Politecnico di Milano
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Clustering and Reproduction of Players' Exploration Paths in Video Games

Supervisor: Professor Francesco AMIGONI
Co-supervisors: Professor Daniele LOIACONO,
Dr. Davide AZZALINI

Thesis by:
Edoardo ZINI, ID 875275

Academic Year 2018/2019

Abstract

Collecting data from players is a common practice in the video game field. Software houses can exploit those data to identify problems and improve their products. During development data are usually collected via playtests. During a playtest the development team gathers players to collect information about how they react to the game and which are the critical points. Playtests are time consuming and cannot be performed too often.

One piece of information collected during these playtests is the path each player follows while exploring the virtual environment. Being able to cluster together similar paths, and then reproduce the paths in a cluster in an automated fashion, would not only benefit the game development process, but also constitutes an interesting scientific challenge.

In this thesis, we leverage an already existing framework capable of collecting data from human players and moving an autonomous agent in a virtual environment; we improve it to fit our needs and we perform a data collection process in order to gather enough human trajectories for our tests. We investigate which clustering procedure is the most suited for our needs, taking into account that we ideally want clusters based on the abilities of players. We also have to evaluate which metric should be used by the clustering procedure. We consider geometry-based metrics, like Euclidean, longest common subsequence, and dynamic time warping. Then we consider hidden Markov models and the answers to a survey we submitted to our players during data collection. Lastly, we assemble a list of measures capable of capturing the behaviour of the players without being geometrically bound.

After clustering player's exploration trajectories we look for ways for reproducing trajectories belonging to a cluster using an autonomous agent. We discover that the behaviour of the agent is not as deterministic as we expect, so we split our efforts: on one side we adapt our approach to deal with this issue, on the other side we analyse the autonomous agent in order to better understand the nature of this non-determinism. We find three feasible ways for reproducing trajectories in a cluster and we detail them highlighting their pros and cons.

Sommario

Raccogliere dati dai giocatori è una pratica comune nell'ambito videoludico. I team di sviluppo possono sfruttare questi dati per individuare problemi e migliorare i loro prodotti. Durante lo sviluppo i dati sono raccolti nelle sessioni di playtest. In un playtest il team di sviluppo raduna dei giocatori al fine di raccogliere informazioni su come questi reagiscono al gioco e su quali sono i punti critici. Organizzare un playtest richiede tempo, perciò sono rari.

Una delle informazioni raccolte in un playtest sono i percorsi seguiti dai giocatori mentre esplorano l'ambiente virtuale. Essere in grado di raggruppare percorsi simili tra loro, e poi riprodurre automaticamente i percorsi appartenenti ad un gruppo, non solo sarebbe utile per il processo di sviluppo, ma rappresenta anche una interessante sfida scientifica.

In questa tesi, sfruttiamo un framework pre-esistente in grado di raccogliere dati da giocatori umani e muovere un agente autonomo in un ambiente virtuale; lo miglioriamo per adattarlo alle nostre esigenze e avviamo una raccolta dati in modo da avere abbastanza percorsi seguiti da giocatori umani per i nostri test. Contemporaneamente consideriamo quale procedura per il clustering è la più adatta alle nostre necessità, tenendo in considerazione che vorremmo avere dei cluster basati sulle abilità di ciascun giocatore. Inoltre dobbiamo valutare quale metrica sia la più adatta per il clustering. Consideriamo metriche basate sulla geometria, come quella euclidea, longest common subsequence e dynamic time warping. Poi consideriamo gli hidden Markov model e le risposte a un questionario che abbiamo sottoposto ai giocatori. Infine, componiamo una lista di misure in grado di catturare il comportamento dei giocatori senza basarsi sulla geometria.

Dopo aver raggruppato i vari percorsi dei giocatori cerchiamo dei modi per riprodurre quelli appartenenti ad un cluster usando le abilità di un agente autonomo. Scopriamo che il comportamento dell'agente non è deterministico, quindi dividiamo i nostri sforzi: da un lato adattiamo il nostro approccio per far fronte a questo problema, dall'altro analizziamo l'agente in modo da comprendere meglio la natura del non-determinismo. Troviamo tre possibili modi per riprodurre un cluster e per ciascuno evidenziamo pregi e difetti.

Thanks

Firstly, I would like to thank professor Francesco Amigoni, professor Daniele Loiacono, and Dr. Davide Azzalini for the opportunity to work on a video game related topic, for always being available to talk in person about the latest developments, and for making this mandatory thesis a challenging and interesting opportunity to learn new concepts and tools.

I would also like to thank all my friends, from middle school to university; their willingness to spend their time with me is a precious gift.

Lastly, I would like to thank my brother for tolerating me all these years, and my parents, who have always supported me in any decision, even when it meant wasting time and money. Without them I would not be where I am now, and for this I will always be grateful.

Edoardo Zini

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today, new data are constantly generated and collected. This process happens in a wide variety of fields, including video games. Software houses working on a new title, or looking for critical issues in an already published one, rely on data to improve their products. Before a game is published, data are usually gathered via playtests. In a playtest, a group of players is allowed to play a game before it is published, these players usually provide their feedback via surveys created by the development team; in addition to the surveys, all actions performed by the players in the game are monitored and recorded. Among these actions, the paths followed by the players while exploring the virtual environment are particularly interesting in order to identify critical spots, to understand how players navigate the virtual environment, and, ultimately, to design better environments. When considering several players, it is likely that they can be clustered in different groups, for example seasoned players or more casual ones.

Our goal in this thesis is to group players together based on their skills and on the trajectories they followed in exploring an initially unknown virtual environment, which means that we must be able to cluster trajectories based on appropriate metrics. Once the clustering is done, our next goal is being able to generate a new trajectory that is similar to those belonging to a given cluster of human trajectories. In order to do so, we must devise an autonomous agent able to explore an unknown virtual environment. Achieving this goal would decrease the need for new playtests, increasing the speed at which software houses can develop their products.

Our work leverages on what was done by Simone Lazzaretti and Yuan Zhan (from now on L&Z) in their master's thesis [2]. They were able to deploy an autonomous agent that reproduces a given trajectory in the closest way possible. We want our autonomous agent to be more flexible and able to

generate a new trajectory that looks like those followed by human players of a certain ability. Since we want to be able to change the ability level of the humans to imitate, we must be able to cluster together players with similar abilities. Our contribution is inserted in the literature about clustering of paths and trajectories. We find that most papers, like [1], [3], and [4], share the same metrics and clustering techniques, with the exception of [4] that leads us to [5]. All these possible metrics have to be tested in order to understand if they are appropriate for our needs.

In Chapter 2 we recap what was done by L&Z. In Chapter 3 we state our goals in more details, we report the state of the art as far as clustering of trajectories is concerned, and we illustrate which additions are performed to L&Z's work in order to meet our needs. In Chapter 4 we tackle the clustering of human trajectories, we present different techniques, and for each one that does not work we highlight its drawbacks and limitations. In Chapter 5 we tackle the reproduction of trajectories by an autonomous agent; we present several techniques and for each one we discuss its pros and cons. In Chapter 6 we summarise our results, we present issues that we encountered and potential future developments. Lastly, in Appendix A we list all acronyms used throughout the thesis, followed by the definitions of the words that are mentioned by not explained in the text; in the remaining Appendices B, C, and D we document the tools and the code used in this thesis.

Chapter 2

Previous work

This thesis carries on the foundation laid out by Simone Lazzaretti and Yuan Zhan (from now on L&Z) in their master's thesis *Simulating Human Behaviour in Environment Exploration in Video Games* [2], for this reason we have to recap what was done by L&Z.

2.1 Goals

L&Z's main purpose was the development of a software framework capable of collecting data from human players while they explore an unknown virtual environment with limited visibility, analysing them and devising an autonomous agent able to reproduce a given human trajectory. A trajectory is the path an agent followed when it explored the map. In order to reach their goal L&Z started from a pre-existing Unity project designed to test procedurally generated video game maps, and they expanded it to suit their needs.

2.2 Pre-existing Unity project

The pre-existing Unity project L&Z worked with offers several useful tools: it allows the runtime generation of a 3D map from a .txt file containing the layout of that map. No different heights are allowed and all maps have a flat floor, this allows the **map** to be **defined as an $n \times m$ matrix** where each cell can take only one value. Two formats are allowed: char-based and numeric-based. The idea behind them is the same: the .txt file contains several rows, each one made up of several symbols separated by commas, each symbol represents the value of that map cell. The possible values for the symbols are shown in Table 2.1. In addition to the runtime creation of

Name	Meaning	Char-based values	Numeric-based values
Starting location	Where the agent is spawned.	s	3
Wall	Non-transparent insurmountable object. It works exactly like a real wall.	w	1 (or 5, 6,... for coloured walls)
Walkable	Flat empty surface where the agent can move freely.	r	0
Unknown	Area of the map where the agent has never been, as such they are not able to tell the content of that map cell.	u	2
Target	Walkable map cell containing a target the agent must get close to.	g	4

Table 2.1: Legend of symbols in a map .txt file.

a map from a .txt file, this project can also collect data on agent movement and spawn the agent in the map with one or more weapons. Due to the nature of their thesis, L&Z disabled all weapons related functionalities.

2.3 Testing environment

L&Z created two testing environments inside Unity: one for the human players and one for the autonomous agents that from now on will be called robots. In both cases the goal of the agent exploring the environment is the same: find all targets in the map. They designed four maps and a final survey. The **four maps** are:

- *open1*: map with perpendicular corridors, it features some open areas where the agent can find itself in a spot where no wall is visible; it contains only one target.
- *open2*: map with non-perpendicular corridors, it features some open areas where the agent can find itself in a spot where no wall is visible; it contains four targets.
- *uffici1*: map with perpendicular corridors, it features several rooms and recesses which create an office-like environment; it contains four targets.

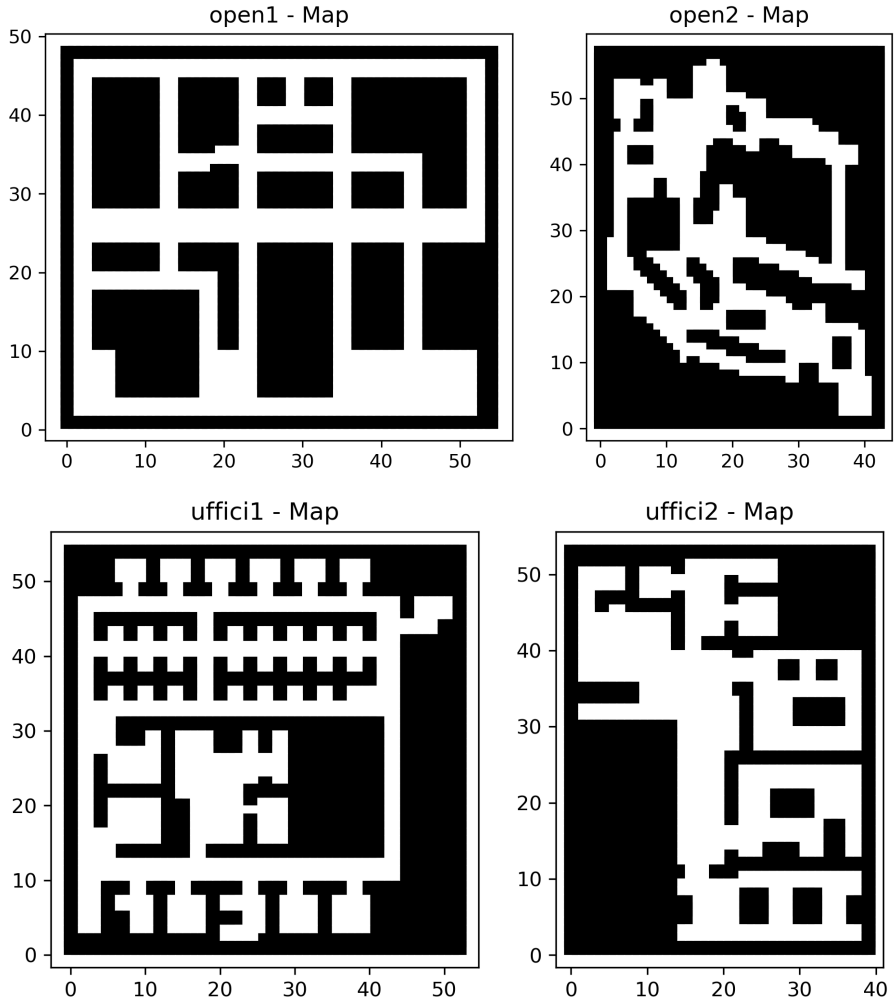

Figure 2.1: Map layouts.

- *uffici2*: map with perpendicular corridors, it features some rooms and recesses which create an office-like environment; it contains only one target.

You can see the map layouts in Figure 2.1. The final survey is shown only to human players and it can be found in the Section 2.4.1.

In order to ease the playing experience L&Z opted for a **WebGL release** that can be played in a compatible browser, this way only a link to the server hosting the game is needed in order to access and play it. Alternatively, the folder containing all the game files can be sent via email, given that it weights only few megabytes.

2.4 Data collection

Any agent, whether human or autonomous, **is periodically sampled** so that its position and orientation, also called rotation, are recorded. The time interval between each sample can be changed in the Unity inspector, by default it is set to 1.0 second. The series of collected positions creates a trajectory. A trajectory is a series of coordinate pairs that represent points on a plane, in particular the path the human followed when exploring the map is the one that the robot should imitate. In addition to position and rotation the following data are saved as well for both humans and robots: name of the map where the experiment takes place; time required for exploration completion; OS name and version; IP address. More data are collected, but since they are different between humans and robots they are listed and explained in the next sections.

2.4.1 Human data collection

L&Z's goal requires the collection of as much data as possible for all maps, for this reason they divided the four maps in two groups, for the purpose of this thesis we will call them *original* and *alternative*. Then L&Z deployed a web server in charge of providing each player with the least played group. The **maps are split in this way:**

- *original*: open1, uffici1.
- *alternative*: uffici2, open2.

This way each group contains first the map with only one target, then the map with four targets. Number of targets aside, the group structures are specular: one group starts with the open map, the other with the office-like map; in one group the map with four targets resembles an office, in the other group it resembles an open environment.

After completing each group the player is given a **survey** containing the following questions and possible answers:

1. How many hours per week do you play games?
   (a) Less than 1 hour.
   (b) Between 1 and 3 hours.
   (c) Between 3 and 7 hours.
   (d) More than 7 hours.

2. Do you often play First Person Shooter games?
   (a) Yes.
   (b) Never.
   (c) Used to.
3. What amount of area of the last map do you think you have explored?
   (a) Less than 50%.
   (b) Between 50% and 75%.
   (c) More than 75%.
4. Now think about the last map you have played. Which one of these maps was it?
   (a) [correct image]
   (b) [wrong image]
   (c) [wrong image]

In order to **limit player visibility** a fog of war is used. This means the player is at the center of sphere, everything inside this sphere is lit and can be seen while outside objects are not visible; however, each cell containing a wall has an embrasure at the top which is always visible from any distance, this means players may use those embrasures to orient themselves in the map.

Before exploring one of the two groups of maps each player must complete a s-shaped **tutorial map** to ensure they have understood how to move and what their goal is. Players are allowed to spend as much time as they need in order to find all targets. If a player quits before all targets have been found their Result will not be uploaded to the server.

2.4.2 Conclusions on human Results

L&Z relied on 12 tester to gather a total of 28 human trajectories:

- 9 human trajectories for map open1
- 9 human trajectories for map open2
- 4 human trajectories for map uffici1
- 6 human trajectories for map uffici2

Combining the trajectories with the answers to the survey they found that:

- Humans often underestimate the amount of area they explored.
- Experienced players require the same time as unexperienced players to find the targets...
- ...but they get a better mental representation of the map, either because of good memory or because of a higher ability to extract information from what they see.
- Indoor office-like maps with more walls are easier to explore w.r.t. open maps.
- Players navigate near a wall almost all the time.
- Players tend to keep the same direction.

2.4.3 Robot structure

A robot is a game object inside Unity composed of two parts: a head and a body. The head is where the virtual camera is located, this way it is possible to watch the virtual environment from the robot's point of view. The body is a parallelepiped surrounded by **proximity sensors**. The parallelepiped is used by Unity to ensure the robot does not enter a wall, while 270 sensors are placed on its front and sides in order to cover a 270° angle. **Each sensor shoots** an outward facing **ray** that stops after a given distance. This distance is meant to simulate the fog of war that prevents human players from seeing too far into the distance. Rays can be seen in Figure 2.2 where the blue ones are on the left hand side of the robot, the green ones on the right hand side and the red one between the other two groups is the front of the robot. These sensors are the only way the robot perceives the environment: if a ray hits a wall then the robot updates its internal map representation by putting the wall symbol on the cell located where the ray was interrupted; if the ray hits a target then the robot performs all the necessary actions in order to immediately reach that target; if the ray hits nothing then all the cells between the robot position and end of the ray are marked walkable. The choice of modelling the autonomous agent this way was deliberate: the idea was to design it so that it would work like a real robot, even tough without the intrinsic imprecision of a real world environment.

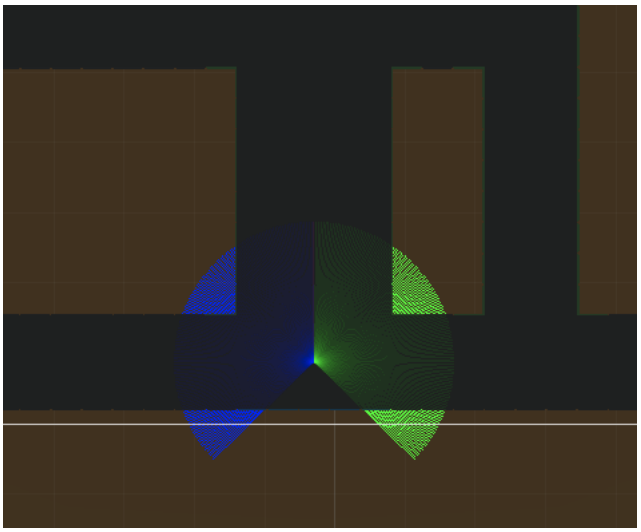

Figure 2.2: Robot's rays.

The robot uses its map representation to decide where to go next. Each known cell adjacent to an unknown cell is part of a frontier. Groups of adjacent cells are bundled together to create a frontier. For each frontier a centroid is computed, this point will be used by any policy that needs to compute the distance between that frontier and another position. Several policies can be used to decide which frontier should be explored next. In their work L&Z presented more than one **policy**, however we focus our attention only on the last one called ***RobotDMUtilityCloseWall***, which proved to be the best. **This policy relies on two parameters** to make a decision: α and β. $\boldsymbol{\alpha}$ is used as weight of the distance between the frontier and its closest wall; this distance is important since humans are less likely to explore areas that are far away from walls, given that by doing so they would lost sight contact with a point of reference, thus increasing the chances of getting lost. $\boldsymbol{\beta}$ is used to weight the inertia of the robot movement, this matters since humans are not likely to reverse the direction of their movement. A combination of α and β is used as weight of the distance between the robot and the frontier point. The chosen frontier is the one that minimises the utility as defined in Algorithm 1, where *Distance(frontier, wall)* is the Euclidean distance between the frontier centroid and the closest wall to that centroid among all walls known by the robot. This policy entails that the robot makes **greedy decisions**, so it may not behave in the most efficient way possible; furthermore, decisions about where to go next are taken periodically, thus a robot may go back and forth if it constantly changes its mind.

Algorithm 1: Utility

Utility(α, β)
- **if** $1 - \alpha - \beta \geq 0$ **then**
  - **return** $(1 - \alpha - \beta) \times Distance(robot, frontier) + \alpha \times Distance(frontier, wall) + \beta * Inertia(robot, frontier)$
- **else**
  - **return** $(1 - \alpha_{Norm} - \beta_{Norm}) \times Distance(robot, frontier) + \alpha_{Norm} \times Distance(frontier, wall) + \beta_{Norm} \times Inertia(robot, frontier)$
- **end**

end

Distance(a, b)
- **return** $EuclideanDistance(a, b)$

end

Inertia(robot, frontier)
- **switch** *frontier point position w.r.t. robot* **do**
  - **case** *in front* **do**
    - **return** *0*
  - **case** *sideway* **do**
    - **return** -50
  - **otherwise do**
    - **return** $+50$
  - **end**
- **end**

end

$\boldsymbol{\alpha_{Norm}}$
- **return** $\frac{\alpha}{1+\alpha+\beta}$

end

$\boldsymbol{\beta_{Norm}}$
- **return** $\frac{\beta}{1+\alpha+\beta}$

end

Once the next frontier to be explored is chosen a robot component checks if it can be immediately reached, if not a **theta*** algorithm is used **to compute the best path** to reach that frontier.

In addition to α and β, **a third parameter**, called $\boldsymbol{\delta}$ or **forgetting factor**, is used to periodically reset the status of a cell in the robot map representation. When a cell in the map representation changes from unknown to known a timer starts, after an amount of seconds equals to the forgetting factor that cell is set back to unknown. In practice a higher forgetting factor means a robot with more memory of the previously observed map.

2.4.4 Robot data collection

In L&Z's thesis finding the appropriate values of α, β, and δ yields a robot behaviour that closely resembles a given human trajectory, for this reason also these three values are saved together with all other collected data. L&Z found the best values via **grid search**. α and β can take any value between 0 and 1, the chosen increment is 0.1, thus the following pairs are tested: (0.0, 0.0), (0.1, 0.0), (0.2, 0.0),... (0.9, 0.0), (1.0, 0.0), (0.0, 0.1),... which means 11×11, i.e., 121, pairs. Each pair must have an associated forgetting factor, L&Z chose the following values: 30, 60, 120, 180. Four possible values for δ means a total of $11 \times 11 \times 4$, i.e., 484, parameter combinations.

In order to prevent a robot from being stuck in a loop a maximum of 420 seconds is given, once that time limit is reached the robot exploration is automatically terminated and the robot Result is deemed useless.

2.4.5 Data processing

Once the data have been collected they must be processed in order to find the robot that best resembles the human trajectory of interest. The comparison between a human path and each robot path is performed using a **metric**. L&Z tested several metrics, for the purpose of this thesis only three metrics are considered: Euclidean, longest common subsequence, and dynamic time warping.

Euclidean computes the Euclidean distance between the i-th position of one trajectory and the i-th position of the other trajectory, then all these distances are summed up to get the final overall distance. The robot trajectory with the lowest overall distance is selected. It is important to highlight that this technique can only be used when the two trajectories have the same length.

Longest common subsequence, or LCSS, checks any subsequence of one trajectory against any subsequence of the other trajectory. Two sequences are considered to be in common if all their points are close to one another, i.e., within a certain assigned distance. The longer the subsequence, the higher the amount of positions it contains and the higher the LCSS value. In practice the LCSS value is the length of the subsequence. This means two trajectories may represent the same exploration path only partially and at different time instants while keeping a high LCSS value. However, there is a drawback: the time-based nature of the sampling process, where the i-th position is sampled after the same amount of time for both trajectories. This means that identical paths exploring the environment at different speeds result in two different sequences that LCSS no longer consider in common after a while. LCSS can handle trajectories of different lengths w.r.t. one another.

Dynamic time warping, or DTW, checks similarities between two trajectories in a similar fashion w.r.t. LCSS, however DTW can manage different exploration speeds by warping time. This means that if two trajectories are identical in shape, but not in time, DTW will be able to compensate for this difference and recognise the two trajectories as similar. DTW has proved to be the best metric according to L&Z's results.

All the code needed to compute the distance between two trajectories is implemented in Python. In the end we choose the robot trajectory with the lowest (or highest in LCSS case) distance.

2.4.6 Conclusions on robot Results

L&Z tested all robot parameters according to the grid search described in Section 2.4.4. They discovered that most of the Results obtained in maps with multiple targets had to be discarded, but among the remaining ones **DTW is the best metric** to identify a robot trajectory that best resembles the chosen human trajectory. Furthermore, better Results can be found when α and β are different from one another, but this is not a strict rule. Unfortunately, they found no way of determining an optimal direction to be explored while looking for potentially better parameter values.

Chapter 3

Goals and state of the the art

In this chapter we explain the goals of the whole research project, the goals of this specific thesis and how we leverage what was done in previous theses.

Then, we analyse the state of the art as far as clustering methods are concerned, by looking for the most effective methods and metrics.

Finally, we take into account which improvements are performed to the existing work overviewed in Chapter 2. These improvements are the basis on which what is explained in the next chapters is built upon.

3.1 Goals

The overall goal is implementing an autonomous agent able to move in a virtual environment in a similar fashion w.r.t. what human beings would do. This goal requires several steps to be accomplished, some of them were already completed when this thesis started, while some were not. What was already done is the result of Simone Lazzaretti and Yuan Zhan's master's thesis [2] and it is laid out in Chapter 2. It is important to highlight that L&Z's goal was to reach a state where all the elements of the system, from those that collect data on human behaviour to those that drive the robots around, were implemented and operational. Inevitably, this led to some implementation choices that are not the most efficient ones.

This thesis carries on L&Z's work by improving it on two fronts: first by moving from imitating one human trajectory to a **cluster of human trajectories**, which means moving from a geometric-based metric to a **higher level metric** to measure the quality of the exploration; then by looking for **more efficient ways of getting the appropriate robot parameters**. In details, this means implementing a clustering procedure able to properly separate groups of humans based on their abilities, which requires

the development of new metrics able to distinguish different ability levels. Furthermore, the robot parameters optimisation process developed by L&Z is single threaded and relies on a brute-force approach. Looking for more efficient ways would be useful to speed up the overall procedure.

3.2 State of the art

Clustering trajectories is a known topic in data mining, several papers have already been published, thus we check which method has proved to be the most effective. Furthermore, some additions to L&Z's work are performed in order to have a more scalable basis for this thesis to work on.

3.2.1 Clustering

In [1], authors compare the metrics shown in Table 3.1, where $\boldsymbol{n}$ is the total number of points of trajectory 1, $\boldsymbol{m}$ is the total number of points of trajectory 2 and **unified length** means $m = n$. After that, they take into consideration some densely clustering methods: DBSCAN, adaptive multi-kernel based, k-means, expectation maximisation, fuzzy c-means; and some hierarchical clustering methods: agglomerative and divisive. They discover that densely clustering methods classify trajectories mainly by distance metrics, thus weighting too much the spatial information of the trajectories. **Better Results are obtained using hierarchical clustering** methods which have shown to take into account more attributes. As far as the metrics are concerned, **LCSS has proved to be the best**, also due to its ability to cope with trajectories of different lengths.

