(2), 331–351. doi:10.1007/s12369-016-0342-2

[10] Morgan Quigley, Brian Gerkeyy, Ken Conleyy, Josh Fausty, Tully Footey, Jeremy Leibsz, Eric Bergery, Rob Wheelery, Andrew Ng, ROS: an open-source Robot Operating System, Conference Paper, (2009)

[11] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In Autonomous robot vehicles, pages 396–404. Springer, 1986.

[12] Johann Borenstein, Yoram Koren, et al. The vector field histogram-fast obstacle avoidance for mobile robots. IEEE transactions on robotics and automation, 7(3):278–288, 1991.

[13] Iwan Ulrich and Johann Borenstein. Vfh+: Reliable obstacle avoidance for fast mobile robots. In Proceedings. 1998 IEEE international conference on robotics and automation (Cat. No. 98CH36146), volume 2, pages 1572–1577. IEEE, 1998.

[14] Iwan Ulrich and Johann Borenstein. Vfh/sup*: Local obstacle avoidance with look-ahead verification. In Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065), volume 3, pages 2505–2511. IEEE, 2000.

[15] Sean Quinlan and Oussama Khatib. Elastic bands: Connecting path planning and control. In [1993] Proceedings IEEE International Conference on Robotics and Automation, pages 802–807. IEEE, 1993.

[16] John Von Neumann, Oskar Morgenstern, and Harold William Kuhn. Theory of games and economic behavior (commemorative edition). Princeton university press, 2007.

[17] John Nash. Non-cooperative games. Annals of mathematics, pages 286–295, 1951.

[18] Kevin Leyton-Brown and Yoav Shoham. Essentials of game theory: A concise multidisciplinary introduction. Synthesis lectures on artificial intelligence and machine learning, 2(1):1–88, 2008.

[19] Nash, John. "Non-Cooperative Games." Annals of Mathematics, Second Series, 54, no. 2 (1951): 286-95. Accessed August 12, 2020. doi:10.2307/1969529.

[20] von Ahn, Luis. "Preliminaries of Game Theory" (PDF). Archived from the original (PDF) on 2011-10-18. Retrieved 2008-11-07.

[21] Sébastien Paris, Julien Pettré, and Stéphane Donikian. Pedestrian reactive navigation for crowd simulation: a predictive approach. In Computer Graphics Forum, volume 26, pages 665–674. Wiley Online Library, 2007.

[22] Paolo Fiorini and Zvi Shiller. Motion planning in dynamic environments using velocity obstacles. The International Journal of Robotics Research, 17(7):760– 772, 1998.

[23] Ioannis Karamouzas and Mark Overmars. A velocity-based approach for simulating human collision avoidance. In International Conference on Intelligent Virtual Agents, pages 180–186. Springer, 2010.

[24] William H Warren. Collective motion in human crowds. Current directions in psychological science, 27(4):232–240, 2018.

[25] Edward Twitchell Hall. The hidden dimension, volume 609. Garden City, NY: Doubleday, 1910.

[26] Pete Trautman, Jeremy Ma, Richard M Murray, and Andreas Krause. Robot

navigation in dense human crowds: Statistical models and experimental studies of human–robot cooperation. The International Journal of Robotics Research, 34(3):335–356, 2015.

[27] Dirk Helbing and Peter Molnar. Social force model for pedestrian dynamics. Physical review E, 51(5):4282, 1995.

[28] Kai O Arras, Slawomir Grzonka, Matthias Luber, and Wolfram Burgard. Efficient people tracking in laser range data using a multi-hypothesis leg-tracker with adaptive occlusion probabilities. In 2008 IEEE International Conference on Robotics and Automation, pages 1710–1715. IEEE, 2008.

[29] Satoshi Tadokoro, Masaki Hayashi, Yasuhiro Manabe, Yoshihiro Nakami, and Toshi Takamori. On motion planning of mobile robots which coexist and cooperate with human. In Proceedings 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human Robot Interaction and Cooperative Robots, volume 2, pages 518–523. IEEE, 1995.

[30] Andreas Schadschneider. Cellular automaton approach to pedestrian dynamicstheory. arXiv preprint cond-mat/0112117, 2001.

[31] Frank Hoeller, Dirk Schulz, Mark Moors, and Frank E Schneider. Accompanying persons with a mobile robot using motion prediction and probabilistic roadmaps. In 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 1260–1265. IEEE, 2007.

[32] Maren Bennewitz, Wolfram Burgard, Grzegorz Cielniak, and Sebastian Thrun. Learning motion patterns of people for compliant robot motion. The International Journal of Robotics Research, 24(1):31–48, 2005.

[33] Amalia F Foka and Panos E Trahanias. Predictive autonomous robot navigation. In IEEE/RSJ international conference on intelligent robots and systems, volume 1, pages 490–495. IEEE, 2002.

[34] Alexandre Alahi, Kratarth Goel, Vignesh Ramanathan, Alexandre Robicquet, Li Fei-Fei, and Silvio Savarese. Social lstm: Human trajectory prediction in crowded spaces. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 961–971, 2016.

[35] Agrim Gupta, Justin Johnson, Li Fei-Fei, Silvio Savarese, and Alexandre Alahi. Social gan: Socially acceptable trajectories with generative adversarial networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 2255–2264, 2018.

[36] Junwei Liang, Lu Jiang, Juan Carlos Niebles, Alexander G Hauptmann, and Li Fei-Fei. Peeking into the future: Predicting future person activities and locations in videos. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 5725–5734, 2019.

[37] Serge Hoogendoorn and Piet HL Bovy. Simulation of pedestrian flows by optimal control and differential games. Optimal Control Applications and Methods, 24(3):153–172, 2003.

[38] Jun Tanimoto, Aya Hagishima, and Yasukaka Tanaka. Study of bottleneck effect at an emergency evacuation exit using cellular automata model, mean field approximation analysis, and game theory. Physica A: statistical mechanics and its applications, 389(24):5611–5618, 2010.

[39] Bryan L Mesmer and Christina L Bloebaum. Modeling decision and game theory based pedestrian velocity vector decisions with interacting individuals. Safety science, 87:116–130, 2016.

[40] Annemarie Turnwald, Daniel Althoff, Dirk Wollherr, and Martin Buss. Understanding human avoidance behavior: interaction-aware decision making based on game theory. International Journal of Social Robotics, 8(2):331–351, 2016.

[41] Souvik Roy, A Borzì, and Abderrahmane Habbal. Pedestrian motion modelled by fokker–planck nash games. Royal Society open science, 4(9):170648, 2017.

[42] Waytz A, Epley N, Cacioppo JT. Social Cognition Unbound: Insights Into Anthropomorphism and Dehumanization. Curr Dir Psychol Sci. 2010;19(1):58-62. doi:10.1177/0963721409359302