In [3], authors discover that the **clustering method has a small impact** over the quality of the final result, while the **distance measure has a much greater influence**.

In [4], authors compare mostly the same metrics shown in Table 3.1, with a noticeable addition: Piciarelli-Foresti (PF). They discover that LCSS is more robust to noise and outliers than DTW, and they highlight the fact that **PF can work with incomplete trajectories**. All metrics we have considered so far require completed trajectories, which means that a robot exploring the environment must end its exploration before we can assess the similarity of its exploration w.r.t. the desired trajectory; PF removes this limitation. As far as the clustering methods are concerned, again, Brendan Morris et al. compare mostly the same metrics used by Jiang Biana at al., but they take into account different types of trajectories. We focus on the results

Metric	Required unified length	Complexity	Explanation
Euclidean	Yes	$\mathcal{O}(n)$	It measures the contemporary instantiations of trajectories.
Hausdorff	No	$\mathcal{O}(m \times n)$	It is the greatest distance between any point of one trajectory and its closest point on the other trajectory.
Bhatta-charyya	Yes	$\mathcal{O}(n)$	It measures the similarity of two probability distributions, this means each trajectory is treated as a series of samples from a distribution.
Frechet	No	$\mathcal{O}(m \times n)$	It measures the similarities between two curves by taking into account location and time ordering.
DTW	No	$\mathcal{O}(m \times n)$	It is a sequence alignment method to find an optimal matching between two trajectories without considering lengths and time ordering.
LCSS	No	$\mathcal{O}(m \times n)$	It finds the longest common subsequence in all sequences and its length is the similarity between the two trajectories.

Table 3.1: Metrics comparison in [1].

of the human trajectories: **LCSS is the best metric**; **agglomerative hierarchical clustering is the best clustering method**.

Given the potential usefulness of Piciarelli-Foresti we look for more details about this metric. In [5], Piciarelli and Foresti describe their technique. The idea is to map a trajectory on a tree of nodes obtained by applying PF to each trajectory. A node can be considered as the area containing a group of points belonging to one or more trajectories. The root node of the tree is located where the trajectory starts, this means that if we considered more than one trajectory then multiple trees with different starting points might be created. As the trajectory goes on it is mapped onto the tree nodes on a distance basis. Each point of the trajectory is assigned to the closest node, the newly assigned point is used to update that node, this means that as time goes by a node will change. Similar trajectories take the same path across the various nodes, but when a trajectory gets too far from an already

existing node a new node is created, and the older node is either kept or split depending on whether the new node is placed at the end or in the middle of the older one. Periodically, or at the end, a tree maintenance procedure is applied in order to merge nodes that are close to one another, or nodes that, due to how the tree evolved over time, end up being a unique series of consecutive nodes. This concatenation process can only happen if pruning is used. The older the trajectory the less important its points, thus it is useful to weight those points taking time into account. Pruning ensures that points belonging to older trajectories are discarded, and this may cause an entire series of nodes to disappear, potentially eliminating crossroads and resulting in a branch composed only of consecutive nodes.

Despite its useful features **PF has some drawbacks**, in [5] it is applied to trajectories of cars on a highway exit, this makes those trajectories extremely similar due to the few possibilities a car has: always go forward and never pass on the same spot twice. These conditions are perfect for the anomaly detection goal considered by Piciarelli and Foresti, but they are the opposite of what we are dealing with in our project. Furthermore, even if the paper [5] contains a detailed description of PF, it does not precisely explain how each and every step should be implemented. This, in addition to the lack of an already implemented solution, the drawbacks mentioned before, and the fact that PF would be heavily geometrically based, make us **decide to avoid using PF**.

After reading papers about trajectory clustering we find out that most of them take into account the same metrics, and they agree on considering **LCSS the best** one, however they also point out that the performance of a metric is dataset-dependent. In particular, all papers consider trajectories that never pass on the same location twice or more, since their human trajectories represent people moving from point A to point B in the most efficient fashion, avoiding unnecessary turns. These papers also agree on **considering the metric far more important than the clustering method**.

Previous papers suggest agglomerative hierarchical clustering as a potential candidate for the clustering method, however, we decide to compare some techniques to assess their requirements. We compare hierarchical clustering, k-means, BFR, mean shift, expectation maximisation, and density-based DBSCAN. Based on [6] we concluded that k-means, BFR, and expectation maximisation should be discarded due to the need of knowing in advance the desired number of clusters; mean shift requires us to properly size the window, this would be an additional task, so this method is discarded too; density-based DBSCAN not only requires us to provide two parameters (radius ϵ and

MinPts), but it may fail if there are major density fluctuation in the data to be clustered, thus it is discarded as well; **hierarchical clustering** can work without additional parameters, and the previously defined metrics are ideal for the computation of a distance matrix among our trajectory samples.

In conclusion **agglomerative hierarchical clustering is the chosen clustering method**, since it has proved to be effective and it does not require additional input parameters.

3.2.2 Additions to previous work

L&Z's work can perform all the steps from collecting human data to identifying the best robot parameters. We decide to improve some of those steps in order to better suit our needs.

General additions

Firstly we **add distance-based sampling** to the already implemented time-based sampling described in Section 2.4. The idea is to sample position and rotation of the agent every n meters, actually units are used instead of meters since we are working within the Unity coordinate system. The actual value of n can be set in the Unity inspector of Player and Robot, more on this in Appendix B. This means that, given two trajectories, the i-th sample of the first trajectory corresponds to the i-th sample of the second trajectory in terms of walked distance, but likely not in terms of elapsed time. This can be useful in order to remove the temporal element, and thus the exploration speed of each agent. An average speed can still be computed by dividing the overall walked distance by the completion time.

Then we change the **amount of time** a robot is given **before the exploration is automatically terminated**, we increase it from 420 to 480 seconds, i.e., from 7 to 8 minutes. We also **add this time limit to human players**, since L&Z gave humans no limit. With these changes we can now consider valid all robot trajectories, since we are likely to collect trajectories where the agent has not found all targets for both humans and robots.

Humans-related additions

Concerning the experiments involving **human players**, we change both the front-end and the back-end. L&Z needed as many trajectories as possible for each map, while we need as many trajectories as possible for a single

map. For this reason the Heroku server they used as back-end is no longer used. It is replaced with a direct upload of the experiment data to a newly established Firebase Realtime Database, then the data can be downloaded as a unique json file from the Firebase web interface. This file containing all data is converted into multiple files, one for each trajectory, by the Python script *FirebaseAdapter.py*. Getting rid of the Heroku back-end requires a new way of selecting which map group should be played. Our goal is to maximise the amount of collected data, ideally **all players should play the original group of maps** (Section 2.4.1), then the most willing players should be allowed to play the alternative group either immediately or at a later date, without the need to replay the original one. This means either **leaving the choice of which group to play to the final user** or implementing a user identification system that would require a login. The latter poses additional work for us, and also increases friction for the users. Thus we decide to leave the responsibility to the final user, however some **countermeasures** are implemented.

Firstly, we need to consider that our testers, those receiving the link to play the game, are relatives, friends, or other students. The latter receive a mail explaining the experiment and asking them to complete it, however they are not forced to play, which means only those who are interested will play it. Furthermore, we can assume a higher than average computer literacy, given that they are all computer science and engineering students. As far as the other testers are concerned we have direct contact with them, so any issue can be directly addressed. All of this is to say that players can be provided with a detailed explanation about how to procede (e.g., disable any ad-block, change your browser) and **we can be confident in their ability to follow those instructions**.

Secondly, a **safety net** is implemented **in UX and UI**: if a player skips all instructions and just presses the *next* button they will end up pressing the *play originals* button located in the same spot, as shown in Figure 3.1. Furthermore, the *play originals* button is bigger than the other ones to let players instinctively know that that is the button to press. This means players are likely to automatically do what they are supposed to do: play the original group first. At the end of the survey for the first group of maps a message is shown asking the player to play the second group.

The **game is provided** to testers **as a link to a Firebase Hosting instance** that lets people play the game in their browser, so no installation is required, streamlining the process. In case of necessity we can update the game with virtually no latency: we can push a new build to Firebase Hosting and within few seconds this new build is delivered to anyone opening the link.

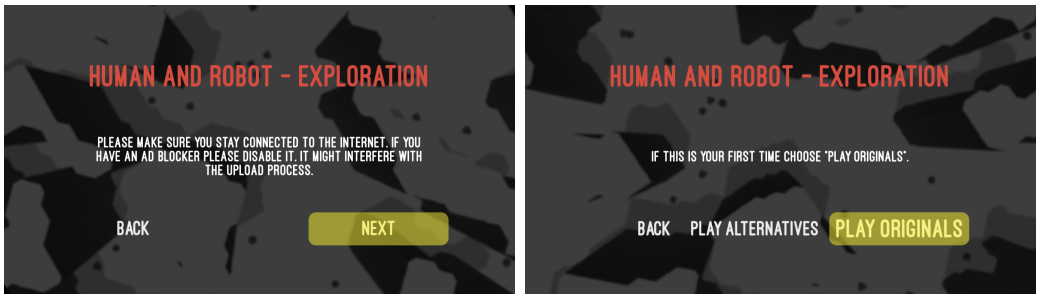

Figure 3.1: Experiment instructions (yellow rectangle superimposed to highlight the same spot on both images).

This was not the case with a possible alternative to Firebase Hosting: simmer.io (`https://simmer.io`). Simmer.io is a YouTube-like service for WebGL games. It has the advantage of letting us add a description text that can be updated independently from the game, and its similarity with YouTube makes it more user-friendly. However, it has proved to have severe latency when it comes to pushing an updated build to people loading the game page; for this reason it is not used.

Robot-related additions

During the **grid search** (Section 2.4.4) L&Z tested one robot parameter combination at a time using a real time simulation. In order to speed up this process we add the possibility to **change the time scale** via the Unity inspector of the Exploration Iterator, the component in charge of handling the grid search, more on this in Appendix B. However, the way the robot is implemented, combined with the Unity engine, makes changing the time scale unfeasible. The reason is the following: the robot can either go forward or rotate until it faces the current destination. It is aware of where that destination is w.r.t. itself, so it knows which rotation direction is the best to face the destination. In order to check whether it is facing the destination or not, a forward facing ray is shot from the front of the robot. When this ray collide with a parallelepiped collider representing the destination the robot stops rotating and starts to move forward. The collision check is performed once per frame. Increasing the time scale means the elapsed time between two frames is increased, thus the robot might turn too much between two frames and overshoot its destination. This would result in the robot starting to rotate on the opposite direction, but then it would overshoot it again, and so on. The robot would get stuck; for this reason we choose to keep the time scale to 1.0, i.e., real time.

In addition to time scale changes, we aim at **parallelising the robot exploration** by having **multiple robots at the same time**. L&Z's implementation was designed to destroy and recreate the Unity scene each time a new robot is spawned, we change it so that the scene is kept after a robot ends its exploration. Furthermore, we change how the target positions are handled so that it is possible to move a target and have its corresponding position updated in real time, previously even if the target was moved, the game logic would still consider its original spawn point as target location.

It is possible to change the increment used during the grid search from the default 0.1, but it must be done directly in the Exploration Iteration C# code. Since 0.1 is not perfectly convertible into a binary number, summing 0.1 multiple times could result in a wrong number. So we add a rounding phase after each summation in order to minimise the error.

Lastly, after some tests we decide to **change the forgetting factors** from the original 30, 60, 120, 180 to 30, 90, 180, 360. Since α, β, and their increment are not changed, the total number of possible combinations is still 484, but the maximum memory of the robot has been raised from 180 to 360 seconds.

3.2.3 Voronoi

L&Z's code works on the original trajectory positions. Each position of the agent is approximated to the closest integer in the $n \times m$ map matrix (Section 2.2), which means that two agents walking down the same corridor are likely to be close to one another, but not exactly on the same map cell. This difference is negligible for our purposes and we would like to get rid of it. This can be done by **converting each position to the corresponding Voronoi equivalent**. Voronoi points are all those points that are equidistant from their two closest obstacles. In our case, walls are the obstacles, thus, for example, the Voronoi points of a corridor are all the points on the axis of that corridor, as we can see in Figure 3.2.

Figure 3.3 shows the map layout, followed by all Voronoi points in orange. These points can be found inside walls too, this is because walls are defined over an integer grid, as such the space between each integer value can contain valid Voronoi points. This is why we have to filter out unwanted Voronoi points, which results in the image *Filtered Voronoi*. The last image overlaps the original map with the filtered Voronoi points to make it easier to understand how they are distributed on the map.

For each map, the list of filtered Voronoi points is saved; once the data collection is over, each point of each trajectory is mapped on to the closest

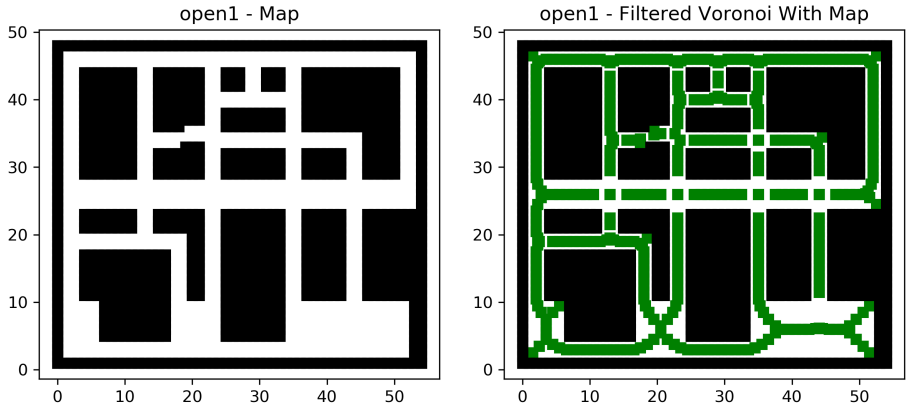

Figure 3.2: Voronoi points - map open1.

Voronoi point. At the end of this process the resulting trajectories are as long as the original ones, but without the noise caused by minor variations in the exploration: two agents walking down the same corridor will walk over the same spots. We acknowledge that this approximation does impact the local exploration speed: multiple distinct points might be mapped onto the same Voronoi point zeroing the speed, and a jump from two distant Voronoi points would represent a possibly unrealistic high speed spike. In our case, given that we consider only the overall speed, this is not a problem.

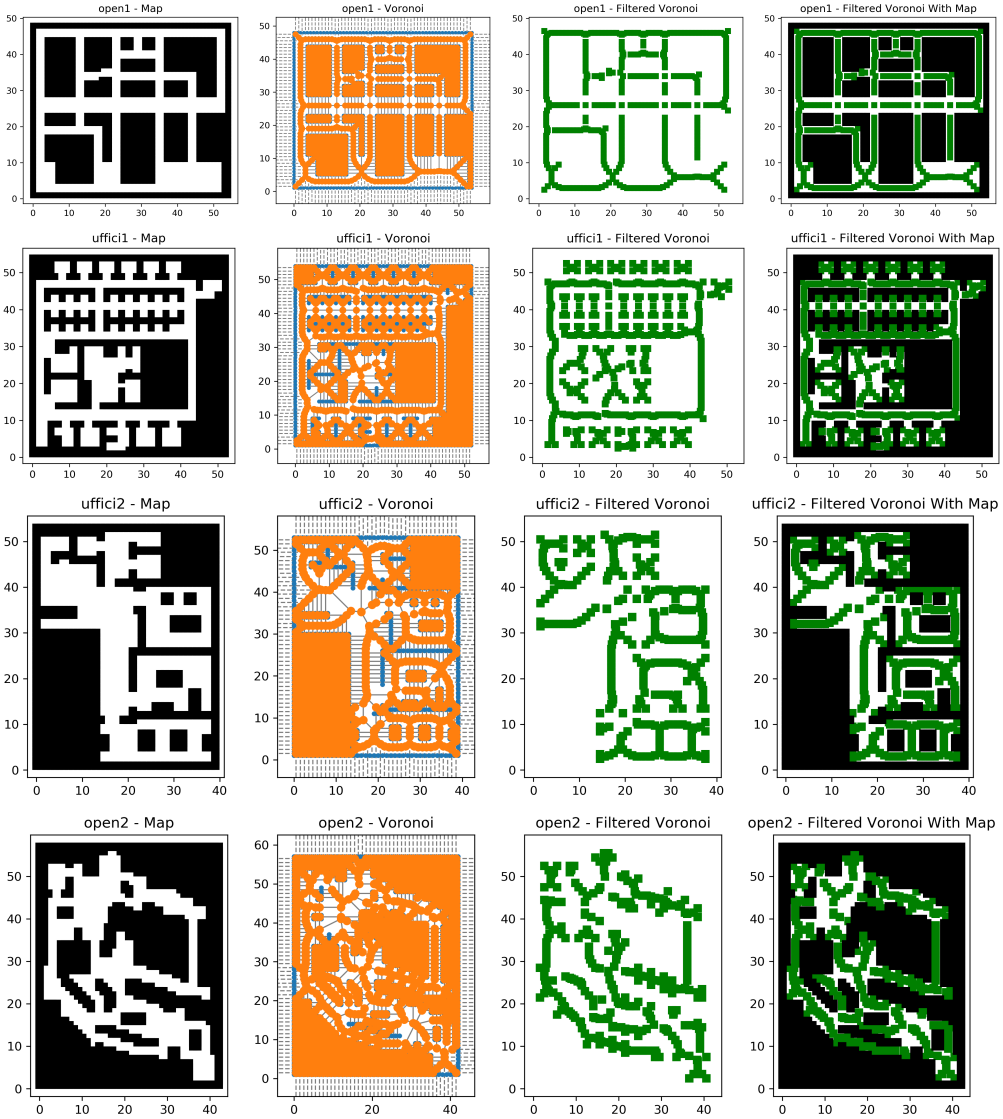

Figure 3.3: Voronoi points - all maps.

Chapter 4

Clustering

In this chapter we use the previously described metrics (Section 2.4.5) to perform agglomerative hierarchal clustering over a set of trajectories. We discover that those metrics are not appropriate for our purposes, thus we look for different approaches.

The first new approach is based on Hidden Markov Models (HMMs), which does not prove useful for clustering purposes; then we focus on the survey presented in Section 2.4.1, and its collected answers. Although they are not useful per se, it is from them that we take inspiration to define the measures that are used in the winning approach.

By using these measures as features we apply hierarchical clustering and this time we get results in line with our expectations. We end this chapter with a conclusion that sums up our results and highlights their limits.

4.1 Data collection

In order to perform a meaningful clustering procedure we need more trajectories than those collected by L&Z (Section 2.4.2), thus we implement all the additions described in Section 3.2.2, then we send the Firebase link to the testers. After few weeks **we collected the amount of trajectories reported in Table 4.1**.

4.2 SciPy

We choose SciPy (pronounced "*Sigh Pie*"), a Python library containing many scientific algorithms [7], including the ones to perform hierarchical clustering [8]. SciPy's hierarchical clustering requires the following input: the distances between each pair of samples, or the matrix containing the positions of each

Map	Total number of trajectories	Trajectories with survey
open1	102	83
uffici1	83	83
uffici2	40	34
open2	35	34
TOTAL	260	234

Table 4.1: Collected trajectories.

sample in a n-dimensional space; the *method* to be used when computing the distances between clusters. Seven **methods** are allowed:

- **Average:** given clusters u and v, their cardinalities $|u|$ and $|v|$, for all points i in u and j in v the distance is $d(u,v) = \sum_{i,j} \frac{d(u[i],v[j])}{|u|*|v|}$.
- **Centroid:** given clusters u and v, their respective centroids c_u and c_v, the distance is $d(u,v) = |c_u - c_v|_2$. When two clusters are combined the new centroid is computed over all the original samples in each of the two original clusters.
- **Complete (or max):** given clusters u and v, for all points i in u and j in v the distance is $d(u,v) = max(d(u[i],v[j]))$.
- **Median:** same as centroid but when two clusters are combined the new centroid is the average of the centroids of the two combined clusters.
- **Single (or min):** given clusters u and v, for all points i in u and j in v the distance is $d(u,v) = min(d(u[i],v[j]))$.
- **Ward:** given clusters s and t, cluster u as the combination of clusters s and t, cluster v as an unused cluster and $T = |v| + |s| + |t|$ where $|a|$ is the cardinality of cluster a, the distance is $d(u,v) = \sqrt{\frac{|v|+|s|}{T}d(v,s)^2 + \frac{|v|+|t|}{T}d(v,t)^2 - \frac{|v|}{T}d(s,t)^2}$.
- **Weighted:** given cluster u composed of clusters s and t, cluster v is a remaining cluster, the distance is $d(u,v) = \frac{d(s,v)+d(t,v)}{2}$.

After computing all clusters and drawing the full dendrogram we need a way to decide the appropriate number of clusters. In order to do so we implement the **Knee/Elbow analysis (KE)**, the idea is to compute **within-cluster sum of squares (WSS)** and **between-clusters sum of squares (BSS)** for any possible number of clusters, then we plot those values and choose the

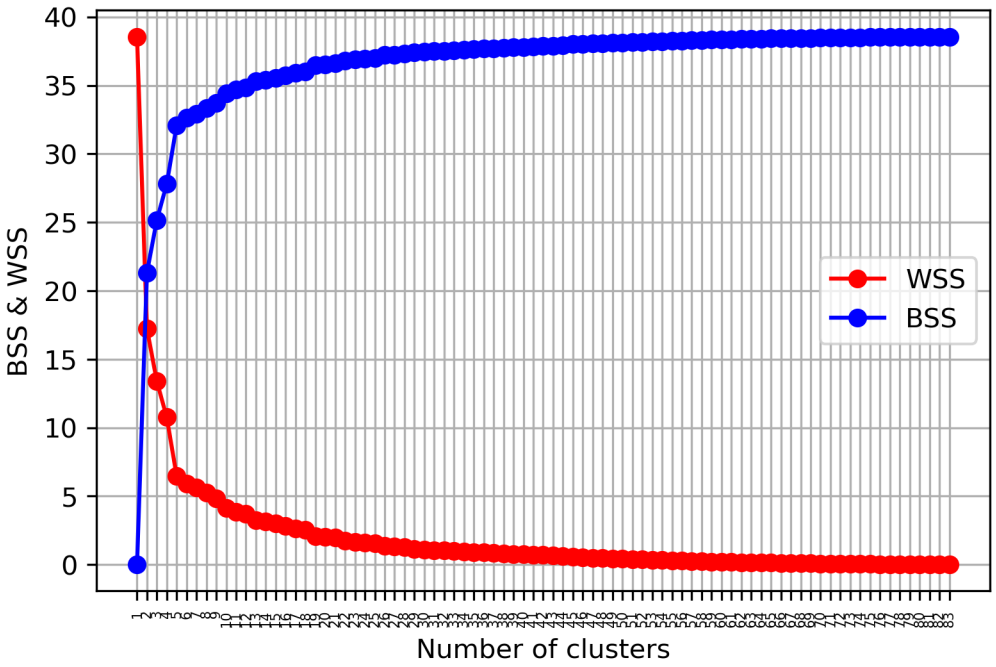

Figure 4.1: Example of Knee/Elbow analysis.

number of clusters where a knee or an elbow can be seen, e.g., 5 clusters in Figure 4.1. Ideally, we would like a low WSS and a high BSS. WSS and BSS formulae are:

$$WSS(C) = \sum_{i=1}^{k} \sum_{x_j \in C_i} d(x_j, \mu_i)^2 \qquad BSS(C) = \sum_{i=1}^{k} |C_i| d(\mu, \mu_i)^2$$

where k is the number of clusters; C is the list containing all k clusters; C_i is a single cluster from C; x_j is a single sample from cluster C_i; $d(a, b)$ is the distance between a and b; μ_i is the centroid of cluster C_i; μ is the centroid of C.

4.3 Metric-based clustering

4.3.1 Metric definitions

Even though according to L&Z DTW is the best metric (Section 2.4.6), according to several other works in the literature (Section 3.2.1) LCSS is the best one. We test both of them, in addition to the Euclidean (EU), to check which one better suits our goals. Since we need to provide the distances between each pair of trajectories, we have to solve two problems: firstly, not

all trajectories have the same length in terms of number of sampled positions; secondly, when working with distances, the lower the distance, the better, but that is not the case with LCSS, where the higher the LCSS, the better. In order to cope with these problems we consider the following metrics:

- **DTW** is exactly the same DTW defined in Section 2.4.5.

- **LCSS Lin** computes LCSS values, then it converts those values into distances via $1000 - LCSS$. We choose *1000* since it is higher than the length of longest trajectory, and it is a round number.

- **LCSS Norm** computes LCSS values, then it converts those values into distances via $\frac{2*LCSS(T_1,T_2)}{|T_1|+|T_2|}$, where T_1 and T_2 are two trajectories and $|T_i|$ is the length of trajectory i.

- **LCSS Inter Lin** computes LCSS values on the extended trajectories rather than on the recorded ones: each time a pair of trajectories do not have the same length, the shortest one is extended by making it as long as the other one; all the intermediate values are obtained via linear interpolation: if a trajectory A is composed of 10 positions, and a trajectory B is composed of 7 positions, then trajectory B is extended to 10 position, its first and last positions are the same as the original B, while the remaining positions are computed by linearly interpolating the 2 closest values in the original B. At the end the LCSS values are converted into distances via $1000 - LCSS$.

- **LCSS Inter Norm** uses the same extended trajectories as LCSS Inter Lin, but a conversion is done via $\frac{2*LCSS(T_1,T_2)}{|T_1|+|T_2|}$, where T_1 is the original trajectory and T_2 is the extended one, or viceversa.

- **EU** requires unified length, i.e., the same length for the two trajectories, if necessary we repeat the last position of the shortest trajectory until it matches the length of the longest one.

- **EU Inter** recalls the idea of interpolation: each time a pair of trajectories do not have the same length, the shortest one is extended by making it as long as the other one; all the intermediate values are obtained via interpolation.

4.3.2 Preliminary results

We rely on the 9 trajectories that L&Z collected for map open1 in order to compute some preliminary results. First we use SciPy to perform hierarchical clustering, then we use the Knee/Elbow analysis to check which number of clusters could be appropriate. This approach has two issues: the minor one is that we do not use a centroid in BSS and WSS formulae, nut we use a clustroid represented by the trajectory that is the closest to the other ones in that cluster; the main issue is that the total number of samples is just 9. We apply the clustering procedure to both the original trajectories and their Voronoi counterparts. We look for metrics that provide consistent results across the various methods, furthermore, we manually cluster the trajectories, each one called "Result" followed by a number, in order to compare our clustering results against what we get from the hierarchical clustering procedure. We report our results in Tables 4.2 and 4.3. If we consider the various methods we find that:

- **DTW** results in 2 clusters, providing an extremely stable result across almost all possible methods. It separates the two longest trajectories (Figure 4.2) from the other ones; however, according to our manual clustering, we would expect at least 3 clusters.

- **LCSS Lin** provides inconsistent results. Even when the cardinalities of the clusters are the same, different methods tend to provide different results. Furthermore, it does not always keep together Result4 and Result17, which are clearly meant to be together due to their strong similarity (Figure 4.3). This means that we cannot rely on this metric for appropriate clustering. The reason for this lies in how LCSS works: it looks for the longest common subsequence, so longer paths may have longer common subsequences despite being far more different than Result4 and Result17.

- **LCSS Norm** provides a mostly consistent result of 2 clusters, and all clusters are identical. Differently form DTW, the shortest paths (Result4 and Result17, Figure 4.3) are the ones that are kept together. This happens because they are very similar and, due to normalisation, they are the most similar. Other solutions with higher number of clusters are similar but not equal to the corresponding manual clustering solutions.

- **LCSS Inter Lin** provides inconsistent results that often have high cardinality, meaning that the number of clusters is almost equal to the

Original	AVG	CENTR	MAX	MED	MIN	WARD	WEIG
DTW	2	2	2	2	2	3 or 4	2
LCSS Lin	2 or 4 or 7 or 8	6 or 8	2 or 7 or 8	4 or 5 or 6 or 8	4 or 8	2 or 7 or 8	2 or 4 or 7 or 8
LCSS Norm	2 (or 8)	2 (or 8)	3 (or 6 or 8)	2 (or 8)	2 (or 8)	2 or 4 (or 8)	2
LCSS Inter Lin	4 or 7	8	2 or 5	8	3 or 7	2 or 6	2 or 4 or 7
LCSS Inter Norm	2	2	2	2	2	2	2
EU	2 or 3 (or 7)	2 or 3 (or 7)	2 or 3 (or 7)	2 or 3 (or 7)	2 or 3 (or 7)	2 or 3	2 or 3 (or 7)
EU Inter	2 or 3	2 or 3	2 or 3	2 or 3	2 or 3	2 or 3	2 or 3

Table 4.2: Numbers of clusters per method - original 9 trajectories.

Voronoi	AVG	CENTR	MAX	MED	MIN	WARD	WEIG
DTW	2	2	2	2	2	2	2
LCSS Lin	2 or 5	8	2 or 6	8	4 or 6 (or 8)	2 or 6	2 or 5
LCSS Norm	3 (or 8)	2 (or 8)	3 (or 8)	3 (or 6 or 8)	2 (or 5 or 8)	3 (or 8)	3 (or 8)
LCSS Inter Lin	2 or 5 or 7	$\emptyset$	2 or 5	8	3 or 6	2 (or 7)	2 or 4 or 7
LCSS Inter Norm	2 (or 8)	2 (or 8)	2 (or 8)	2 (or 8)	2 (or 8)	2 (or 8)	2 (or 8)
EU	2 or 3	2 or 3	2 or 3	2 or 3	2 or 3	2 or 3 or 5	2 or 3
EU Inter	2 or 3 (or 7)	2 or 3 (or 7)	2 or 3 (or 7)	2 or 3 (or 7)	2 or 3 (or 7)	2 or 3 or 6 or 7	2 or 3 (or 7)

Table 4.3: Numbers of clusters per method - Voronoi counterparts of the 9 trajectories.

Note: in all these figures the green dots represent the target.

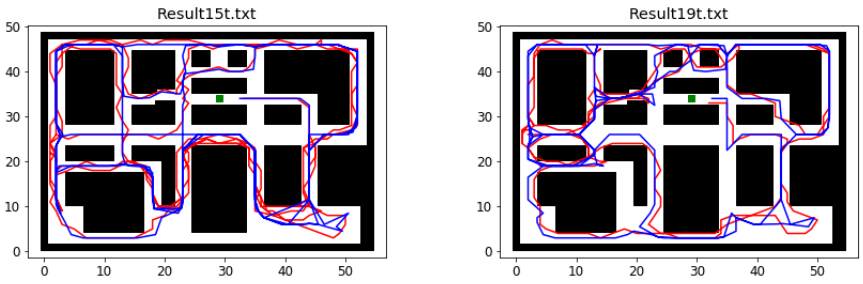

Figure 4.2: Result15 and Result19.

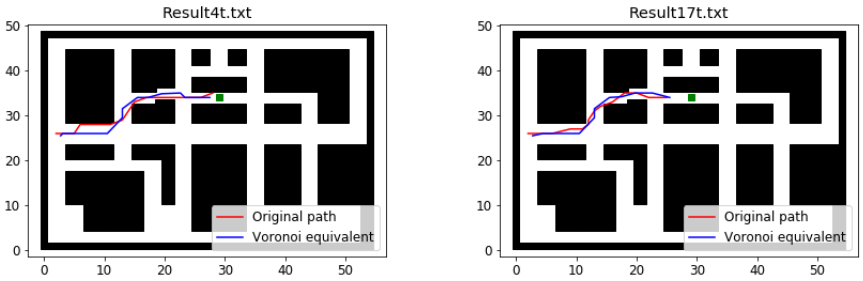

Figure 4.3: Result4 and Result17.

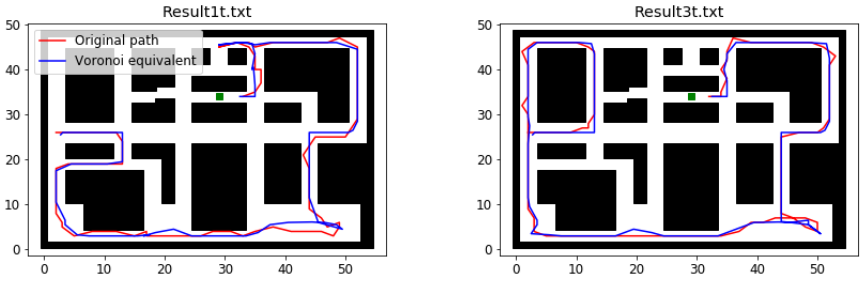

Figure 4.4: Result1 and Result3.

number of trajectories. Furthermore, this solution searates Result4 and Result17 (Figure 4.3), when they should clearly be grouped together.

- **LCSS Inter Norm** is extremely consistent: it always provides 2 clusters which keep Result4 and Result17 (Figure 4.3) together as expected.

- **EU** provides a mostly stable result: either 2 or 3 clusters where the 2 longest paths (Figure 4.2) are separated from all the other ones, and also from each other. If a cardinality is chosen then the content of the clusters is the same for all methods, making this alternative extremely consistent. However, none of these solutions is equal to the expected manual clustering solutions, even if they are both reasonable. Ideally, we should have low WSS and high BSS, thus the 2 clusters solution should be discarded since its WSS and BSS are almost identical and they are both high, while they are much farther apart in the 3 clusters case. Furthermore, by inspecting higher cardinality solutions we can see that the pairs Result4-Result17 (Figure 4.3) and Result1-Result3 (Figure 4.4) are kept together as expected.

- **EU Inter** is very similar to EU, with most of the results having cardinality 2 or 3, and the same clusters as EU.

Preliminary conclusions

For both original trajectories and Voronoi equivalents the best and worst metrics are the same. Good clustering metrics: DTW, LCSS Norm, LCSS Inter Norm, EU, and EU Inter. Bad clustering metrics: LCSS Lin, and LCSS Inter Lin. We can conclude that:

- None of the tested metrics and methods provided the same results as a manually-made clustering. This is likely due to the low number of samples and to the large differences between each other. Collecting more samples could confirm or reduce the set of good clustering metrics.

- DTW is stable and consistent, and provides good results for both original and Voronoi trajectories.

- LCSS should be used only in normalised form, whether directly or after interpolation does not seem to have a major impact, although it should be noted that LCSS Inter Norm provided more consistent results.

- Euclidean metrics are consistent and provide the same results between EU and EU Inter.

Voronoi	AVG	CENTR	MAX	MED	MIN	WARD	WEIG
DTW	13	14	9	4	8, 17	6	4

Table 4.4: Numbers of clusters per method - 93 trajectories.

4.3.3 Metric-based clustering results

We collect 93 trajectories, and for each one we have the distance-based sampled positions (detailed in Section 3.2.2), thus we switch to these new samples. The sampling frequency used by the distance-based sampling process is too high, in order to reduce the noise and make the trajectories more realistic we consider only their even positions when computing the Voronoi equivalents. All data are still available in the original files if needed, more on this in Appendices B and D. After checking our results we find that DTW is the most promising metric, its numbers of clusters can be seen in Table 4.4. We check the content of each one of the clustering results, and we discover that all clusters are different, with the solely exception of average and weighted. Furthermore, not only all clusters are different, but **looking at the trajectories we see that they are not clustered as we would like**. The cause of this is intrinsic in the way all these metrics work: they compare the positions that constitute each path, thus the geometry of the trajectories has a major impact over the final result. As stated in Section 3.1, we need to move from geometric-based metrics to higher level metrics able to distinguish among different ability levels, for example distinguishing players that explored the same areas only once from players who got lost and travelled over the same spots multiple times. In conclusion **the metric-based approach** does not provide good results and it **is discarded**.

4.4 Hidden Markov model based clustering

4.4.1 Definitions

Once we acknowledge the need for a new approach we decide to try Hidden Markov Models (HMMs) [9]. The idea is based on [10] where the authors apply HMMs to model trajectories on a road intersection, a highway, and a laboratory in order to perform anomaly detection.

Before detailing this approach we need to provide some definitions. As the name suggests, HMMs extend the concept of **Markov chains**, the components of a Markov chain are Q, A, and π and their definitions can be found in Table 4.5. The underlying idea is that we have several nodes, i.e., states,

Component	Definition
$Q = q_1 q_2 ... q_N$	Set of N **states**.
$A = a_{11} ... a_{ij} ... a_{NN}$	**Transition probability matrix** A where each a_{ij} represents the probability of moving from state i to state j such that $\sum_{j=1}^{N} a_{ij} = 1 \ \forall i$.
$O = o_1 o_2 ... o_T$	Sequence of T **observations**.
$B = b_i(o_t)$	Sequence of **observation likelihoods**, also know as **emission probabilities**, each expressing the probability of an observation o_t being generated from a state i.
$\pi = \pi_1, \pi_2, ..., \pi_N$	**Initial probability distribution** over states. π_i is probability that the Markov chain will start in state i. It is possible to have some states j where $\pi_j = 0$, i.e., they cannot be initial states. π is such that $\sum_{i=1}^{n} \pi_i = 1$.

Table 4.5: Components of a hidden Markov model.

and we have a certain probability of moving from one node to another, i.e., the transition probability; we use the initial probability distribution to pick which node is the starting one. Both Markov chains and HMMs rely on the **Markov assumption**: when predicting the future, the past does not matter, only the present is relevant. Formally: $P(q_i|q_1...q_{i-1}) = P(q_i|q_{i-1})$.

Hidden Markov models expand Markov chains and they use all components listed in Table 4.5. In a Markov chain we move from an observable state to another observable state, this means we know in which state we are at any given moment. In HMMs that is no longer the case, these models are *hidden* because we cannot observe the states directly, we can only see observations. The actual underlying process moves from one state to another, but since we only see observations we have to estimate which states are the ones involved. We use first-order HMMs, which means we are working under two assumptions: the aforementioned Markov assumption and the **output independence**, i.e., the probability of an observation depends only on the current state that produced that observation and not on any other states or observations. Formally: $P(o_i|q_1, ..., q_i, ..., q_T, o_1, ..., o_i, ..., o_T) = P(o_i|q_i)$.

Both Markov and output independence assumptions are needed in order to keep our model simple, but they do limit the modelling abilities of the HMM.

HMMs involve three basic problems [9]:

1. **Likelihood (forward algorithm):** given an HMM $\lambda = (A, B)$ and

an observation sequence O, determine the likelihood $P(O|\lambda)$.

2. **Decoding (Viterbi algorithm):** given an HMM $\lambda = (A, B)$ and an observation sequence O, determine the best hidden state sequence Q.

3. **Learning (forward-backward or Baum-Welch algorithm):** given the number of states in the HMM and an observation sequence O, learn A and B, i.e., transition probabilities and emission probabilities. Time complexity: $T \times N^2$, where T is the length of the observation sequence and N is the number of states. At first the algorithm places states randomly, after that it iteratively refines them n_iter times; this leads the overall time complexity to $n\_iter \times T \times N^2$.

4.4.2 HMM implementation

We use Baum-Welch in order to learn the HMM model, then we can use this model to compute the likelihood of any trajectory, both complete and incomplete, or we can sample the model to get a new trajectory.

Our full HMM implementation is composed of several steps that will be explained in details:

1. Import trajectories from files.
2. Remove subsequent duplicated points.
3. Interpolate a cubic spline from the remaining points and uniformly sample it to get 1.5 times the number of points used to generate it.
4. Add Gaussian noise.
5. Compute heading direction, i.e., orientation, and make it circular.
6. Duplicate x and y coordinates to increase their weight.
7. Perform HMM learning, possibly with BIC.
8. Compute final states.
9. Prevent self transitions.
10. Try to compute stable version of transition matrix.
11. Plot learned states.
12. Get samples and plot them.

The code can be found in *HMM.py*, details in Appendix D.7.

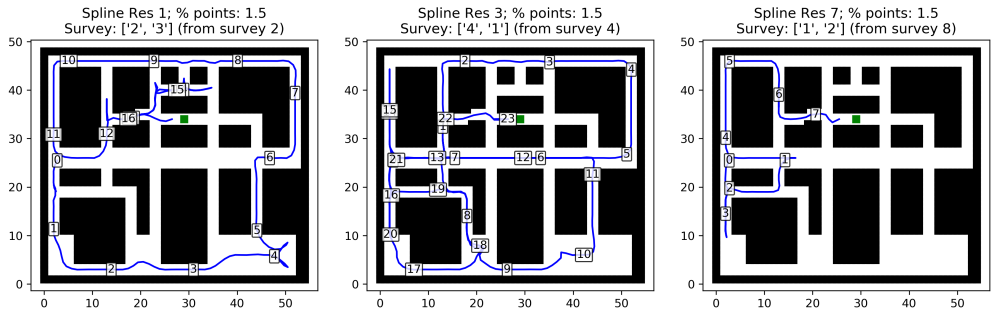

Figure 4.5: Splines of Voronoi equivalents of Results 1, 3, and 7.

Import trajectories from files

The first step is trajectory imports, our code can work with a variety of trajectories (human original, human Voronoi, robot Voronoi,...), in this case we use the Voronoi-equivalent paths. As input to the entire procedure we specify both the type of trajectories and the corresponding Result numbers that are used to univocally identify the trajectories. If we have already imported them, and we have treated them so that they are ready to be learnt, then we can use their values as input of the entire procedure and skip all steps from 1 to 6. In the next paragraphs we will use the three trajectories in Figure 4.5 as examples.

Remove subsequent duplicated points

For each trajectory we need to remove duplicated positions that are one immediately after the other, otherwise we would have a problem when adding the Gaussian noise: the noise might potentially change the geometric meaning of the points. For instance, consider an agent moving from left to right in a straight line, if a point is duplicated it means the agent moves, then stops, then moves again. The movement is always from left to right. The Gaussian noise slightly displace all points, which may result in the agent moving from left to right, then from right to left, then from left to right again. This way we introduced two reverses that are not present in the original trajectory. This will impact the computation of the heading, thus it must be avoided.

Get a spline from the remaining points and uniformly sample it to get 1.5 times the number of points used to generate it

Since Voronoi points are not uniformly distributed in the environment we convert each trajectory into a spline, then we extract a uniformly distributed

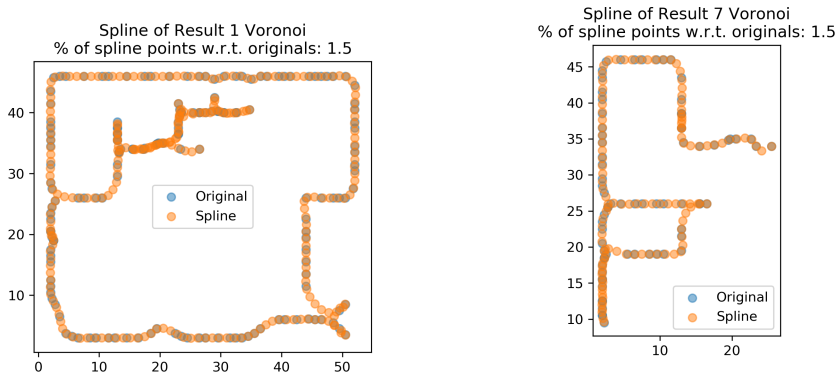

Figure 4.6: Newly computed splines of Results 1 and 7.

trajectory with 50% more points than the original one, i.e., the one used to create the spline. We can see two examples in Figure 4.6. We choose 50% since it strikes a balanced between getting a smooth trajectory and keeping the number of samples under control. In case of need, this percentage can be changed.

Add Gaussian noise

The need to add Gaussian noise may seem counter-intuitive after we converted all trajectories to Voronoi for the exact reason of reducing all possible noise. The reason we add some noise back is due to how Baum-Welch works. If we kept several points aligned in a straight line the algorithm would have no variance on the perpendicular axis to that line. Avoiding this not only allows the algorithm to work as designed, but also provides a nicer visual feedback once we plot the states of the HMM: the ellipses representing the states would still be ellipses, otherwise they would be reduced to straight lines.

Compute heading direction, i.e., orientation, and make it circular

The heading direction is computed from the new positions obtained from the spline. We cannot use the orientation sampled during the data collection process because it would not align with the new positions, and, more importantly, it would not be what we actually want. The sampled orientation is where the agent is looking, but we want the agent's direction of movement. If the agent is the robot, then the sample orientation represents where the robot is heading for, but this is not the case with human players. Humans are likely to move to a certain location while their field of view is focused on

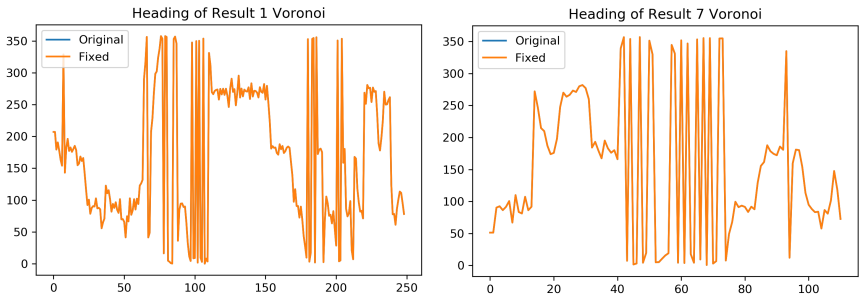

Figure 4.7: Heading of Results 1 and 7.

somewhere else.

Once the heading direction is computed we need to make it circular. The problem is the actual interval used to represent it: from 0 to 360; we can see a couple of examples in Figure 4.7. If an agent's heading direction changed from 359° to 1°, the HMM learning algorithm would see those two values as much more distant than, for example, 1° and 3°. The problem is the circularity nature of the degrees, while numbers are linear. In order to solve this we rely on the same approach used in [11] and [12]: we split the orientation value into $cos(2\pi \times orientation/360)$ and $sin(2\pi \times orientation/360)$ so that the jump $360° - 0°$, and viceversa, is completely removed.

The reasons we use the heading direction is twofold. On one side when working with robots it is good practice to take into account not only positional information but directional too [13]; on the other side if two agents walk down the same corridor in opposite directions we want them to be considered differently, without the heading direction that would not be possible since both agents would walk over the same spots.

Duplicate x and y coordinates to increase their weight

For each point of each trajectory we have x, y, cos, and sin. If we kept these four values we would get an HMM where points in different parallel corridors would be assigned to the same state state: if the agent walked down those corridors in the same direction, and the corridors are parallel, i.e., same x or same y, then three out of four values would be almost identical. This is a problem because we want states to be centred in a walkable space, not in the wall between two walkable spaces. In order to solve this problem we increase the weight of the positions, to do so in a simple way we double them so that for each trajectory point we have: x, y, cos, sin, x, and y.

Perform HMM learning, possibly with BIC

In order to learn the HMM we use the hmmlearn Python library [14], in particular we use GaussianHMM, which means hidden Markov model with Gaussian emissions. In addition to positions and orientations of the trajectories we must provide the number of components, i.e., the number of states of our HMM. This is a problem since we do not know what is the appropriate value to be set. We can solve this problem in three ways: first, we may provide a high enough number of states, in the worst case scenario we get several states where only one would be enough, but this is not ideal in terms of complexity, and it does not solve our issue. How many states are "high enough"? Thus we discard this option.

Another option is to manually test some values until we get a convincing final result, but this requires a lot of time and it is not scientifically sound.

The third solution is to rely on a criterion like Bayesian Information Criterion (BIC) [15]. We compute the BIC value via $BIC = log(n) \times k - 2 \times logLikelihood$ where log is the natural logarithm; n is the total number of points in all trajectories under consideration, i.e., the sum of their lengths; k is the number of parameters estimated by the model, i.e., the sum of the number of starting probabilities, the size of the transition matrix, 6 $\times$ the number of states used by the HMM model. We compute the BIC for all possible numbers of states starting from 1, and we stop once the BIC has worsen enough. We cannot stop as soon as it gets worse since the BIC trend presents local minima, as such we need to ensure we do not select one of them. In practice we stop once its value is above the sum of the best BIC value found so far and $min(750 \times number\,Of\,Trajectories\,Under\,Consideration, 2500)$, the idea is that we use 750 or 1500 or 2250 or 2500 if we are considering one or two or three or more than three trajectories respectively. In Figure 4.8 we can see two examples: the first with only one trajectory; the second with three. For extra information we also include the value for the logLikelihood. At the end of the entire procedure we visually inspect the HMM to ensure the result is in line with our expectations, since it is, we consider BIC a suitable approach to solve our problem.

Compute final states

Once the HMM model is learnt we compute which of its states are final, i.e., which states are the ones the agent needs to reach in order find the target(s). Since the states are different for each model we cannot establish them once and for all, but we need a way to dynamically compute them. For each trajectory, one of the collected piece of information is *has found all*

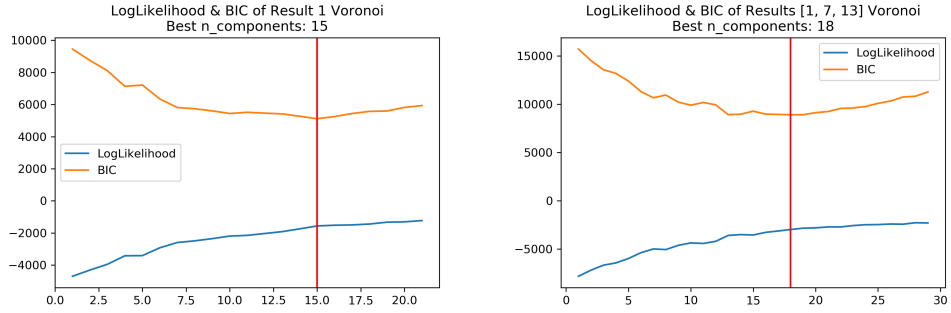

Figure 4.8: LogLikelihood of Result 1 and the combination of Results 1, 7, 13.

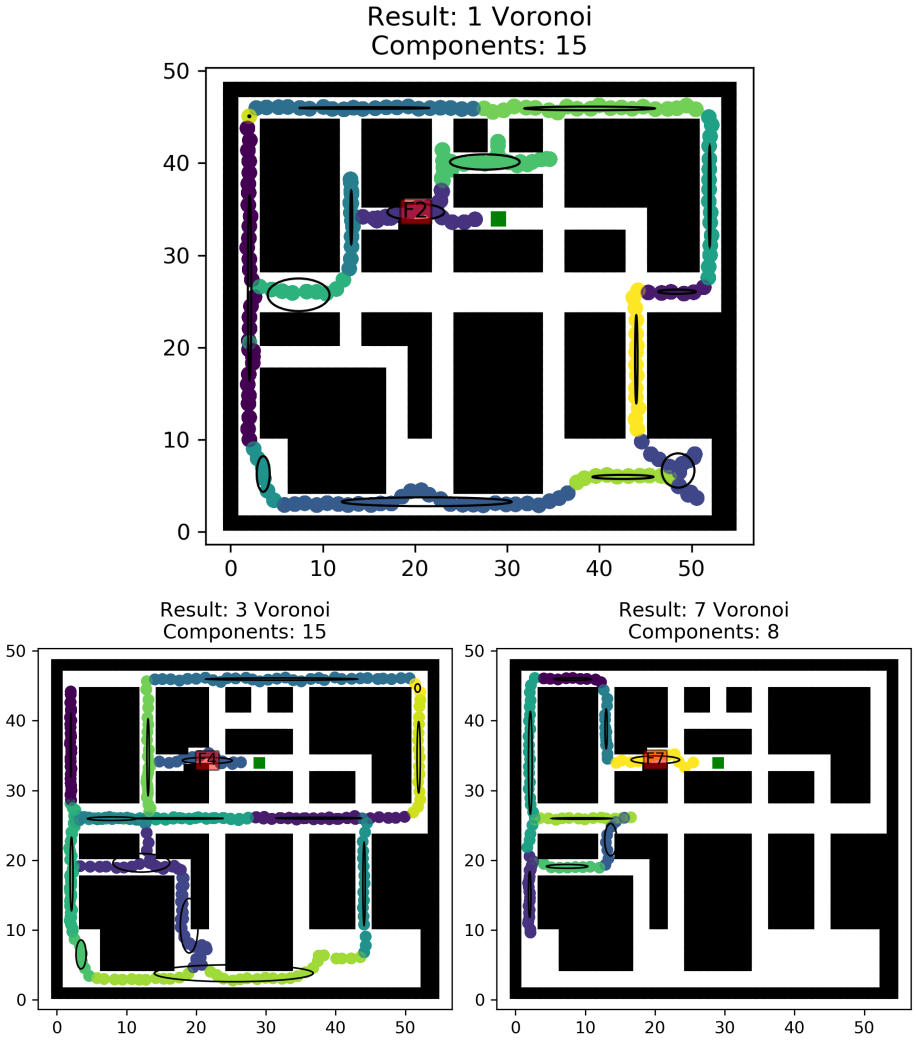

Figure 4.9: HMMs of Results 1, 3, and 7.

targets, which tell us whether the agent has found all targets or not; it can be `True` or `False`. If it is `True` then we check which is the HMM state that most likely generates the ending position of that trajectory and we mark that state as final, we can do this by applying Viterbi (Section 4.4.1). The more the trajectories, the more accurate the marking operation. Since we mark based only on the last trajectory position we need at least as many trajectories as targets, and these trajectories must be such that they all end in different targets and all their *has found all targets* are `True`. The computation of the final states is mandatory only if we want to sample the HMM. Examples of final states con be seen in Figure 4.9, where each final state is labeled with a small red box containing the letter "F" followed by a number.

Prevent self transitions

The transition matrix includes self-transitions, since each state contains several points. This means that the probability of remaining in each state is much higher than the probability of leaving it. This is useful if we want to model the speed at which the player can cross the states, but we are not interested in that. Since we want to be able to sample the HMM, we zero all transition probabilities on the main diagonal, and we distribute each value across the other probabilities on the same row. The distribution is not uniform, but it is weighted on the values of each element in the row; however, if a row i contains only zeroes, with the exception of the element in $[i, i]$, that row is not changed. For example, if the first row of the transition matrix were $[0.8, 0.1, 0.0, 0.1]$, then it would become $[0.0, 0.5, 0.0, 0.5]$; if the second row of the transition matrix were $[0.0, 0.7, 0.1, 0.2]$, then it would become $[0.0, 0.0, 0.333, 0.666]$, and so on.

Try to compute stable version of transition matrix

Given the limitations of the aforementioned approach to final states identification, we look for a possible solution by powering the transition matrix several times until it no longer changes, at this point it reaches the stable form. We check the stable matrix for any absorbing state, i.e., a state that can be reached but not left. Any absorbing state is marked as final. Alternatively, we can avoid powering the matrix by solving a linear system of equations. We try this approach, but we cannot find any absorbing state, thus we keep using the previous approach.

Plot learned states

We plot the HMM states, in Figure 4.9 we can see the states as black ellipses with additional labels over the final ones. Note that in this case we computed one HMM for each trajectory, but we can learn an HMM for multiple trajectories. In Figure 4.9 we can also see that, despite being learned from different trajectories, most of the states can be overlapped. This confirms BIC as a suitable solution to determine the number of states in an HMM.

Get samples and plot them

In order sample the HMM we could use its default sampling function. This works and avoid self-transitions since we have updated the transition matrix, however we want more control over the sampling process, for this reason we implement a custom sampling procedure. This is beyond the purpose of this chapter, we will explain this procedure in Section 5.10.2.

4.4.3 HMM-based clustering

Once an HMM model is learnt, we can save it and use it to compute the logLikelihood of any trajectory. The idea is to apply the forward algorithm (Section 4.4.1) to get the logLikelihood and then use it as distance in an agglomerative hierarchical clustering procedure. We start by considering each trajectory as its own cluster, and we stop once all trajectories are in the same cluster.

Firstly we compute one HMM for each cluster, then we compute the logLikelihood of each cluster w.r.t each HMM. We insert those results in a matrix where the element in cell $[i, j]$ is the logLikelihood of cluster i w.r.t. the HMM learnt from cluster j. The resulting matrix is not a distance matrix yet, it is not symmetric. Since each HMM is computed with BIC they all have different numbers of states, this makes their logLikelihood values incomparable, furthermore, we must take into account the lengths of the trajectories since the longer the trajectory the lower the logLikelihood. To cope with these issues we apply the same technique used in [10]: we multiply each value in our matrix by γ, given a cell $[i, j]$ we can compute its γ via $\gamma = \frac{average\ lengths\ of\ traj.\ in\ cluster\ i}{average\ lengths\ of\ traj.\ in\ cluster\ j}$. Once all cells have been multiplied by their γ, we average them in the following way: $cell[i, j] = \frac{cell[i,j]+cell[j,i]}{2}$. Now the matrix is symmetric and we can use it as distance matrix. We select the highest value in the upper-right triangular matrix (main diagonal excluded), then all trajectories in the cluster corresponding to the selected row must be added to cluster corresponding to the selected column. At this point we have

one less cluster and we can restart the process.

What we have described is the theoretical approach, the practice has proved more challenging. We have to take into account the fact that, for each new HMM that must be learnt, a computationally expensive procedure is performed; not only that, for one actual HMM several others have to be learnt and discarded due to how BIC works. Furthermore, hmmlearn has shown signs of instability: sometimes it throws an error, other times it works as expected. During our experiments we gave up when an HMM with twelve trajectories threw an error after more than an hour of computation. We highlight the fact that by using BIC we were effectively multiplying the aforementioned Baum-Welch complexity (Section 4.4.1) by the number of BIC iterations. This, combined with the instability of hmmlearn, makes us decide to **abandon this approach, at least as far as clustering is concerned**. We will consider HMMs again in Section 5.10.2.

4.5 Survey-based clustering

We need a new approach, we decide to try with the surveys. The idea is to cluster trajectories based on what players answered to the survey questions (Section 2.4.1), this means that from now on we will consider only those trajectories whose survey answers are available. In order to understand if this might yield a good result we need to check if the various groups created based on the answers are actually distinct from each other. To do so, we look for measures that can show major differences among the various groups. We take into consideration some measures that are directly obtained from the collected data, but we also include measures that are computed on the fly.

4.5.1 Critical Voronoi points and groups

Before listing and explaining all measures we need to explain the concept of critical Voronoi points and critical Voronoi groups. In Section 3.2.3 we described how we get the Voronoi points for each map. If we look at the leftmost maps in Figure 4.10 we can see that some of those points are less relevant: we want to know if and where a player enters or leaves a certain space (a corridor, a room,...), however, we are less interested in what they do inside that space; for this reason we deploy an algorithm that filters the Voronoi points. This algorithm keeps only those Voronoi points that have one (and only one) closest Voronoi point, e.g., points inside a corridor are discarded since they are between two equidistant points, while the points on

the extremes of the corridor are kept. In this way we obtain the **critical Voronoi points** shown in the central column of Figure 4.10; then we group together all critical points that are close to each other, by close we mean that their distance is below a given threshold, 2 Unity units in our case. This grouping procedure provides the **critical Voronoi groups** shown in the rightmost maps in Figure 4.10.

In the following sections critical Voronoi groups are used as proxy to measure how many map areas a player has explored. Being automatically generated, they are not perfectly aligned with what we may consider an appropriate map area, but on the other hand this approach prevents any human bias from affecting the areas identification process.

4.5.2 Measures

After defining what critical Voronoi groups are we are ready to list all measures that will be used in the following sections. The full list is the following:

- **Average distance between repeated positions**
  Average number of samples between each repeated position pair. Not all repetitions are considered, we only take them into account according to the trajectory order. For example, in a trajectory constituted by the following positions $[..., A_1, ..., A_2, ..., A_3, ...]$ we consider 2 repetitions: $A_1 - A_2$ and $A_2 - A_3$, while $A_1 - A_3$ is not considered. Note that any immediately repeated position is discarded before starting this procedure. If a trajectory has no repeated position, then its value is set to `null`.

- **Completion time**
  Time elapsed before the last remaining target is found, otherwise it is the time limit before the exploration is automatically terminated.

- **Distance**
  Length or number of positions in the Voronoi equivalent path.

- **Average speed**
  It is computed via $\frac{distance}{completion\,time}$.

- **Number of repeated positions within repetition window n**
  Each time a repeated position is found within n positions of distance a counter is increased. This counter is the number of repeated positions. The window size considers n positions after the base position. For example, in the following list of positions $[..., A, B, C, A, ...]$ if we

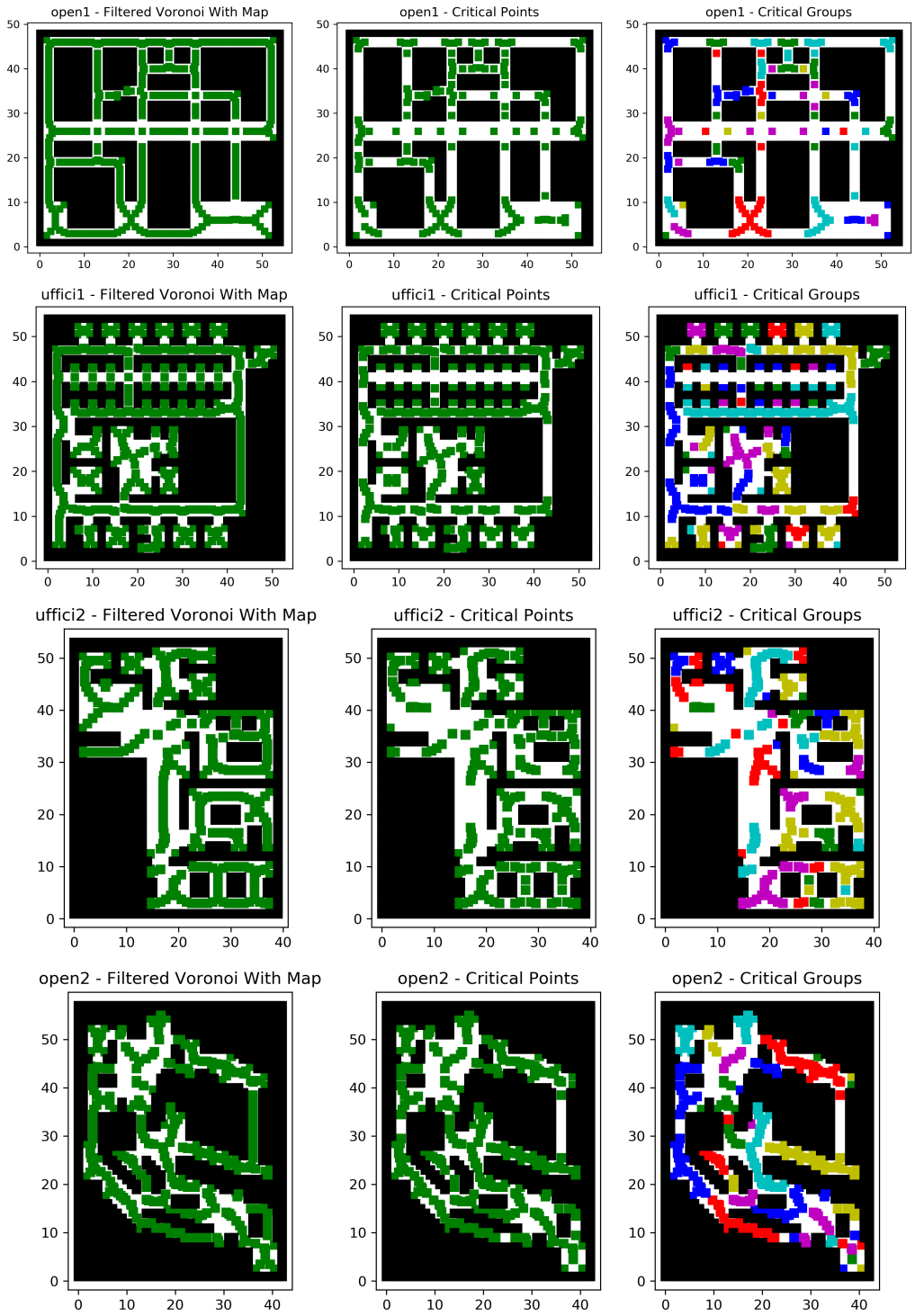

Figure 4.10: Critical Voronoi points and groups.

considered A as the base position then we would not found any duplicate for n equal to 1 or 2, but we would found a duplicate for n equal to 3. Note that any immediately repeated position is discarded before starting this procedure.

- **Percentage of critical Voronoi groups covered by each trajectory**
  It is computed via $\frac{number\ of\ unique\ critical\ Voronoi\ groups\ visited\ by\ trajectory\ i}{total\ number\ of\ critical\ Voronoi\ groups\ in\ that\ map}$.
- **Percentage of optimal exploration**
  It is computed via $\frac{number\ of\ unique\ Voronoi\ points\ in\ trajectory\ i}{total\ number\ of\ Voronoi\ points\ in\ trajectory\ i}$.
  Note that the denominator is the length of the trajectory.
- **Percentage of Voronoi points covered by each trajectory**
  It is computed via $\frac{number\ of\ unique\ Voronoi\ points\ in\ trajectory\ i}{total\ number\ of\ Voronoi\ points\ in\ the\ map\ where\ i\ takes\ place}$.

4.5.3 Survey-based clustering results

For each measure we plot two types of graphs: the first one is the box plot [16], the second one is the normal distribution [17]. A **box plot** is a rectangle, i.e., a box, that starts from the lower quartile values of the data and extends up to the upper quartile. A horizontal line is placed to identify the median. Two lines terminating with whiskers extend from the box, these lines cover all the data points that are not outliers. All outliers, i.e., all points below or above the whiskers, are represented with a black circle around them. **Normal distribution** plots contain some vertical lines, each one representing the mean of the data points of each answer; these points are fitted using a normal distribution and then represented in the curves. In both types of graphs each data point represents a trajectory and it is drawn with a transparency value applied to it, this means that if several data points are overlapped their colours become darker. Whenever possible grey lines are added to the image in order to show the minimum and/or maximum value(s) for that particular measure.

We look for the graphs where the distribution of the data points is different among the various answers. For this reason we plot the graphs for all four questions (Section 2.4.1), in both single map, i.e., all data points come from that one map, and all maps variants, i.e., data points come from all maps. In this document we report only a subset of images due to space constraints. In Figures from 4.11 to 4.23 we report the graphs of all measures for the answers to the first survey question; then we report some of the graphs for the answers to the second question. All these graphs contain data from all

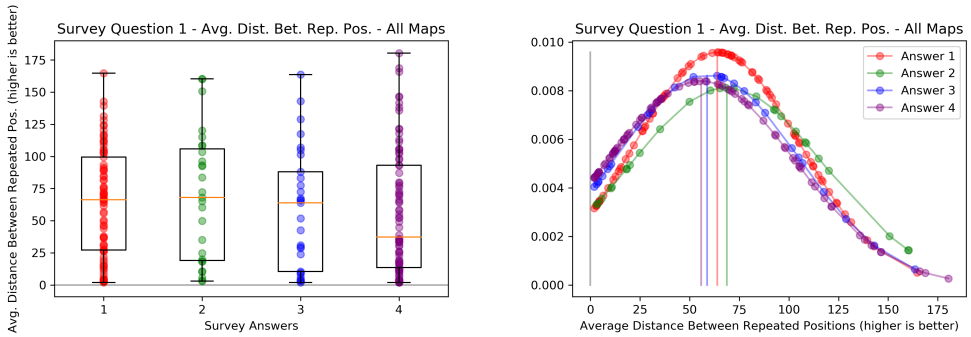

Figure 4.11: Question 1 - Average distance between repeated positions.

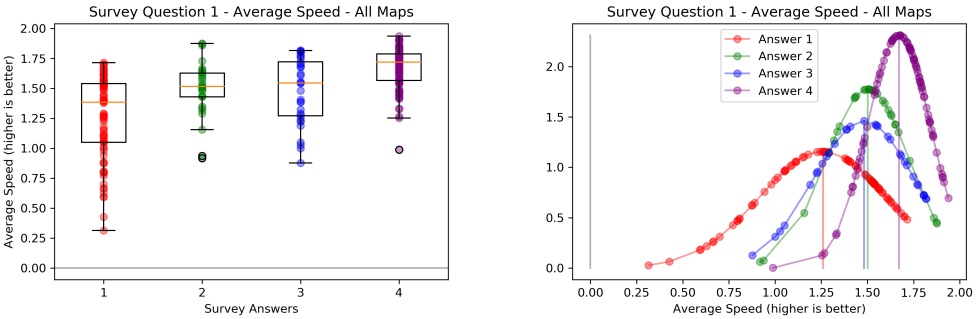

Figure 4.12: Question 1 - Average speed.

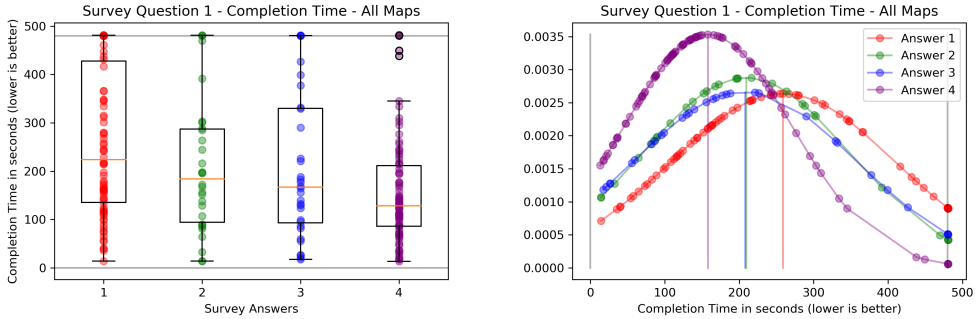

Figure 4.13: Question 1 - Completion time.

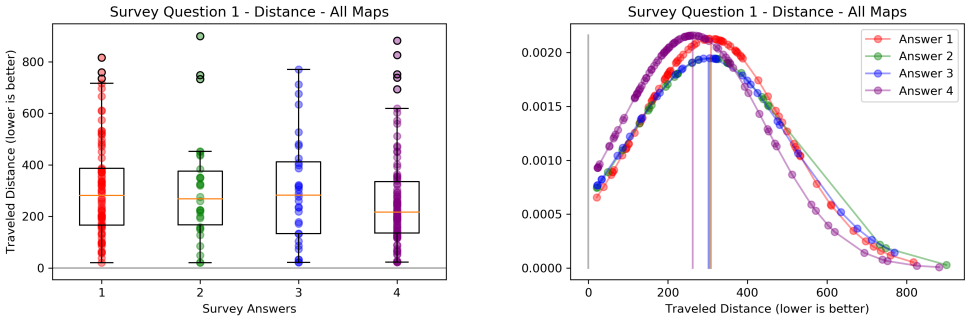

Figure 4.14: Question 1 - Distance.

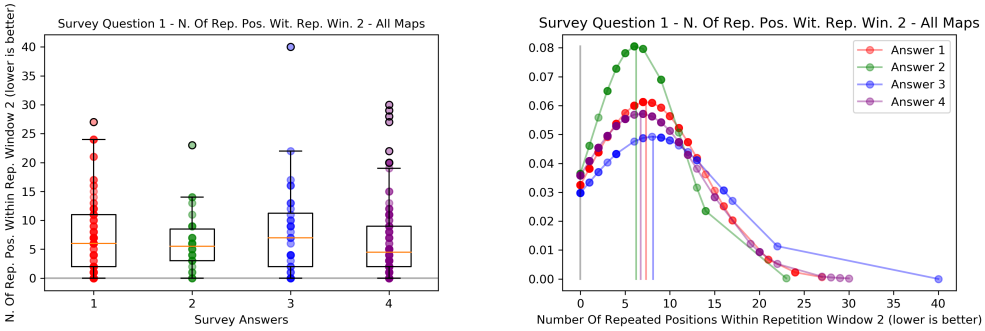

Figure 4.15: Question 1 - Number of repeated positions within repetition window 2.

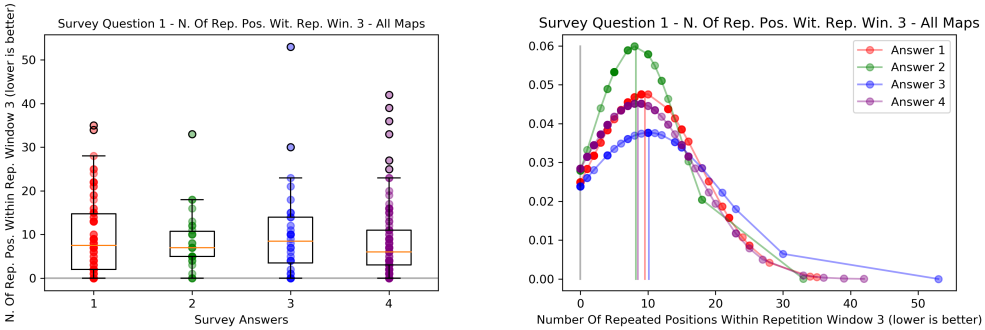

Figure 4.16: Question 1 - Number of repeated positions within repetition window 3.

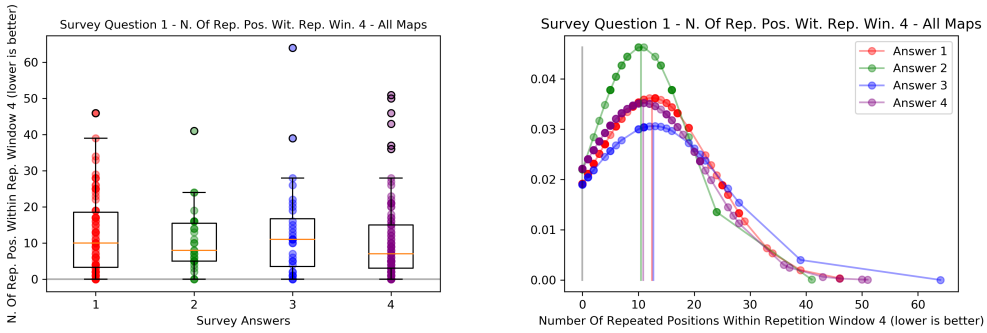

Figure 4.17: Question 1 - Number of repeated positions within repetition window 4.

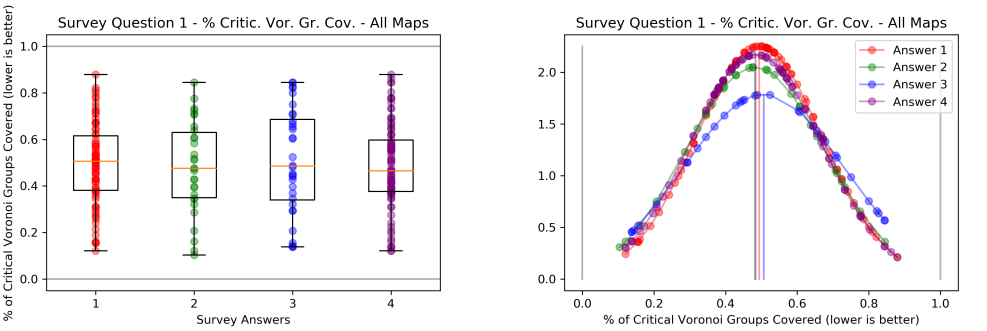

Figure 4.18: Question 1 - Percentage of critical Voronoi groups covered by each trajectory.

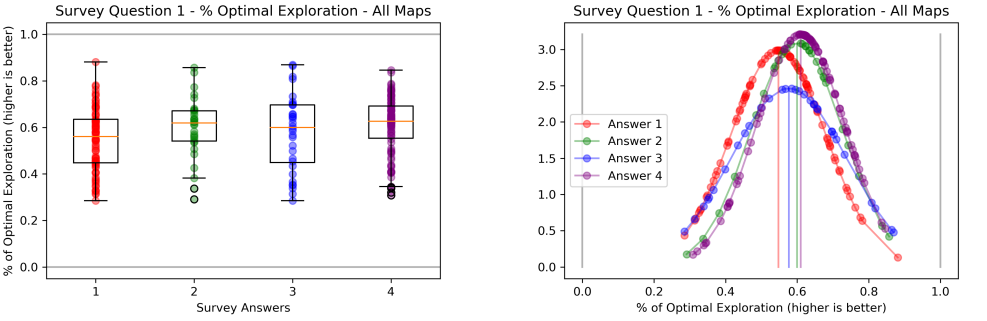

Figure 4.19: Question 1 - Percentage of optimal exploration.

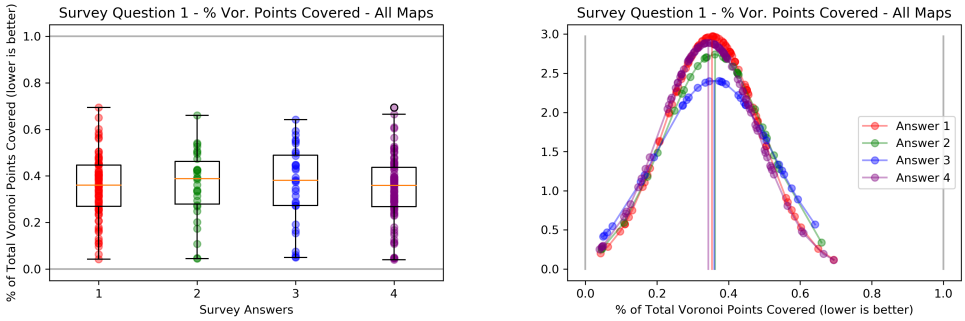

Figure 4.20: Question 1 - Percentage of Voronoi points covered by each trajectory.

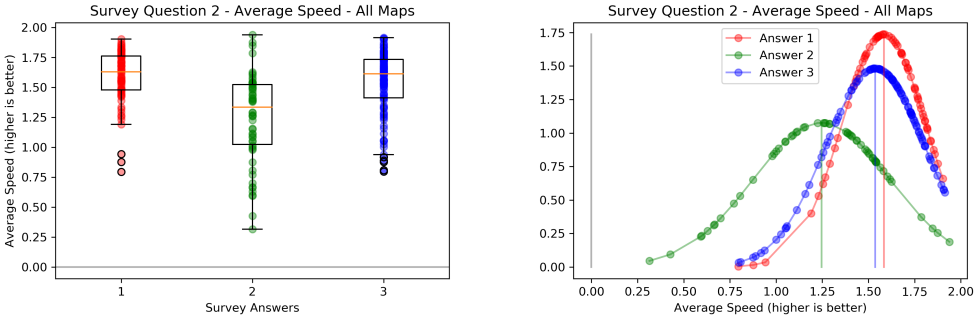

Figure 4.21: Question 2 - Average speed.

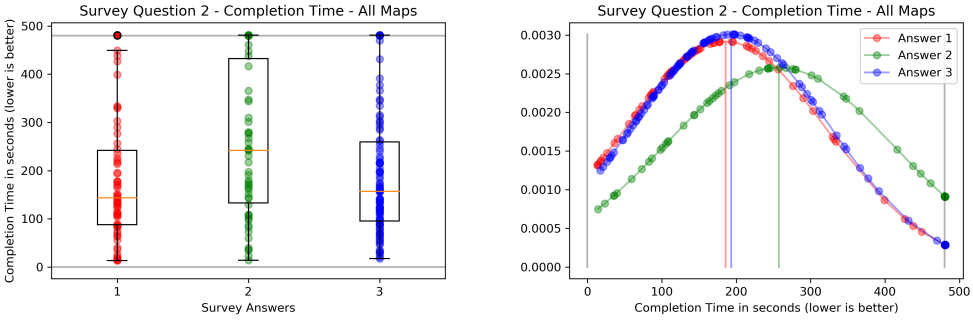

Figure 4.22: Question 2 - Completion time.

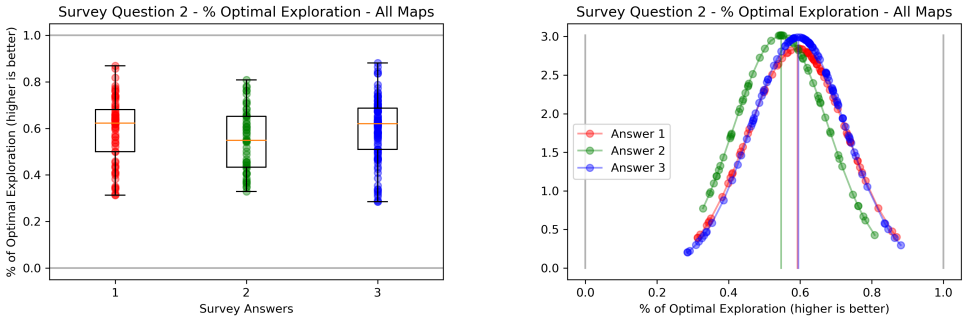

Figure 4.23: Question 2 - Percentage of optimal exploration.

maps. We highlight the fact that answers 1 and 3 of question 2 are technically different, but they identify the same class of users: those with some experience with FPS games.

If we check all images for all questions and all variants we find some promising measures: *average speed* (questions 1 and 2), *completion time* (questions 1 and 2), *distance* (question 1) and *percentage of optimal exploration* (question 2). Since we do not know if players answered truthfully, we decide to discard all outliers and plot all graphs again. The following list contains the outliers for the previously listed measures:

- Q1 - Average speed: 153, 235, 236.
- Q1 - Completion time: 167, 178, 199, 203, 221, 234.
- Q1 - Distance: 41, 44, 72, 141, 167, 178, 199, 209, 221, 234, 246.
- Q2 - Average speed: 74, 114, 116, 117, 125, 192, 193, 236.
- Q2 - Completion time: 44, 72, 193, 203, 221, 234.
- Q2 - Percentage of optimal exploration: $\emptyset$.

We highlighted in blue the outliers in common between different measures of question 1, in red the ones in common between different measures of question 2 and we underlined the ones in common between question 1 and 2. We can see that some of them are shared, this does not come as a surprise since *average speed* is computed from *distance* and *completion time*, while the higher the *distance*, the higher the amount of time required to cover that distance, thus the higher the *completion time*. As stated before, we remove these outliers and then we plot the graphs again. In Figure 4.24 we report an example.

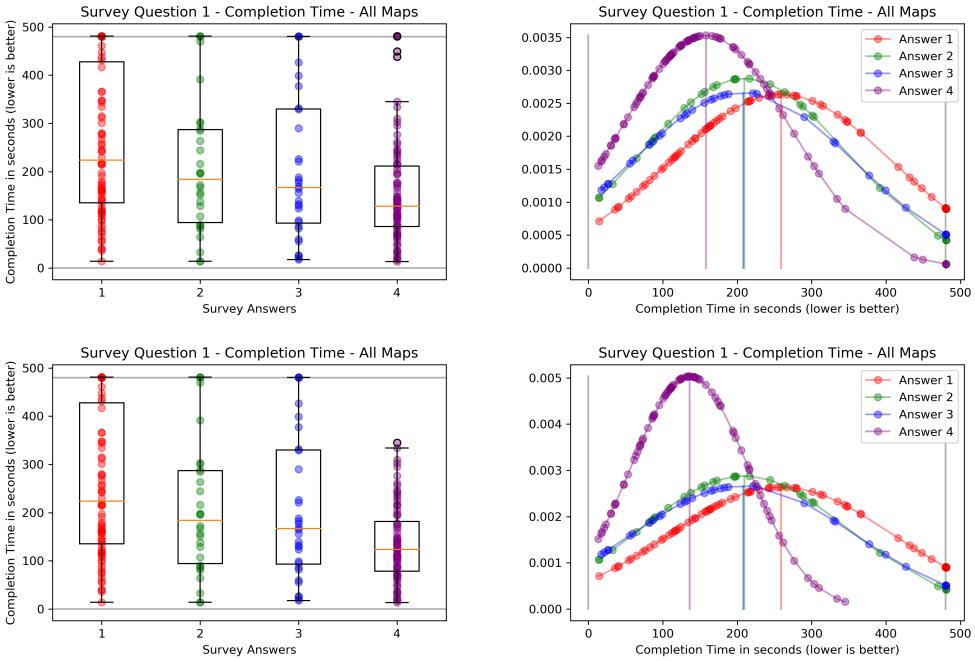

Figure 4.24: Question 1 - Completion time with and without outliers.

After removing all outliers, if any, we collect the trajectories corresponding to the remaining data points and we group them based on the answers. Ideally, we would like to see one group with all the players who got lost, another group with all the players who explored optimally,... However, that is not the case. **In all cases we get three or four groups containing mixed trajectories.** For example, even after removing all outliers of *question 1 - average speed* we can find human Results 32 and 67 in the same group. In Figure 4.25 the two trajectories are represented as splines of the Voronoi equivalent paths with superimposed labels to convey the order in which players explored the map. We can see that the two paths are extremely different, one always explores new areas, the other one gets lost and passes through the same locations multiple times.

Similar conditions are found in other groups, which means that **neither the answers to the survey can be used to appropriately cluster the trajectories**.

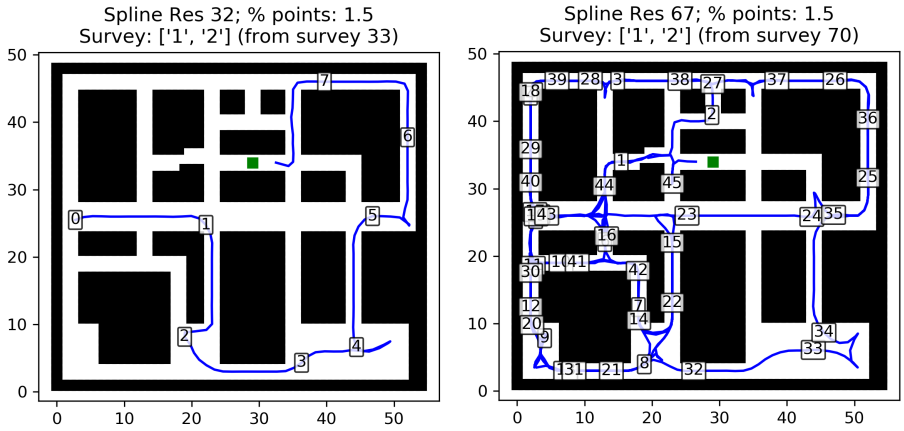

Figure 4.25: Splines of human Results 32 and 67.

4.6 Measure-based clustering

4.6.1 Feature normalisation

Despite its failure, the survey-based approach left us with several measures (Section 4.5.2) that can be used as features for an agglomerative hierarchical clustering procedure. We keep working only on the subset of trajectories whose survey answers are available; this limits the amount of usable trajectories, but on the other hand it allows us to rely on answers to the survey to better characterise clusters. Furthermore, we focus on maps open1 and uffici1 since they are the ones for which we have the highest number of samples. Before using those measures we must solve some problems.

The first one is that each feature has its own range of possible values. In order to solve this, we **normalise the values before the clustering phase**, in this way all values are between 0 and 1.

Most of the features can be normalised without issues, but *average distance between repeated positions* cannot: trajectories where no position is repeated has no value for this feature. From a conceptual point of view the higher the *average distance between repeated positions* the better the trajectory, so we replace all `null` values with *1.25×highest average distance between repeated positions*. This creates a gap among the perfect trajectories with no repeated position and the others. After all `null` values have been replaced we can apply the normalisation.

Another problem is the features *has found all targets*. It was not plotted in the previous section since it can only take two values: `False` or `True`. We decide to convert the Boolean values as 0 and 1, respectively.

Lastly, we choose one value for n in *number of repeated positions within repetition window n*. We pick $n = 2$, in this way we can measure how many times an agent has immediately backtracked.

4.6.2 Feature sets

Now we can normalise all values; once this process is completed we need to select which features should be used by the hierarchical clustering algorithm. We identify the following possible sets:

- **Full**: it uses all 9 features: *average distance between repeated positions* normalised, *average speed* normalised, *completion time* normalised, *distance* normalised, *has found all targets*, *number of repeated positions within repetition window 2* normalised, *percentage of critical Voronoi groups covered*, *percentage of optimal exploration*, and *percentage of Voronoi points covered*.

- **Partial**: it uses a subset of features, the idea is to use those features that might be more representative of the agent ability. For example, if we consider the *completion time* we already know, thanks to L&Z's work, that it cannot be used to distinguish skilled players from less skilled ones. The *average speed*, on the other hand, combines *distance* and *completion time* in a way that is likely to provide a more clear distinction between non-players and FPS players that are more familiar with the control scheme, and as such are able to move faster. This is suggested by the graph in Figure 4.21 where those who answered 2 are people who never played an FPS game.
  The list of features is: *average distance between repeated positions* normalised, *average speed* normalised, *has found all targets*, *number of repeated positions within repetition window 2* normalised, and *percentage of optimal exploration*.

- **Custom_A**: this is a testing subset where we try to take the idea of the Partial set to the extreme by keeping only: *average distance between repeated positions* normalised, *number of repeated positions within repetition window 2* normalised, and *percentage of optimal exploration*.

4.6.3 Principal component analysis

In order to figure out which set of features is the one providing more structure to our samples we apply Principal Component Analysis (PCA) [6] and

t-distributed Stochastic Neighbour Embedding (t-SNE) [18]. The purpose of both techniques is to represent our data points in two and three dimensional spaces. **PCA** analyses the n-dimensional feature space in order to find which are the most important directions, i.e., the ones over which most of the variance is spread, then it discards the others, inevitably losing information but gaining the possibility of visualising the data points in the new dimensionally-reduced space. **t-SNE** operates with the same goal by relying on probability distributions in order to group similar data points close to one another. We use the Scikit learn implementations for both PCA [19] and t-SNE [20]. It is important to highlight that while PCA does provide identical results if it is run multiple times with the same data points, t-SNE does not. In Figures 4.26 and 4.27 we can see that the **Custom_A set is either similar or worse than the others**, thus we discard it. We report only 2 dimensional graphs since 3 dimensional ones provide similar results.

The other two sets are both potentially valid, so we plot the same PCA graphs with the addition of all 484 data points obtained from the grid search (Section 2.4.4). The PCA has been performed using only human Results, only later a transform function is applied to robot Results in order to reduce them to the lower dimensional spaces. For this reason the positions of the humans on the graphs are unchanged. By looking at Figures 4.28 and 4.29 we can see that **the set** that **better** maps the robot Results over the human ones **is the set Full**, thus we keep it and **discard the Partial one**. The *r1* after the map name in the image title will be explained in Section 5.3, it is not relevant at the moment.

4.6.4 Measure-based clustering comparisons

Now we can apply hierarchical clustering to our Full set of features. In Section 4.2 we presented seven methods. We use all of them. In Figures 4.30 and 4.31 we can see the average method results. After the clustering is done, we perform the Knee/Elbow analysis (KE) (Section 4.2) but this time we use the actual centroid. In order to choose only one method we observe both the KE graph and the dendrogram; ideally we would like few clusters with low WSS and high BSS (Section 4.2).

We discard single link (min) since WSS and BSS are not low and high, respectively, until we reach a high number of clusters. Furthermore, the cluster distribution on the dendrogram is more unbalanced w.r.t. the dendrograms of other methods; many nodes of the dendrogram separate one trajectory from all the others instead of separating one group from another group.

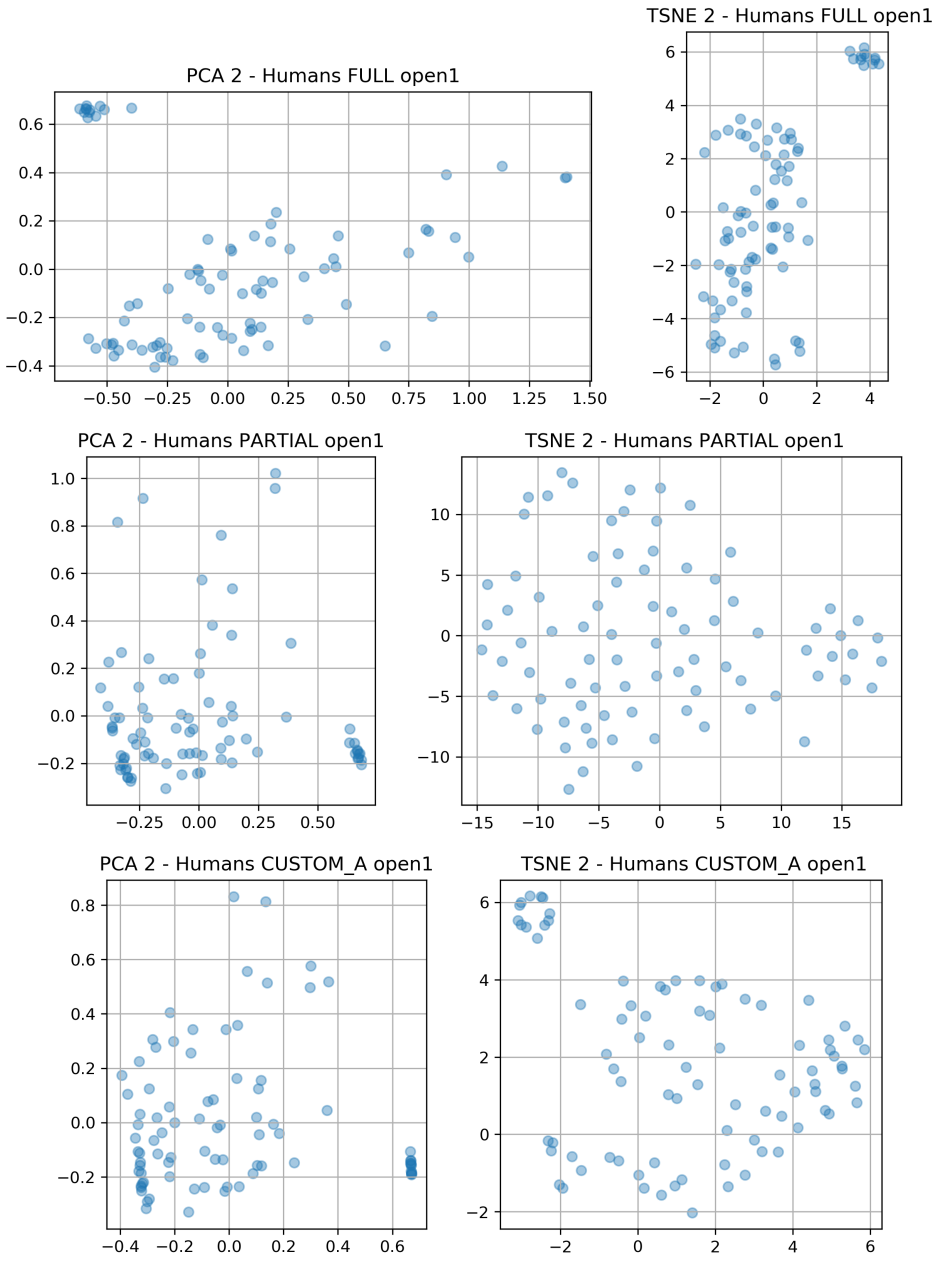

Figure 4.26: PCA and t-SNE - 2 dimensions - map open1.

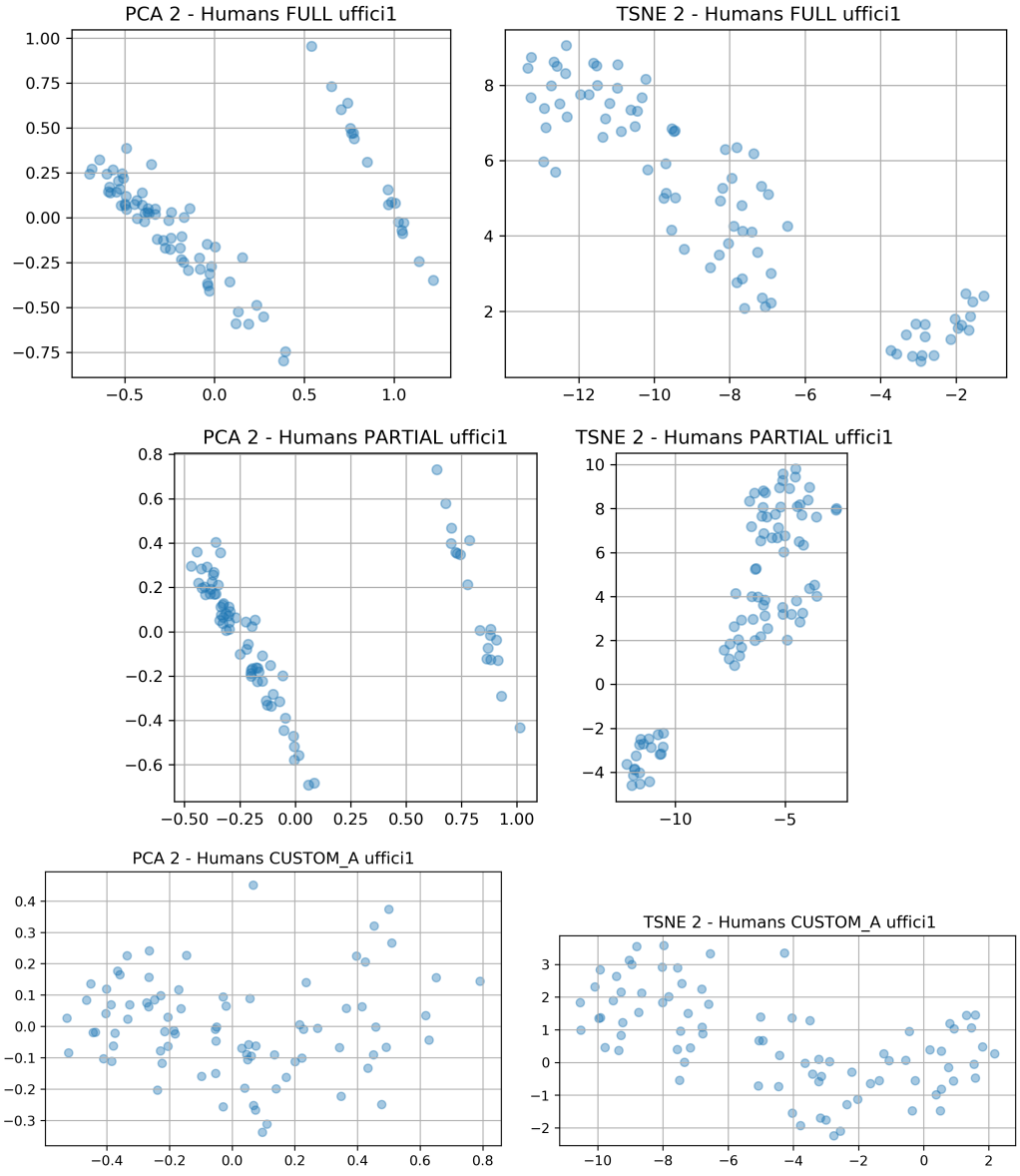

Figure 4.27: PCA and t-SNE - 2 dimensions - map uffici1.

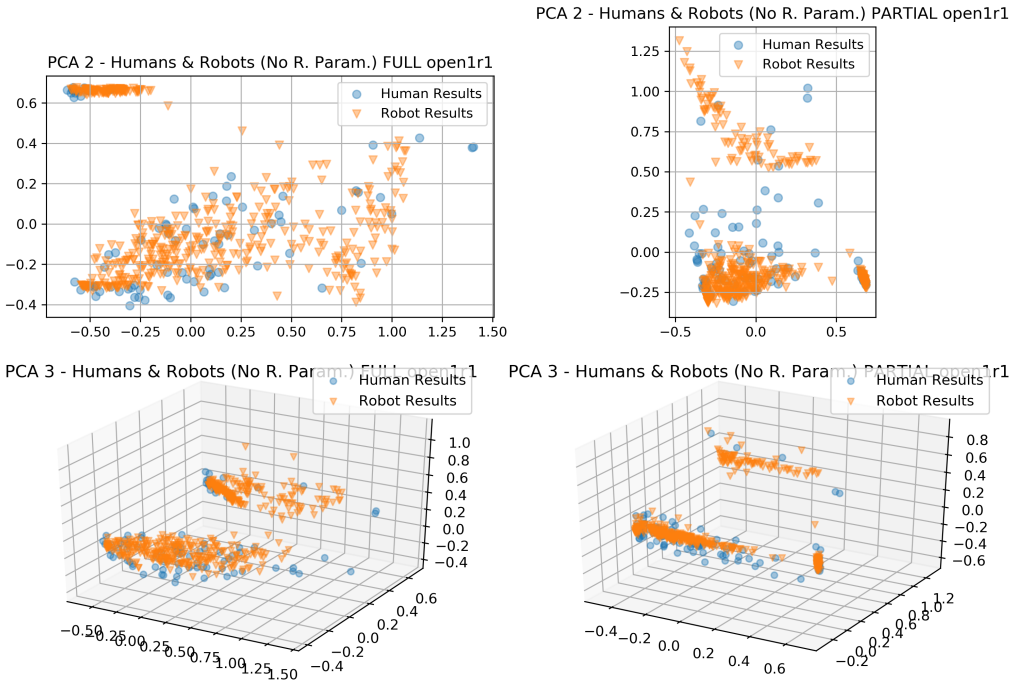

Figure 4.28: PCA - humans and grid search robots - map open1.

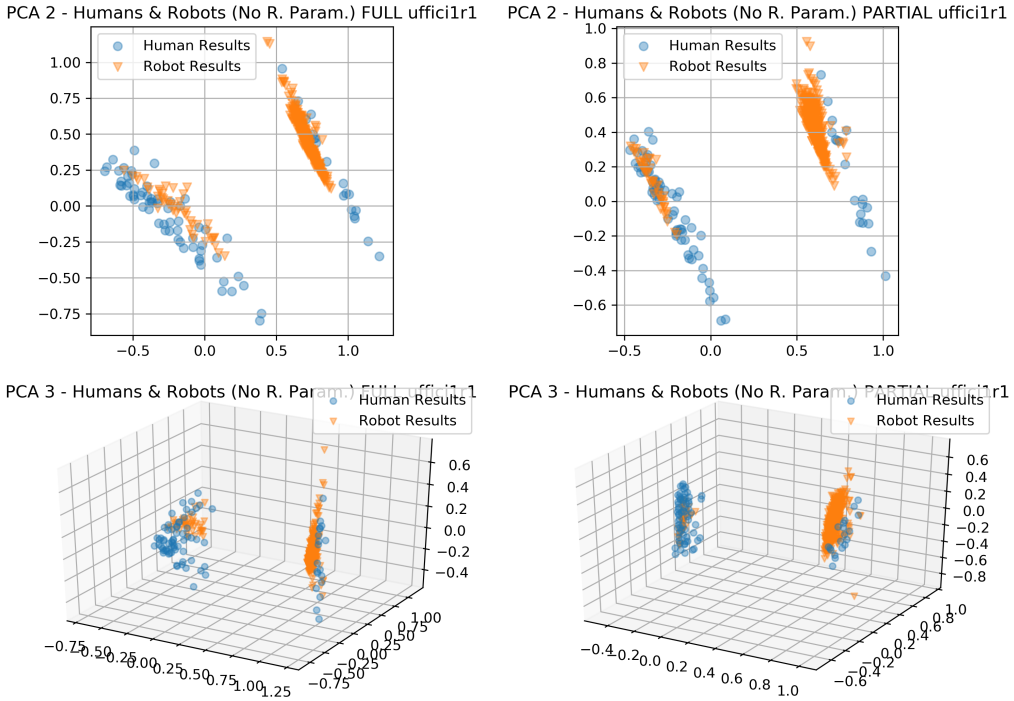

Figure 4.29: PCA - humans and grid search robots - map uffici1.

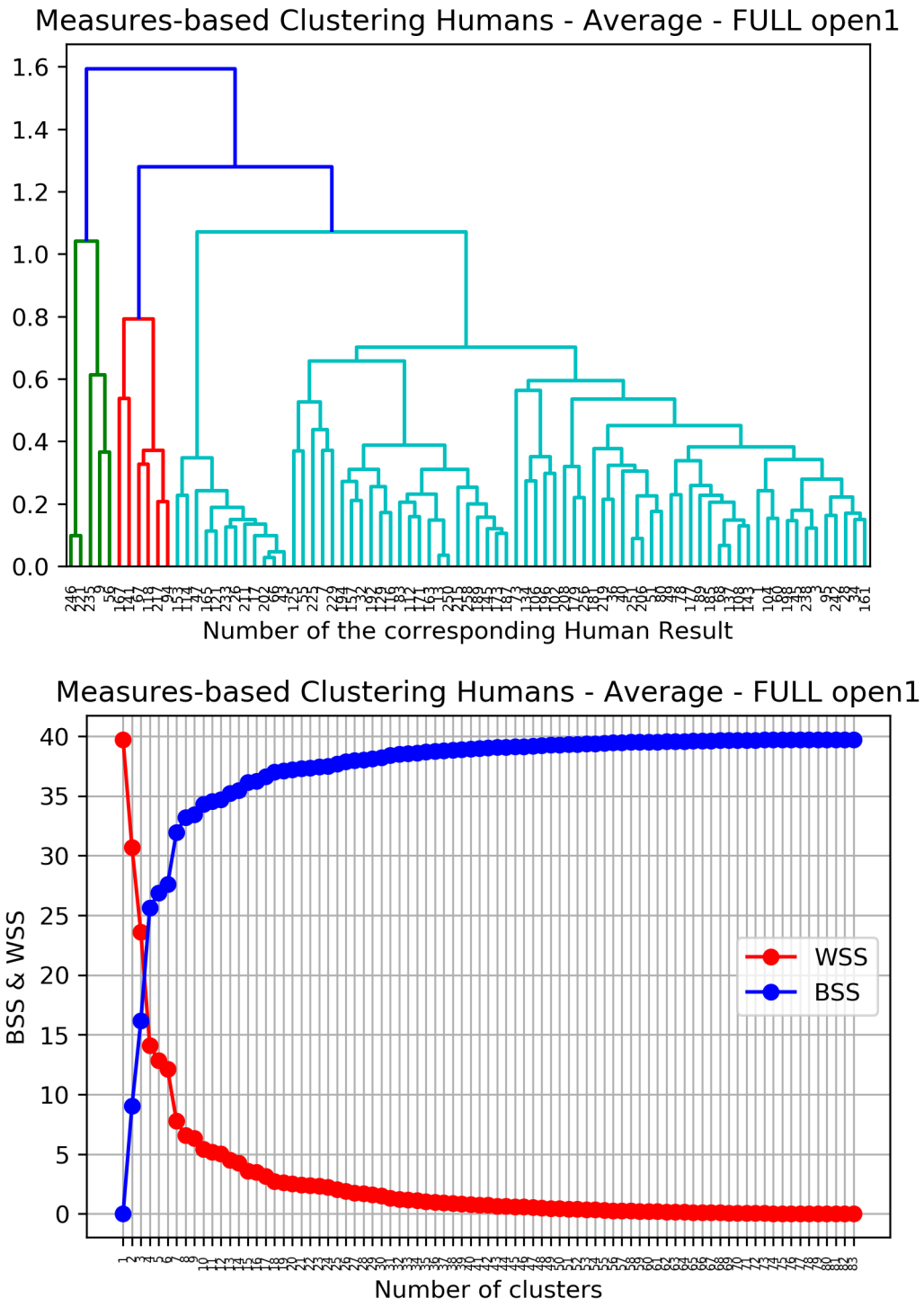

Figure 4.30: Measures-based clustering human Results - Average - map open1.

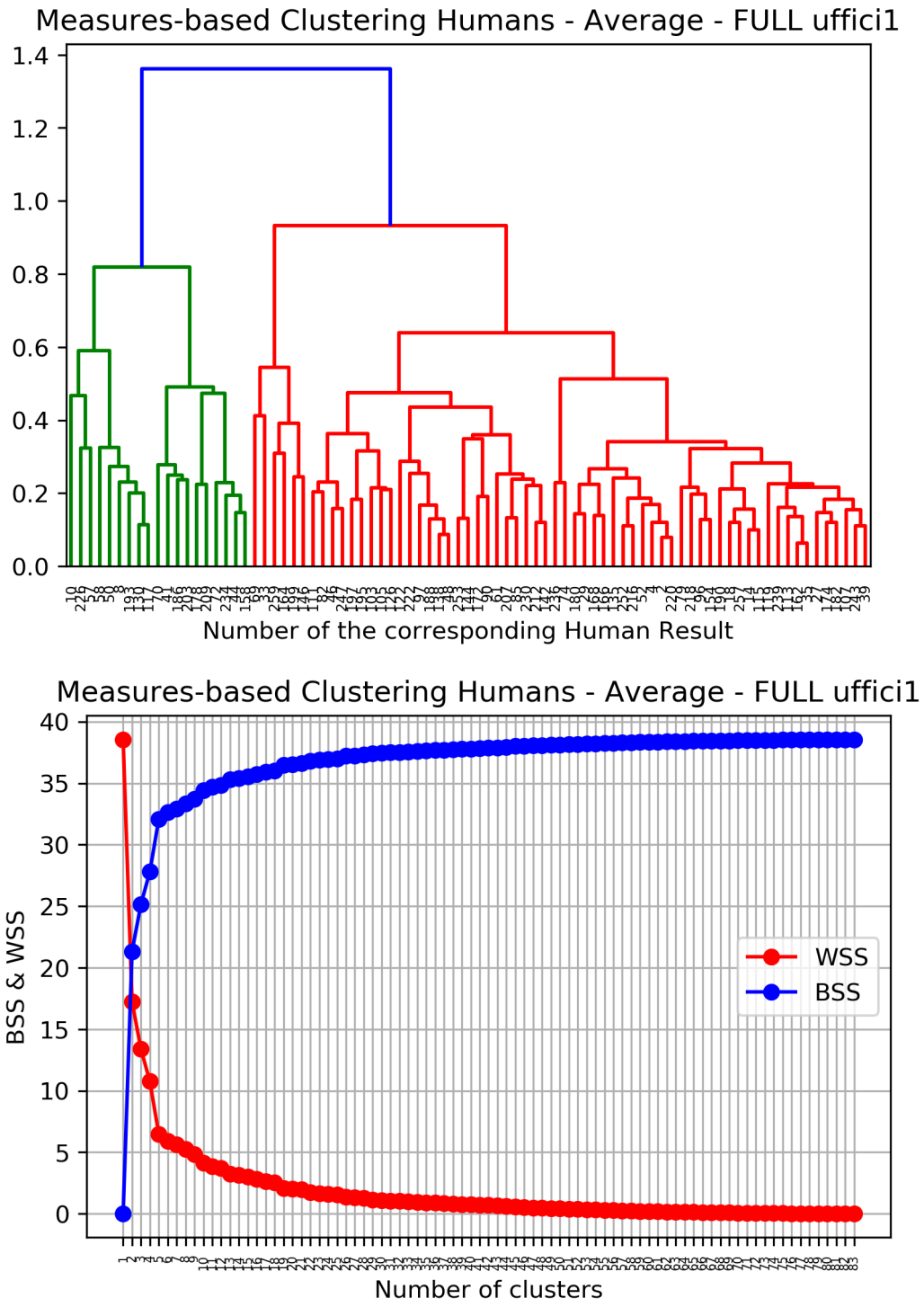

Figure 4.31: Measures-based clustering human Results - Average - map uffici1.

Map	AVG	CENTR	MAX	MED	MIN	WARD	WEIG
open1	8 (7)	10 (9)	9	11 (10)	5, 12, 40	5	8 (7)
uffici1	5	5	5	6 (4)	8, 21, 30	5	6 (4)

Table 4.6: Number of clusters per method.

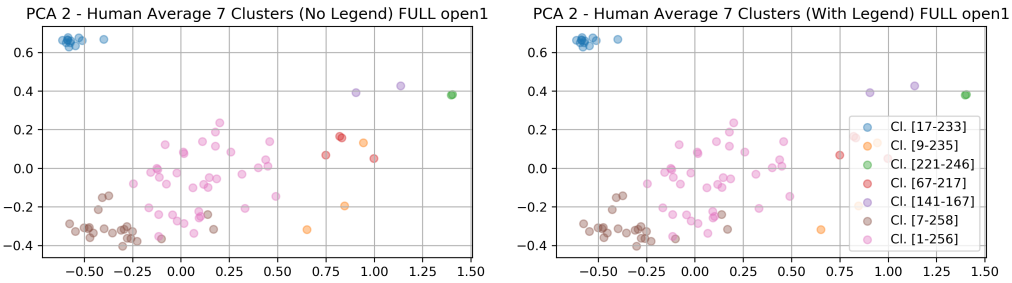

Figure 4.32: PCA - human clusters - map open1.

We discard ward and weighted since their KE graphs have no knees or elbows.

Average, centroid, complete link (max) and median are all promising methods, and they all suggest similar numbers of clusters, with only minor differences as shown in Table 4.6. In Table 4.6 some values are between parenthesis, the reason is that they are not the exact values where the knees/elbows are located, but looking at their WSS and BSS we see that they are very close to the WSS and BSS of the exact values; this makes the values between parenthesis particularly interesting, since it allows a slightly smaller number of clusters while keeping WSS low and BSS high.

4.6.5 Measure-based clustering results

We choose the average method since it is the one with the lower number of clusters. We can see how each cluster is distributed in the reduced space in Figures 4.32 and 4.33. For open1 we opted for 7 clusters instead of 8, while for uffici1 we kept 5. Now that we have the trajectories in each cluster we can check the results. Differently from all previous attempts now we get trajectories that are clustered as we expected. **In the same cluster we find** both geometrically **similar trajectories** and different ones, however, we cannot find together two different trajectories were one explores each location multiple times while the other one never crosses the same spots twice.

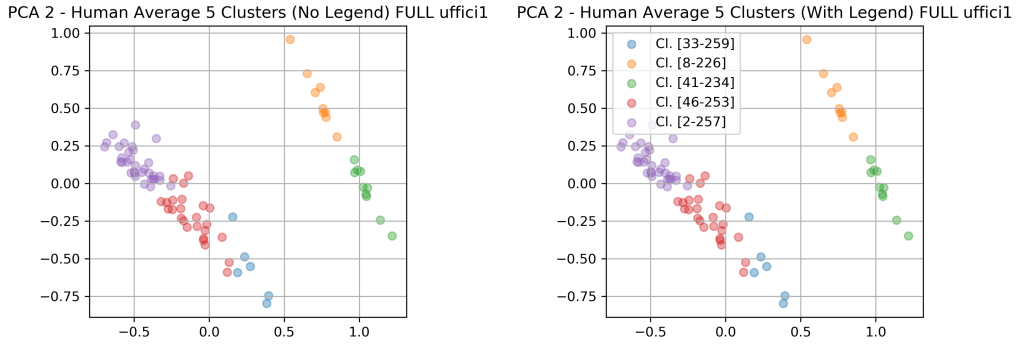

Figure 4.33: PCA - human clusters - map uffici1.

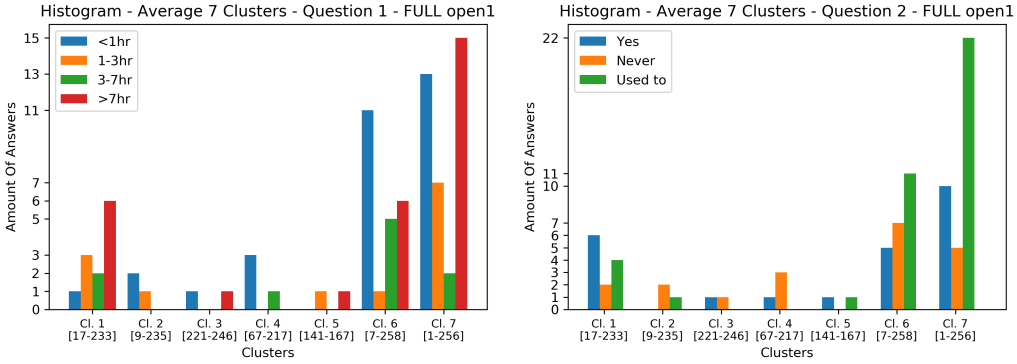

Figure 4.34: Survey answers - map open1.

As far as map **open1** is concerned, in Figures 4.34 we can see how the answers of the survey (Section 2.4.1) are distributed for all 7 clusters. Based on those answers and on a visual inspection of the trajectories we characterise each of the 7 clusters:

- **Cl. 1 - Results 17-233:** lucky gamers[1] who immediately find the target.
- **Cl. 2 - Results 9-235:** non-gamers[2] with no sense of direction, who do not find the target and have no confidence with the FPS control scheme.
- **Cl. 3 - Results 221-246:** people[3] with no sense of direction, who do not find the target, but that are at ease with the FPS control scheme.

[1]Someone who plays FPS games.
[2]Someone who has never played FPS games, but might have played other genres.
[3]They includes gamers, non-gamers, and former gamers.

Centroid	Average
17-233	17-233
9-56	9-235 without 235
221-246	221-246
67-217	67-217
141-167	141-167
7-258	7-258
1-256	1-256 without 73
73	N/A
235	N/A

Table 4.7: Centroid vs average - open1.

- **Cl. 4 - Results 67-217:** non-gamers, plus one gamer, with no sense of direction, who manage to find the target by brute-force.
- **Cl. 5 - Results 141-167:** gamers with no sense of direction who find the target by brute-force.
- **Cl. 6 - Results 7-258:** people with an apparently good sense of direction.
- **Cl. 7 - Results 1-256:** people with a variable sense of direction.

We highlight the similarities between clusters 4 and 5: the two are distinct mainly by the type of players, non-gamers in one case and gamers in the other, and by their *completion time*. This is interesting since it suggests that either being a gamer or not does impact the performance in a noticeable way, however, we cannot state this due to the limited cardinalities of the involved clusters. Cluster 4 contains 4 trajectories and cluster 5 only 2, these numbers are too low to let us draw any conclusion on this matter.

As an additional check, in Table 4.7 we compare the previous 7 clusters against the 9 clusters obtained via centroid method. We can see that the two are extremely similar, only the two clusters containing just one single trajectory being different.

By following a similar approach we analyse the resulting clusters for map **uffici1**; in Figure 4.35, we report the answers to the survey (Section 2.4.1). Questions 3 and 4 concern the last map of each group, as such they are not available for the first map. Based on those answers and on a visual inspection of the trajectories we characterise the 5 clusters:

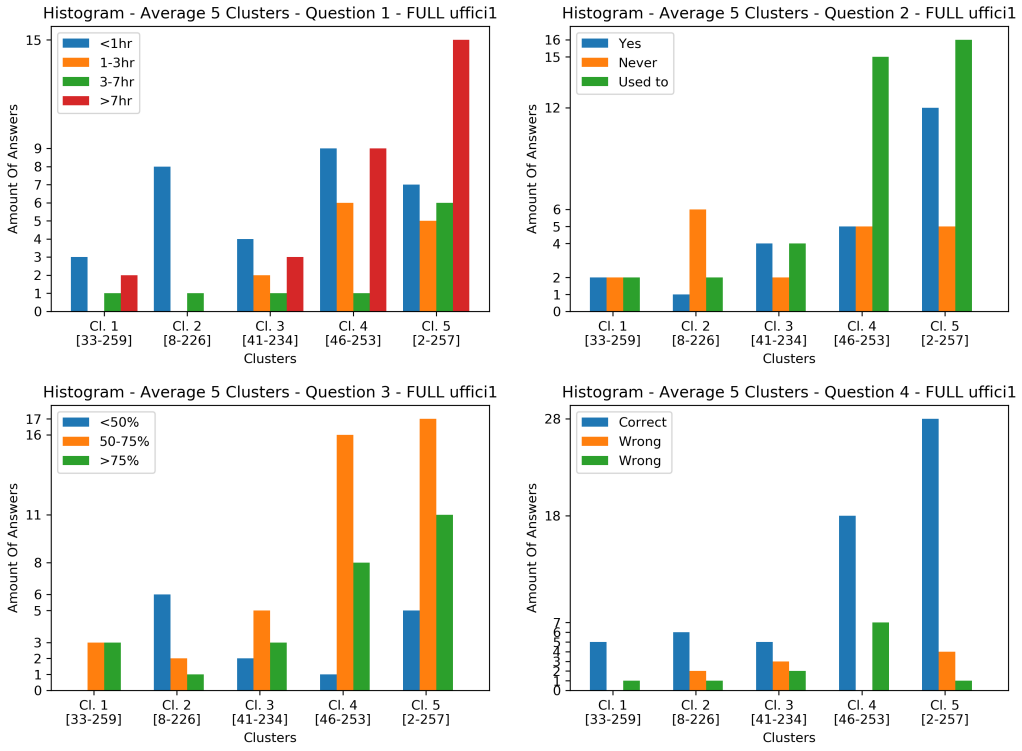

Figure 4.35: Survey answers - map uffici1.

Centroid	Average
146-259	33-259 without 33, 69
8-226	8-226
41-234	41-234
33-253	46-253 with the addition of 33, 69
2-257	2-257

Table 4.8: Centroid vs average - uffici1.

- **Cl. 1 - Results 33-259:** people capable of getting a general idea about their surroundings; they use this idea to meticulously explore the map in order to find the last remaining target, they do not go to unexplored areas straight away.

- **Cl. 2 - Results 8-226:** non-gamers that explore only parts of the map due to their lack of confidence with the FPS control scheme. This results in a bad mental representation of both the map layout and the amount of explored map.

- **Cl. 3 - Results 41-234:** people, but only few non-gamers, with no sense of direction and no ability to mentally reconstruct the map from their surroundings.

- **Cl. 4 - Results 46-253:** mainly former gamers[1], although some gamers and non-gamers are present too, that have a generally good but not always precise idea about the explored environment.

- **Cl. 5 - Results 2-257:** gamers or former gamers with a good sense of direction and a good ability to reconstruct the map from their surroundings. They use these abilities to go straight to the unexplored areas, without the need to rely on a meticulous exploration.

As an additional check, in Table 4.8 we compare the previous 5 clusters against the 5 clusters obtained via centroid method. We can see that the two are extremely similar, 2 trajectories out of 83 are the only difference.

Based on our results we can say that **hierarchical clustering with average method and the Full set of features** described in Section 4.6.2 **is a viable approach to tackle the clustering of human trajectories** in a flat virtual environment with limited visibility.

[1]Someone who used to play FPS games.

4.7 Clustering conclusions

At this point we can conclude that the **geometric-based metrics** (Section 4.3) **are an appropriate** tool **when** dealing with the clustering of trajectories under the assumption that **we want them to be clustered according to** their **geometric paths**. We need **higher level features** (Section 4.6) if we want **to successfully cluster trajectories based on the players' ability**, for example, separating the players who walk over the same locations multiple times from the players who never cross the same spots twice. **HMMs** (Section 4.4) can be used to represent the behaviour of an agent, but they **have not proved successful for our clustering purposes**; despite that, they have provided new tools that will be used in the next chapter. The **survey is useful in order to get more insight on the composition of each cluster** (Section 4.6.5); it is **not useful for clustering** (Section 4.5), but this may depend on the specific questions.

The 9 measures (Sections 4.5.2 and 4.6.2) have proved enough to get good clusters, but **it is likely that these results may be improved by adding or changing one or more features**. It is possible to compute new features based on the original trajectories instead of the Voronoi equivalents, but the sampling procedure (Section 2.4) approximated each position to the nearest integer, thus some precision is lost and cannot be regained.

Furthermore, we assumed that the less an agent walks over the same positions, the better the exploration is. This holds true for the most extreme cases, but it is not true for intermediate ones. By talking to some of the testers we discover that they go back and forth deliberately to cover any possible spot on the map; e.g., Figure 4.36. Their exploration is not optimal according to *percentage of optimal exploration*, but actually it can be considered optimal.

Our features work on flat maps, we expect that if the maps were multi-floor then those features would not be appropriate. For example, if a map had many floors and only one staircase, then that staircase would become a bottleneck over which all agents would likely walk several times. A similar situation can be seen in map uffici1 where the two corridors connecting the upper part of the map with the lower part are much more likely to be explored multiple times w.r.t. the rest of the map.

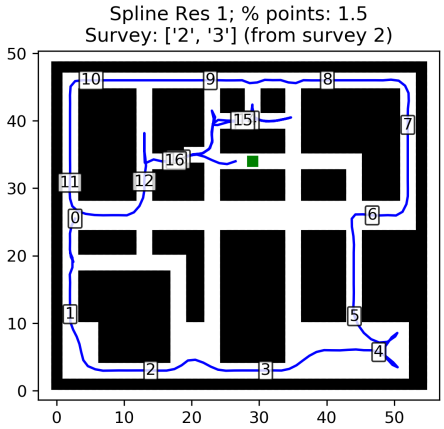

Figure 4.36: Splines of Voronoi equivalent of Result 1.

Chapter 5

Trajectory reproduction

In this chapter we evaluate several techniques to reproduce human trajectories belonging to a cluster. Firstly, we consider simulated annealing, then we take a closer look at the grid search where we highlight the consequences of non-determinism in robot's behaviour.

In order to check our results we perform some experiments using random values, after that we try a possible improvement over grid search in the form of a custom robot distribution exploration procedure.

Then, we expand our analysis on the robot behaviour by evaluating how much it changes, and in turn how much the features are impacted by those changes.

We inspect which of the clusters in Section 4.6 can be effectively reproduced by our robot, and we comment on the nature of the non-determinism in the light of the information gathered in this chapter.

Lastly, we consider how hidden Markov models can be useful for our purposes, before ending the chapter with a conclusion that sums up our results and highlights their limits.

5.1 Robot behaviour - introduction

While bug-fixing the newly implemented functionalities in Unity (Section 3.2.2) we noticed that the **behaviour of the robot** was **not deterministic**. At that time the extent of its non-determinism was not clear. This is an issue that impacts over most of the considerations in this chapter. Here we briefly mention it, we will expand it as needed in the rest of this chapter.

5.2 Simulated annealing

In order to speed up the identification of the best robot parameters to imitate a group of trajectories (Section 2.4.3), we try to look for faster alternatives to grid search. The first attempt is to use **simulated annealing**. The annealing function iteratively submits a parameter combination to the objective function. Each combination is passed to Unity which runs the robot simulation and collects the Result. Once the Result is collected the Python code working as objective function retrieves the Result, converts it to Voronoi, computes the distance according to the specified feature set (Section 4.6.2), and returns that distance to the annealing function. The annealing function is implemented in SciPy (Section 4.2) [21]. However, this approach has **two problems**.

The first one is that for each parameter combination we must wait for the robot to complete its exploration, this means a worst case scenario of 8 minutes per robot. We cannot precompute all possible trajectories because the amount of possible parameter values is no longer limited, α and β are continuous between 0 and 1. Our main goal is to speed up the process w.r.t. grid search, but our annealing function has a recommended maximum of 10^7 objective function calls. 10^7 is much greater than 484, which is the number of parameter combinations in grid search (Section 2.4.4), so **we are slowing down our search for the best parameters**.

The second problem is that this approach works if the distance is consistent across each iteration. This is the case with the metrics (Section 4.3.1), but it is no longer the case with the measures (Sections 4.5.2 and 4.6.1). The issue is the normalisation step: as new robot trajectories are collected the maximum value of a feature might increase, this variation changes the normalised values, which in turn change the older distances our code has already returned to the annealing function. Being already returned, they cannot be changed. This **prevents us from using the features** in a similar fashion w.r.t. what we did in Section 4.6.

Both **problems are the reason why we discard this approach**, however, the newly implemented tools to let Python provide parameter values to Unity, and let Unity provide Results to Python in a fully automated fashion will be useful in Sections 5.6 and 5.7.

5.3 Grid search

In Section 2.4.4 we described the grid search as implemented by L&Z; at the end of Section 3.2.2 we defined the changes implemented during this thesis. We perform the **grid search** for map open1 and uffici1. We keep the time scale set to 1, i.e., real time, and **we allow 7 robots to be active at the same time**. Theoretically any number of robots can be used, as long as it is below the amount of possible combinations of parameters values, i.e., 484. We choose such a low number due to hardware limitations: 7 robots is the maximum number of robots that we can have in a map without constantly maximising the CPU utilisation of our computer, a 2018 Mac mini with a six cores i5 8500B CPU, an Intel UHD Graphics 630 integrated GPU, and 8 GB of RAM. The reason we do not want to constantly hit the maximum clock is due to previous experience with Unity. Unity draws n frames per second (fps, non to be confused with first person shooter), in order to do so it runs all occurrences of functions `Update()` and `FixedUpdate()` in all scripts attached to active objects inside the current scene. Those functions contain the code that tells Unity what should be represented in the current frame.

In our case each robot has its own functions, and the higher the number of robots, the higher the number of functions. If we set Unity to n fps it means all functions must be completed within $\frac{1}{n}$ seconds; with our choice of 30 fps it means a frame time of 33,3 ms. During a previous project we have experienced situations where overshooting that value would not only degrade performance due to stuttering, but also prevent Unity from fully executing the code inside the functions. This could potentially change the behaviour of the robots, thus in order to avoid introducing any potential non-determinism **we limit the number of robots to a value that our computer can handle**.

484 total parameter combinations and 480 seconds of maximum allowed time (Section 3.2.2) mean a worst case scenario of about 65 hours, or 2 days and 17 hours, to complete **a full grid search**, assuming only one robot is used. In our case it **requires about 4 hours for map open1 and about 9 hours for map uffici1**. The first map requires less time since more robots manage to find the target, thus they are able to complete their exploration faster than in the second map. Due to the robot's non-determinism described in Section 5.1, we run the grid search 2 times for each map, in both occasions the overall time is the aforementioned one. Since **we have 2 rounds of grid search for each map** we distinguish them by calling them **"r1" and "r2"**, we add the round number after the map name.

Once all trajectories are collected we convert them to Voronoi and we

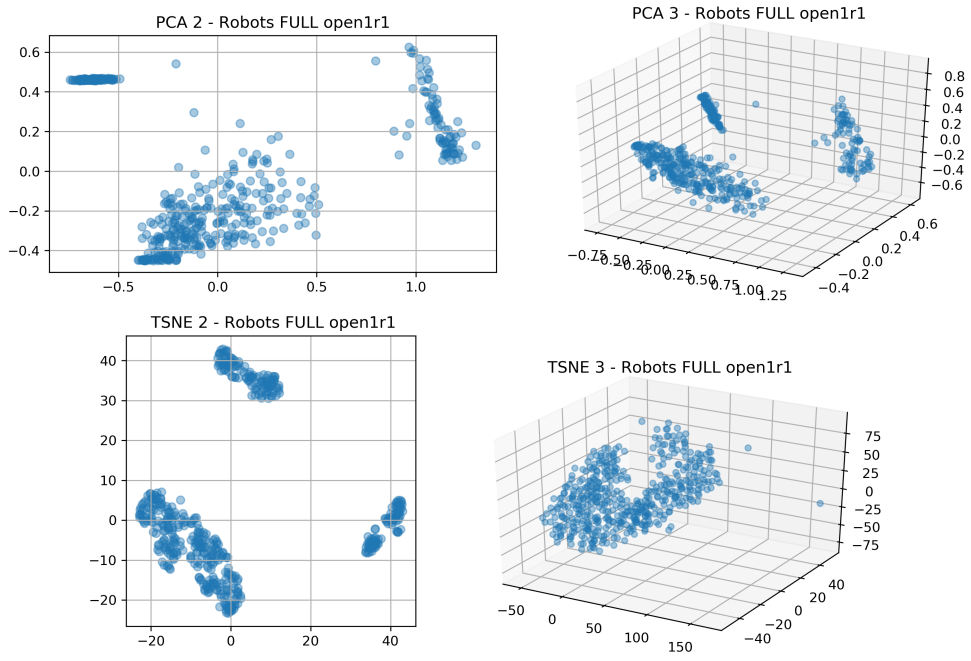

Figure 5.1: PCA and t-SNE robot Results - map open1, round 1.

extract the same features described in Sections 4.5.2 and 4.6.2. We try to cluster them to **see if we can spot any group based on the parameter values**. In Section 4.6 we decided to use average method (Section 4.2) with set Full (Section 4.6.2), thus we use them to compute PCA and t-SNE. In Figure 5.1 we can see PCA and t-SNE for the first round of grid search in map open1. In Figures 5.2 and 5.3 we can see the resulting dendrograms. By visually inspecting them **we cannot find any pattern**.

In Figures 4.28 and 4.29 we showed how the robot Results of grid search arrange themselves in the reduced space. We try to visualise the values of the parameters for each robot Result. In order to do so we assign different transparency levels to the eleven possible values of α, we assign different colours to the eleven possible values of β, and we assign different shapes to the four possible values of δ. One of the resulting images is shown in Figure 5.4 alongside the original PCA. Again, **we can find no group**.

Since PCA images with all robot Results are quite difficult to read, we also plot PCA with only one robot Result at a time. By **comparing a robot with the same parameters on rounds 1 and 2**, we discover that sometimes the robot is placed in the same area of the PCA graphs, other times it is not. **By looking at the trajectories we can see that they**

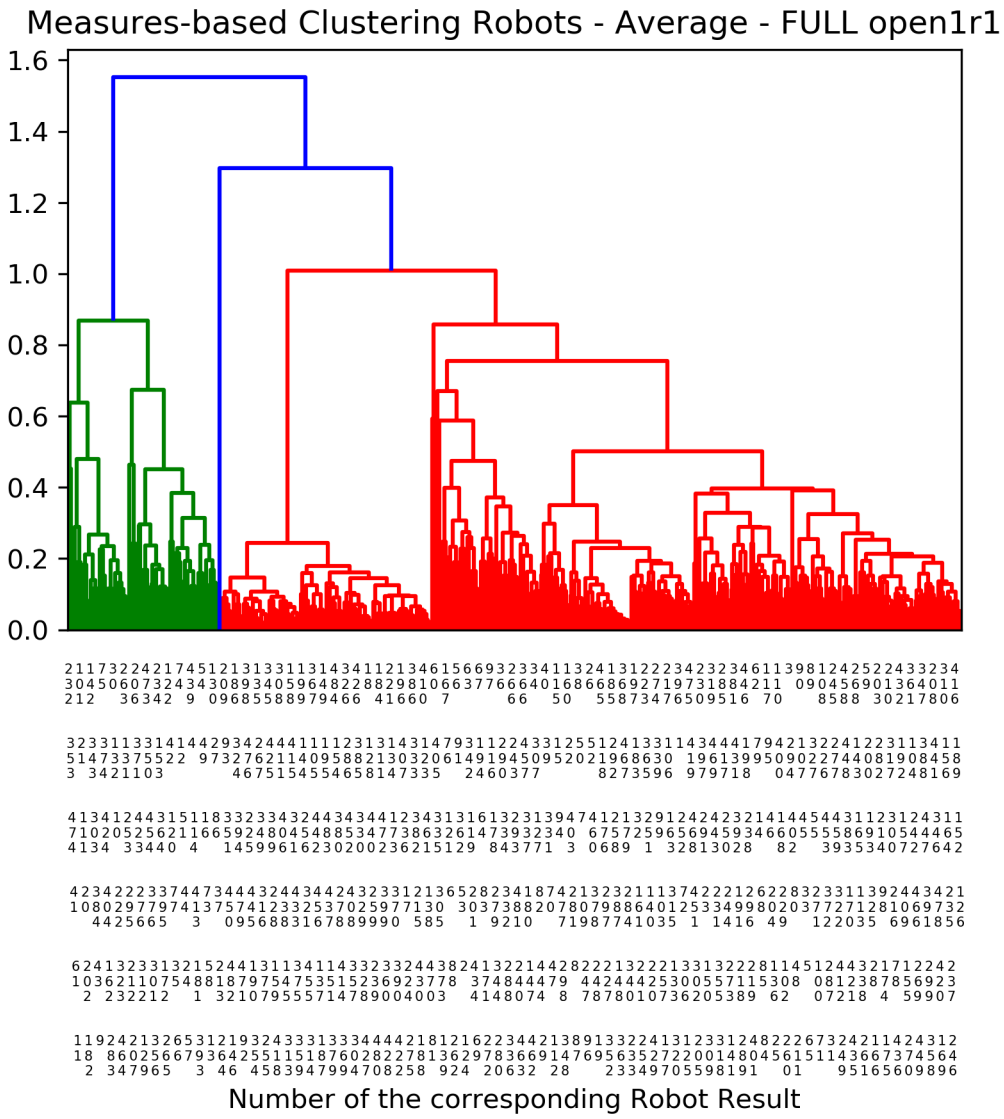

Figure 5.2: Dendrogram robot clustering - average - Full - map open1, round 1 - robot Result numbers.

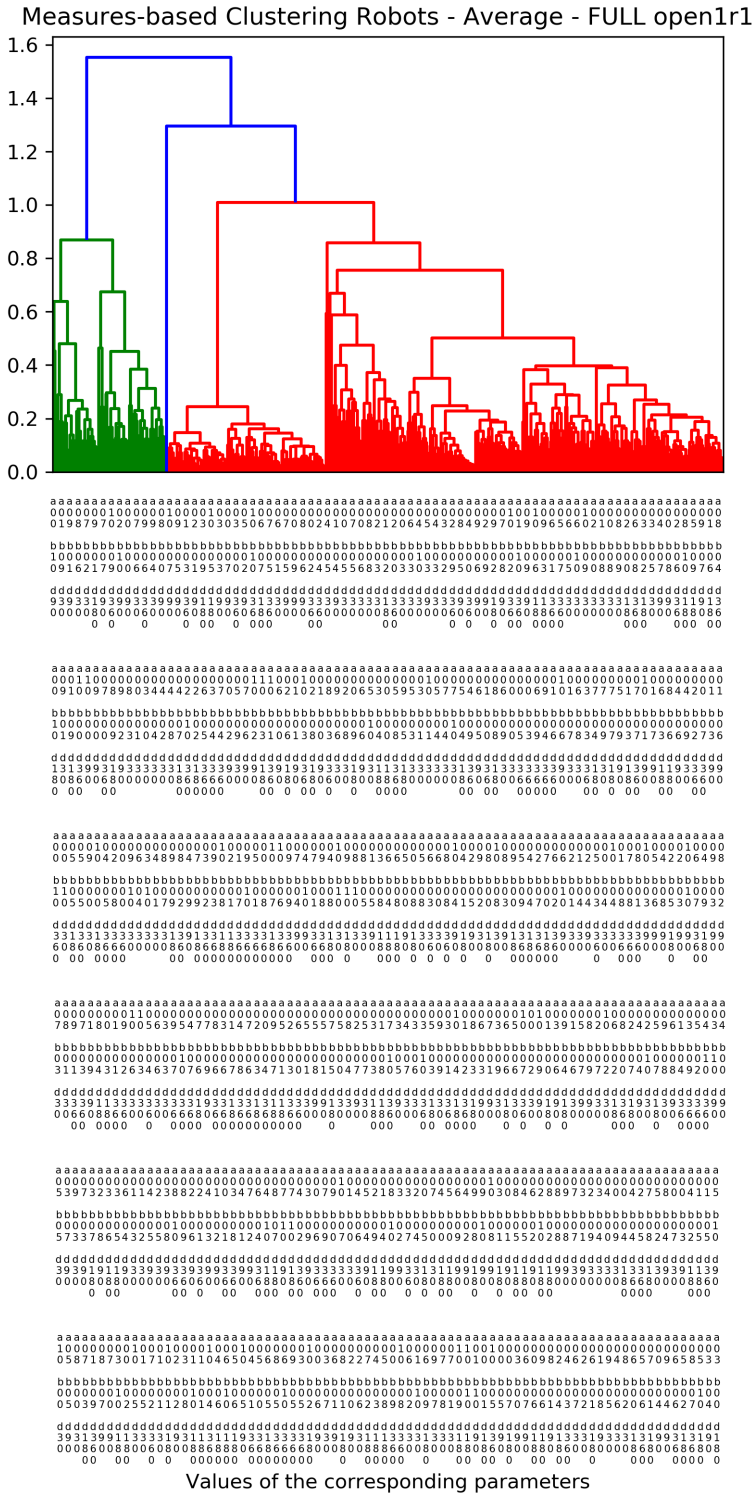

Figure 5.3: Dendrogram robot clustering - average - Full - map open1, round 1 - robot Result parameters.

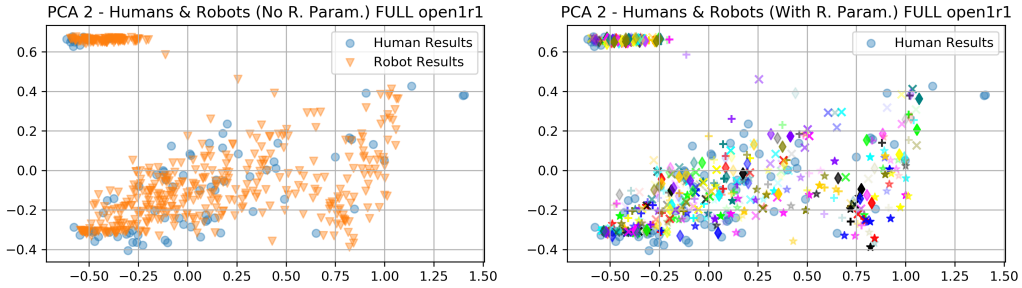

Figure 5.4: PCA - humans and grid search robots with and without robot parameters - map open1, round 1.

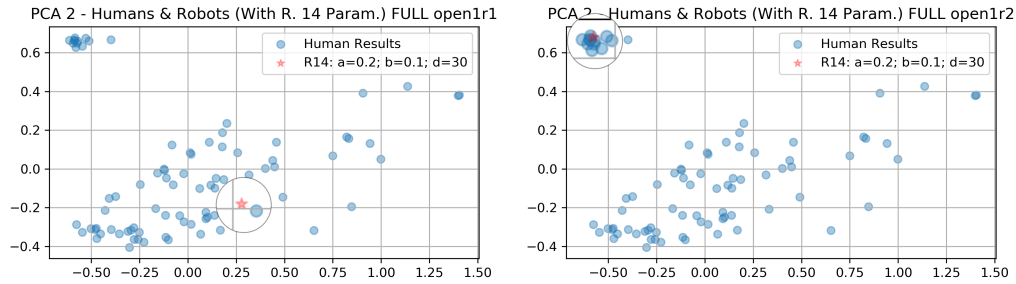

Figure 5.5: PCA - humans and grid search robots with a focus on robot Result 14 - map open1, both rounds.

differ between the two rounds, sometimes the changes are minor, other times they are major. In Figure 5.5 we can see an example where the same parameters led to very different trajectories, in the images we highlighted the two robot Results with a magnifying glass.

What we have just discovered means that **we cannot assume that a parameter combination will always result in the same (good or bad) trajectory**. However, if we compare the PCA graphs of the two rounds we can see that they are actually similar once all 484 trajectories are displayed; an example with PCA in 2 dimensions for map open1 can be seen in Figure 5.6. What this means is that **even if the robot's behaviour is not deterministic, we can still assume to cover more or less the same feature space after collecting all 484 trajectories**.

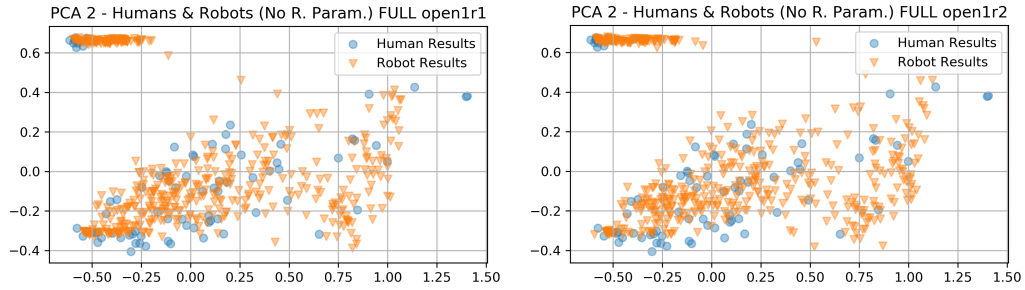

Figure 5.6: PCA - humans and grid search robots - map open1, both rounds.

5.4 Robot behaviour - conclusions

In previous section we analysed the behaviour of the robot in order to better understand the extent of its non-determinism. Now we focus on the Unity implementation in order to find some of the causes.

In Section 5.3 we described how Unity works. In addition to `Update()` and `FixedUpdate()` Unity runs several other scripts in each frame time. For example, some events trigger functions that need to be executed. Each time a function is triggered it is added to the list of functions to be executed during the next frame. This means that if a timer expires and triggers a function, that function is not executed until the computation of next frame starts. We have multiple timers, one of them, for example, stops the exploration after 480 seconds, while others handle the perception and decision processes (Section 2.4.3). Each time those two timers expire in the same frame time, their functions are scheduled on the next frame, creating a race condition. The perception process updates the frontier, thus executing it before or after deciding where to go next may yield a different result.

Each time the robot utility (Algorithm 1 page 10) checks the distances it does so using the current position of the robot. Current means the one in the frame where the distance computation takes place. If we explored two times using the same parameter combination, then, due to approximations, it would be possible that the two robots would be in slightly different positions. These minor changes impact the distance computation, potentially changing which frontier will be explored next. One different decision at the appropriate time has major effects over the final Result.

All these are potential causes of non-determinism. They can be solved, but the amount of required work is not negligible.

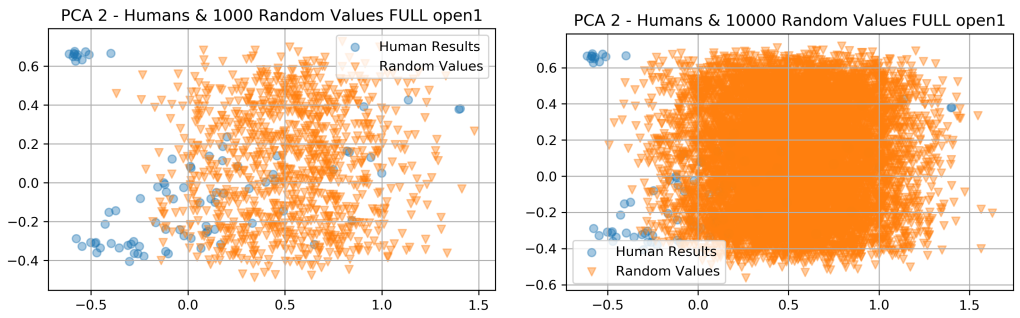

Figure 5.7: PCA - humans and random values - map open1.

5.5 Random features

We want to check if what we discovered in previous section is reasonable: the risk is that all robot Results were displayed over the same areas of the PCA graphs because those areas were the only ones allowed. To check if that is the case we perform PCA using random values for all features. This approach provides several combinations, including some that are not physically possible. All features are sampled uniformly in the interval $[0; 1]$, with the only exception of *has found all targets* which is either 0 or 1.

In Figures 5.7 we report the PCA of human Results from map open1 with 1000 and 10000 random values. These values are converted from the 9-dimensional feature space to the reduced space the same way robot Results' features are. In the figure we can see that there are areas covered by the random values that are not covered by humans or robots, thus since the robot Results were placed closer to the human ones **we can conclude that different executions of grid search do cover the same feature space**.

Figure 5.8 shows the results for uffici1; we can see that even randomly generated values are placed either in one group or in the other. This makes us check if *has found all targets* is what separates the leftmost human cluster from the rightmost one, and we find out that that is the case. The same is true if we consider the 484 robot trajectories; we report the two non-random PCA graphs in Figure 5.9. Looking at the robot distribution w.r.t. random values we can see that **grid search Results cover a feature space that is closer to the human one**, even though it is not as overlapped as in the open1 case.

We can conclude that robots are placed in the PCA graphs only on some areas because those are the actual areas covered by the features derived from

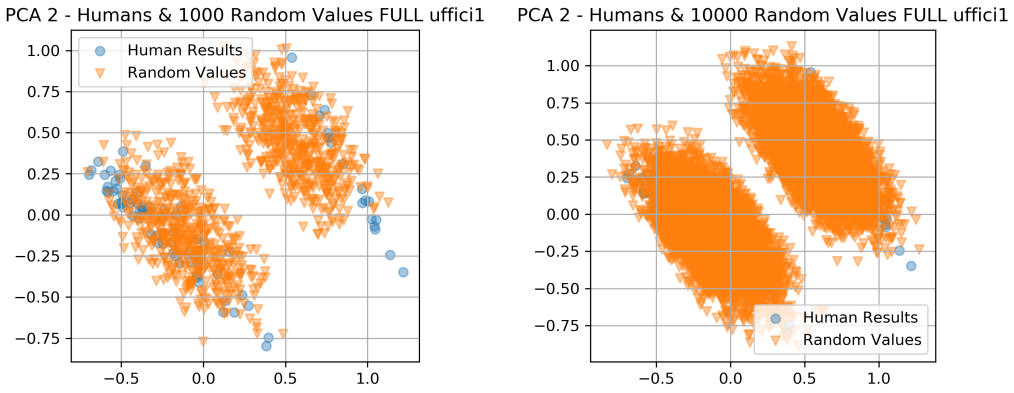

Figure 5.8: PCA - humans and random values - map uffici1.

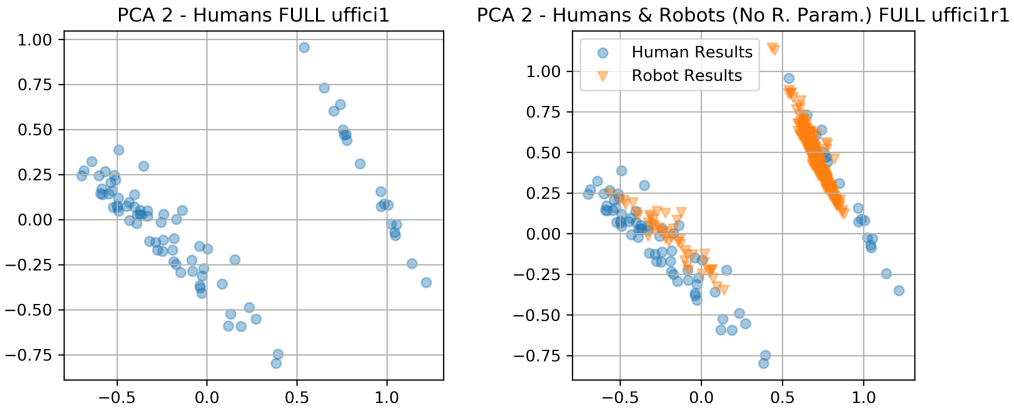

Figure 5.9: PCA - humans, and humans with grid search robots - map uffici1, round 1.

the robots' Results; PCA could fill some of the remaining empty areas if the features were different. This means that **PCA is a reliable tool to be applied to our problem**.

5.6 Robot distribution exploration

So far Grid search has proved to be the best alternative, we know that its main drawback is the lengthy iteration process, and we also know that different explorations with the same combination of α, β, and δ does not guarantee a trajectory with the same qualities. **We** combine these two pieces of information to **devise a procedure that we called robot distribution exploration**, it is the following:

1. Select which are the human trajectories that we would like to imitate.
2. Randomly choose a total of 50 combinations of the robot parameters.
3. Use the same code of simulated annealing (Section 5.2) to provide those combinations to Unity, this time they are all provided at once.
4. Wait for Unity to complete one robot exploration for each combination; 7 robots are enabled at the same time.
5. Collect the new 50 robot Results.
6. Compute the normalised features for both humans and robots.
7. Compute the human centroid based on the normalised features.
8. Compute the distance of each robot from the human centroid.
9. Pick the 10 best results, i.e., the ones that are the closest to the human centroid, among all robot Results collected so far.
10. Check how many times we have performed step 11. If it has already been done 2 times, then go to step 13.
11. For each of the 10 best Results, generate 5 new combinations of values using Gaussian distributions. Each of the three parameters α, β, and δ is sampled from its own Gaussian distribution whose mean is the α, β, and δ of the best Results. The standard deviation is 0.04 for α and β, and 0.5 for δ; they are such that we stay close to the best values, while allowing room for exploration. Once this process ends we have 50 new combinations.

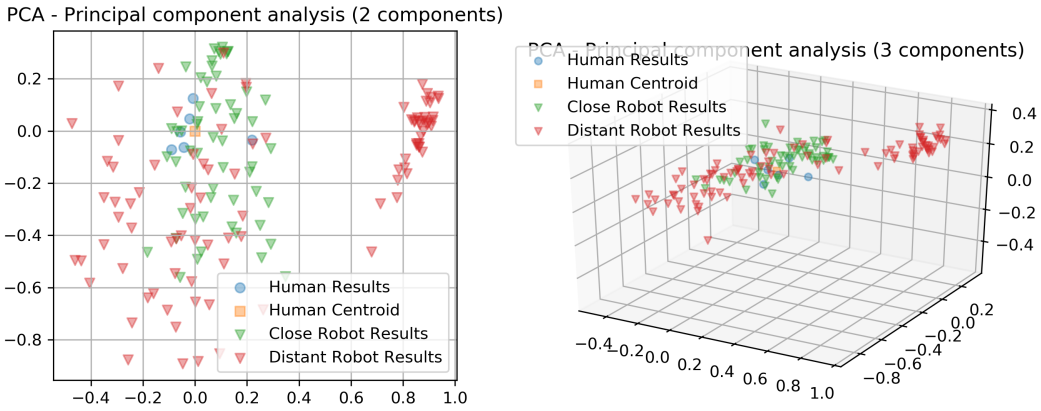

Figure 5.10: PCA - Robot distribution exploration - map open1.

12. Go to step 3.

13. Order all 150 trajectories from best to worst.

At each iteration of step 9 all trajectories are considered, including the ones from previous iterations, and all features are normalised again from their original values. This way, the 10 best Results can be from all iterations, ensuring we focus more on values that have proved more successful. We are aware that in Section 4.6 we preferred the average method over the centroid one, however the two were very similar and for our current needs centroid is faster.

Once the procedure is over we have 150 trajectories instead of the 484 of the grid search, but **we** also **have some robot trajectories** that explored the virtual environment in a **similar** fashion w.r.t. **the provided human trajectories**. In Figure 5.10 we can see an example using human Results 145, 173, 187, 189, 215, 258 as trajectories of interest. In that figure, *close* and *distant* identify the two halves of the robot Results based on the distances between each trajectory and the human centroid.

The advantages of this approach are: its **flexibility**, all the values used by our implementation can be changed, e.g., more or less combinations at each step can be generated; and its **speed compared to grid search**. In our tests we get an average completion time of 1.5 hour for map open1, this means we spent 37% of the time of a full grid search (about 4 hours, Section 5.3) to generate the 30% of the trajectories of a full grid search. **The final result is comparable, but the amount of required time is more than halved.**

5.7 Robot variance analysis

Previous approach has proved useful, but **we want to know how much different robot explorations sharing the same parameter combinations can be**. For this purpose we manually select 10 combinations for each map using the PCA of humans and robots. The idea is to have one combination for each area of the PCA graph. Then we run 10 explorations for each parameter combination. At the end we compute the features for all Results and we plot them in box plot graphs. In addition, we also plot a PCA for each combination.

In Figure 5.11 we report a couple of features: *average distance between repeated positions* and *completion time*. The first column shows the feature values for all 484 Results of the grid search. The other columns show the values for each parameter combination. For each column the combination and its corresponding Result number are written on the horizontal axis; we highlight Result 248 and Result 263. The former is the sixth column, i.e., the central one, while the latter is the one with the most compressed box. Based on these graphs we can expect that the PCA of Result 248 will show a much greater variance than the 263 one. That is the case, as we can see in Figure 5.12.

Somewhat similar results can be obtained for the other map, however in this case all features are more compressed since most of the robots do not find all targets. Checking these new results we notice something unexpected. As we can see in Figure 5.13 there are robots that manage to complete the exploration before the time runs out, so they have found all targets, however all PCA graphs are like the one in this figure: they show all robot Results only on the rightmost macro cluster. What this means is that our statement in Section 5.5 about **the two macro cluster in map uffici1** is wrong: they **are distinguished mainly by the boolean feature *has found all targets*, but not exclusively by it**. This is good since it shows us that the macro clusters do not depend only on one feature.

At the end of our experiment we can state that the degree of variation of each feature is generally very high, this makes **using Gaussian distribution in step 11 of the robot distribution exploration procedure** (Section 5.6) **less useful than expected**, in fact, just using 150 uniformly generated values combination is likely to be just as effective as our implementation.

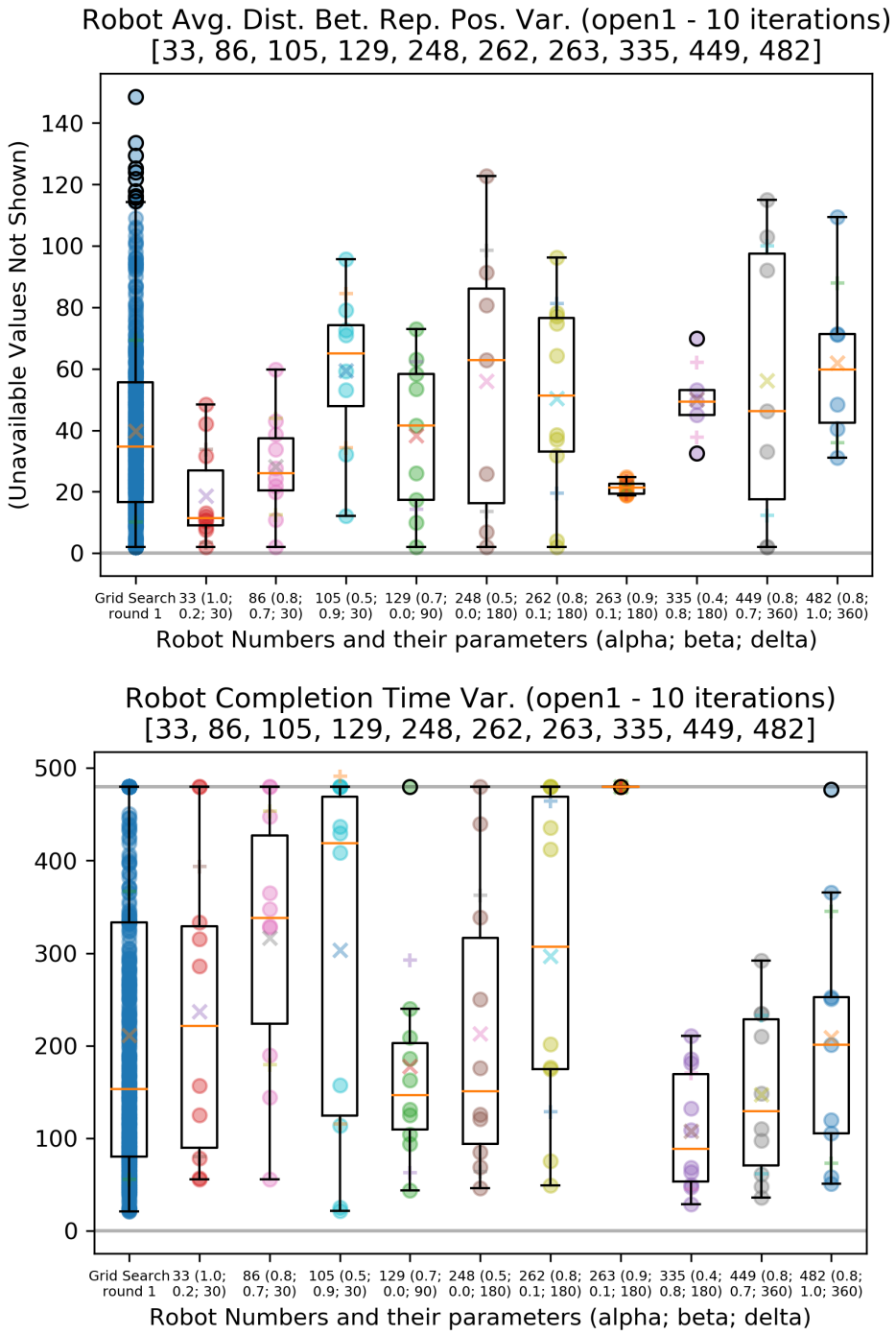

Figure 5.11: Box plots of variations of two features of 10 robot Results - map open1.

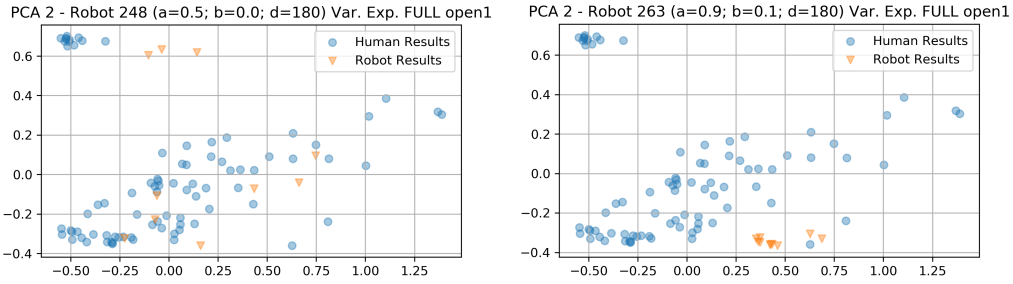

Figure 5.12: PCA - Robot Results 248 and 263 - map open1.

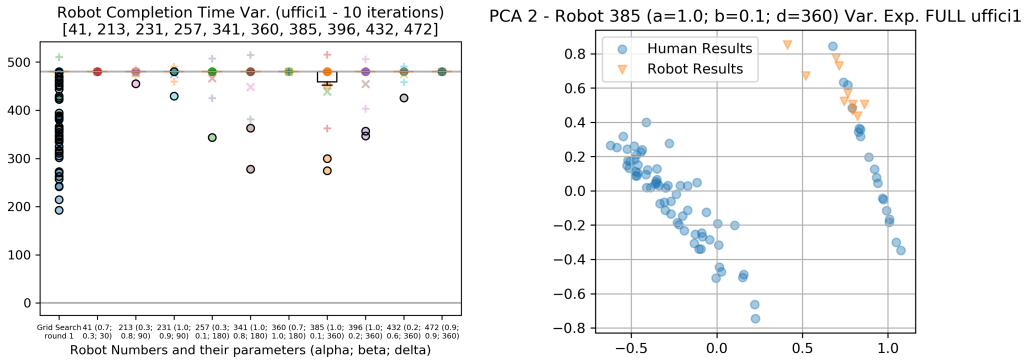

Figure 5.13: Box plots of variations of completion time of 10 robot Results and PCA of robot Result 385 - map uffici1.

5.8 Robot features analysis

In previous section we focused on the Results of several explorations with few parameter combinations, now we focus on all 484 grid search Results. We collect $484 \times 2 \times 2$, i.e., 1936, trajectories by performing 2 rounds of grid search for each map, then we plot the features in **three types of graphs**:

- **Normalised features humans and robots** graphs show all features for both humans and 484 robots of one round of grid search. Features are on the horizontal axis, for each feature humans are on the first column, robots on the second one. The normalised values of the features are on the vertical axis.

- **Single parameter** graphs show how the values of each feature are distributed depending on each parameter. This results in three graphs for each feature, so 27 images for each round of each map. Box plots are used to help visualise the distribution, a line connects the mean of each column identified by an X, two + are used on each column to display $mean \pm standard\,deviation$.

- **Cube helix** graphs rely on the cube helix implementation of seaborn [22] to represent variations in the values of each feature. Each feature has an image containing four graphs, one for each value of δ, while α and β are on the horizontal and vertical axes of each graph. The darker the colour inside a graph, the higher the value of the feature.

5.8.1 Normalised features

Given the conclusion we reached in Section 5.3, we expect the **normalised features graphs** to be **similar between each round of grid search**, and that is the case, as we can see in Figures 5.14 and 5.15. Furthermore, we can see that the **feature distribution is more similar between humans and robots for map open1**, but not so much for uffici1. In Section 5.9 we will expand on this.

It is interesting to note that some of the features cover differently intervals. We can explain those differences. Starting with **open1** and Figure 5.14 we can explain the different boxes in *completion time* with the presence of robot trajectories that constantly keep exploring the same areas, increasing the time needed to find the target. The speed is less relevant that it might seem, as explained in Section 3.2.2 the robot cannot move while rotating, a human player can. This allows humans to move at a higher *average speed*

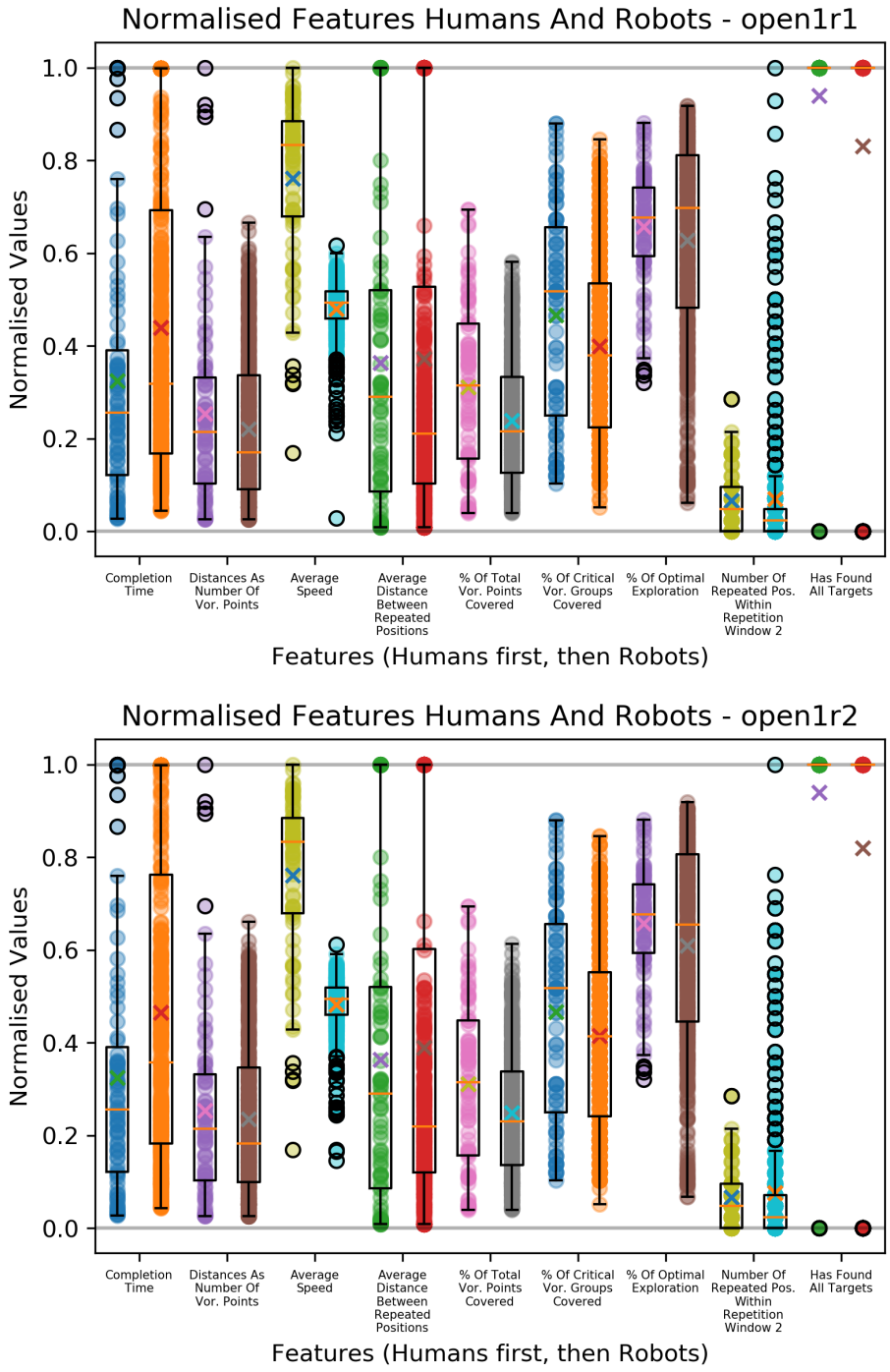

Figure 5.14: Normalised features humans and robots - map open1, both rounds.

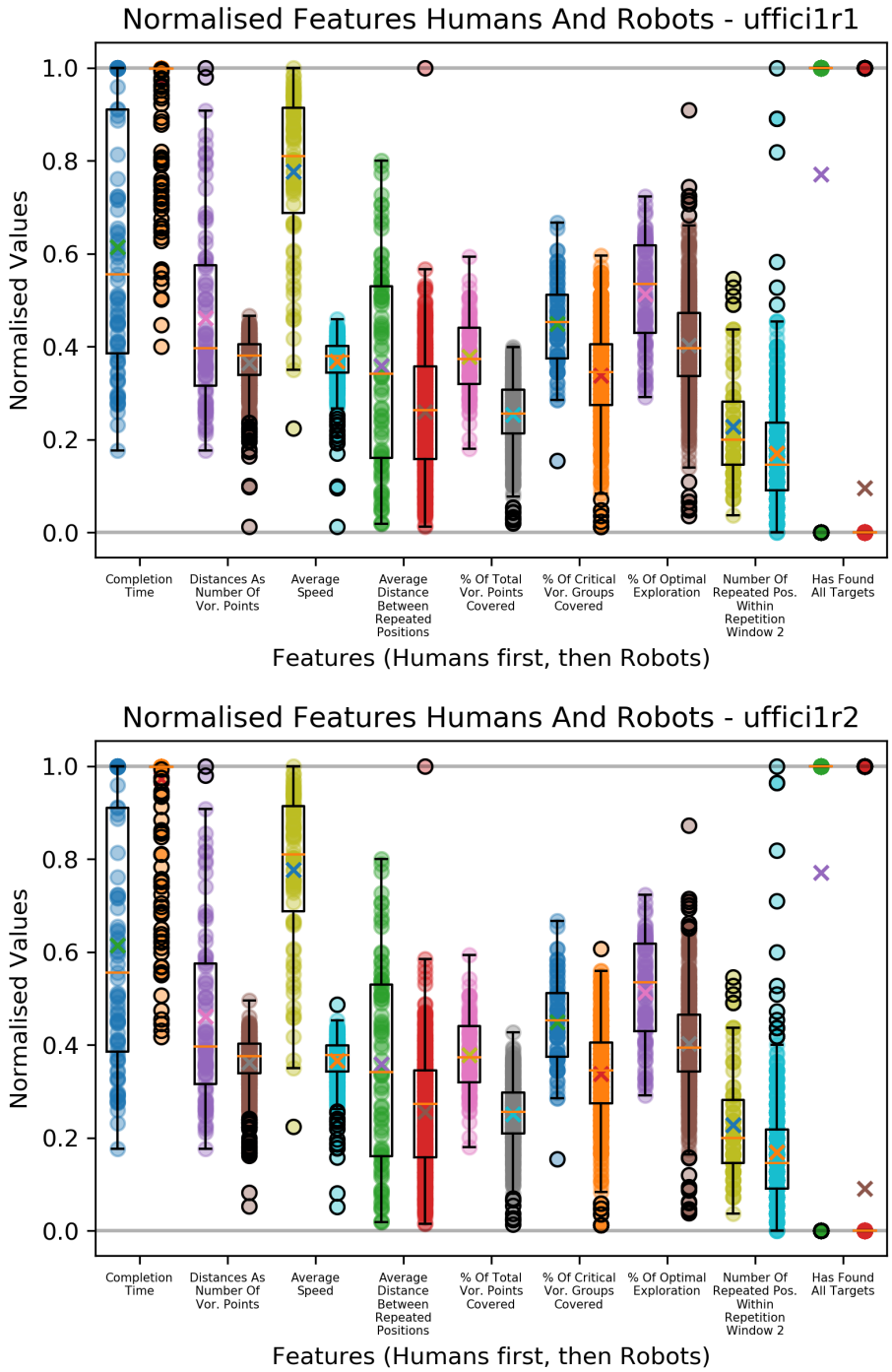

Figure 5.15: Normalised features humans and robots - map uffici1, both rounds.

than a robot. A higher variability in the quality of the exploration can be seen in *percentage of optimal exploration* which has almost the same mean between humans and robots, but covers a broader interval of values for the robot case. Similarly, we have more robot Results with a higher *number of repeated positions within repetition window 2*. Both features can be explained by robots exploring the environment following an extremely bad behaviour, like going back and forth between the same two spots. We highlight the fact that **the mean of *has found all targets* is close between humans and robots**, even though robots are less likely to find the target.

Focusing on uffici1 and Figure 5.15 we see that the boxes representing robots differ more from the human ones than in previous map. For example, looking at *has found all targets* we can see that almost 80% of human player did find the four targets, against the 10% of the robots. These differences are caused by the fact that most of the robots are not able to find all targets, and in fact their *completion time* is so concentrated in the upper bound that any value below 1 is marked as outlier. This means that **the policy we are using to drive the robot exploration is not appropriate for the multi-target map uffici1.**

5.8.2 Single parameter

We compare each pair of single parameter graphs between round one and round two of grid search. **The only consistent results can be found in the graphs of the forgetting factor δ**: as its value increases so does the memory of the robot, which in turn means that the knowledge of the map is widened and thus the robot can explore more efficiently.

As far as map open1 is concerned, in Figure 5.16 we can see an example of graphs for α and δ. α and β graphs are always similar to the reported one: most of the values cover the same band, the trend of the line connecting their means is usually different, but the variations are all concentrated in a small band, as such they are not significant. That is not the case for δ, depending on the feature under consideration the line connecting the means increases or decreases, often with a much bigger variation than α or β.

Similar results can be seen for map uffici1 in Figure 5.17. In this map the features are more compressed, but the increase or decrease of each feature as δ changes are still evident. These changes can be seen in *average speed* as well. The speed at which a robot rotate and move straight are fixed, but as we increase the forgetting factor δ the robot remembers more and more of the map, thus it no longer needs to rotate to travel back to previously explored but forgotten areas because it still knows them.

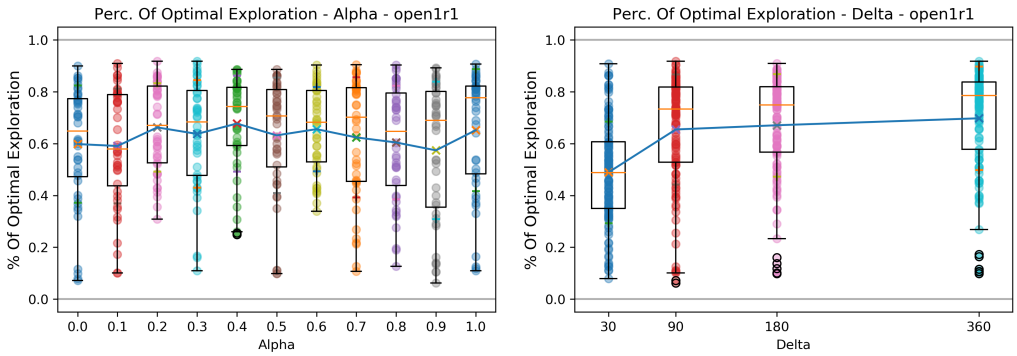

Figure 5.16: Perc. optimal exp. - α and δ graphs - map open1, round 1.

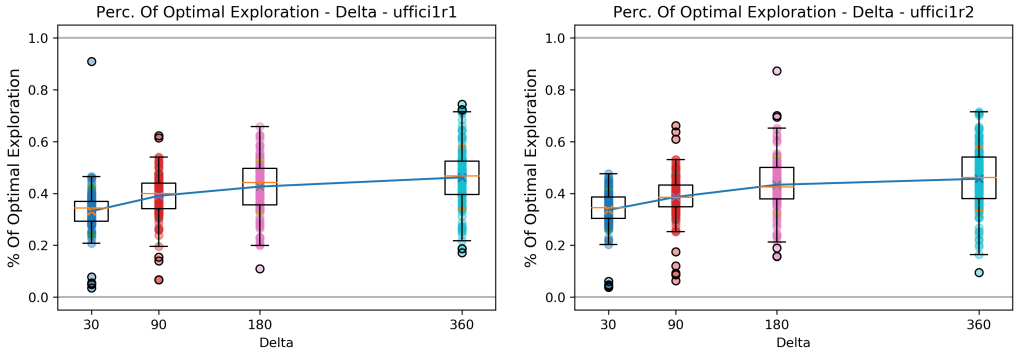

Figure 5.17: Perc. optimal exp. - δ graphs - map uffici1, both rounds.

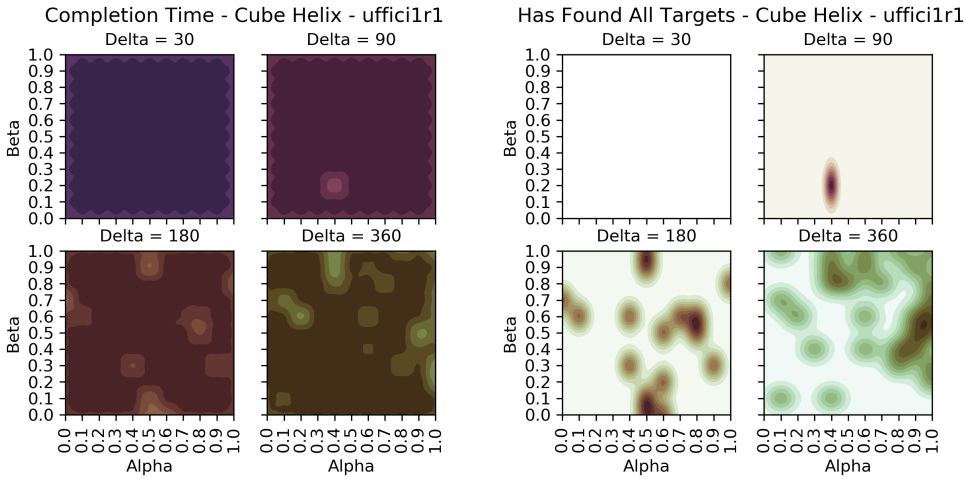

Figure 5.18: Cube helix - completion time and has found all targets - map uffici1, round 1.

5.8.3 Cube helix

We focus on the cube helix graphs that proved more interesting. In Figure 5.18 we highlight the fact that for $\delta = 30$ no robot found all 4 targets in map uffici1, thus *completion time* is the maximum for all parameter combinations. Similar results con be seen for $\delta = 60$, where only one robot managed to find all targets. Results improved once we move to higher δ. This is a different way of viewing what was previously shown in Figure 5.15 in Section 5.8.1. If we consider the second round **in uffici1** we get slightly better results, but the overall trend is the same: **the policy that drives the robot movement is not appropriate** for this map.

While most of the cube helix graphs do not show visible similarities, some of them do, as we can see in Figure 5.19. The similarities are more evident in the graphs of open1, since that map is not filled with small rooms and recesses which force the robot to go back and forth more often and thus increasing the value of *number of repeated positions within repetition window 2*. We can clearly see a trail of darker spots among the main diagonals of all graphs, even in the uffici1 case, despite being less evident. This means that the parameter combinations where $\alpha + \beta = 1$ yield trajectories where the robot reverses the direction of the exploration much more often. *Number of repeated positions within repetition window 2* is the feature where this is more evident, but if we look at the *percentage of optimal exploration* in Figure 5.20 we can still partially see this inefficient behaviour over the main diagonals.

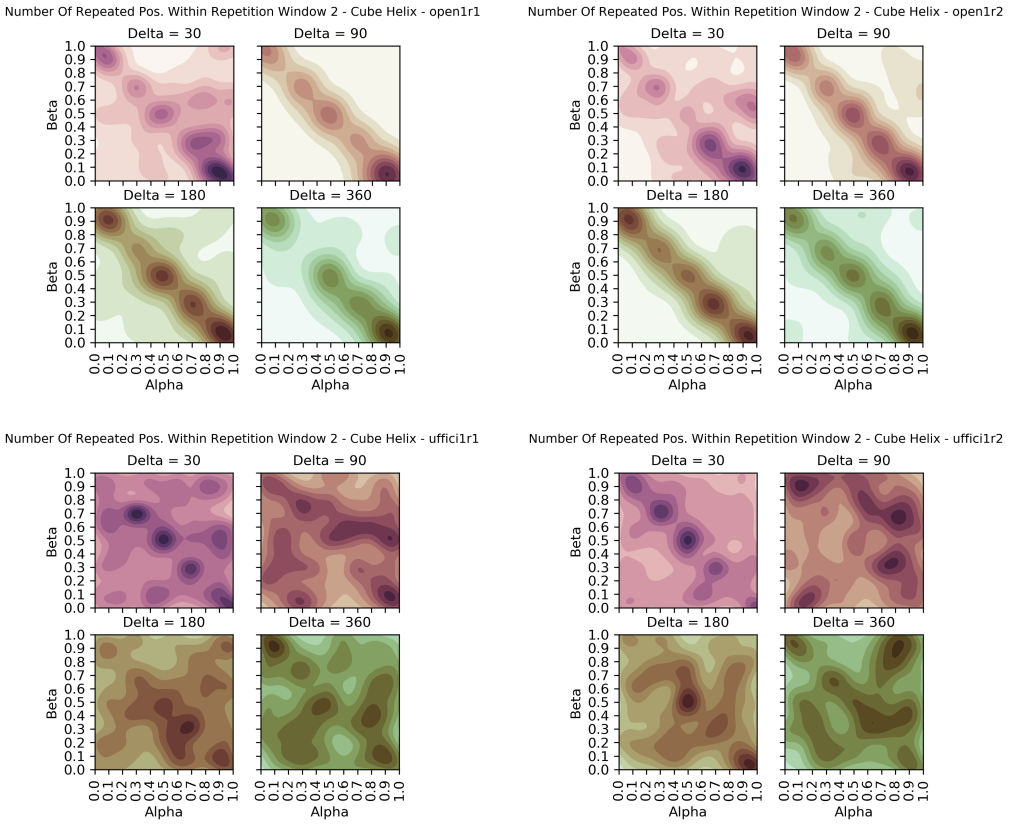

Figure 5.19: Cube helix - number of repeated positions within repetition window 2 - both maps, both rounds.

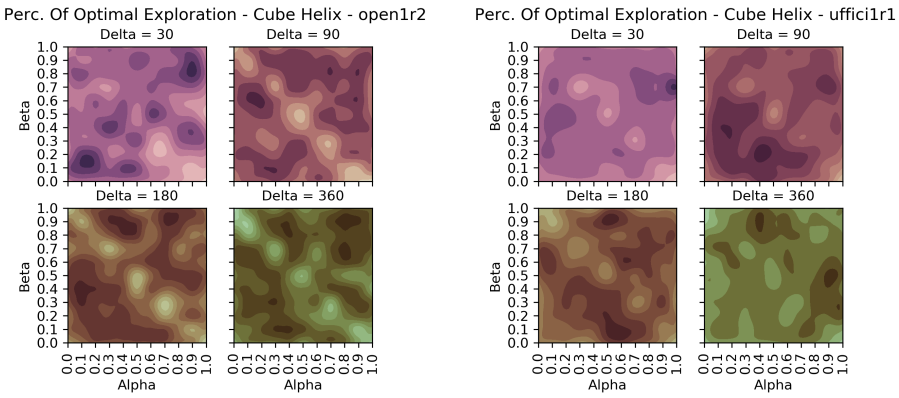

Figure 5.20: Cube helix - percentage of optimal exploration - open1, round 2; uffici1, round 1.

If we look back at the algorithm that drives the robot exploration (Algorithm 1 page 10) we can see that **when $\alpha + \beta = 1$** then $1 - \alpha - \beta = 0$, so the coefficient that multiplies the distance between the robot and the frontier becomes 0. This means that that distance is no longer considered when deciding which frontier should be explored next, which means **the robot is allowed to waste time going back and forth between two or more far away points**.

5.9 Human trajectories reproducibility

Once the grid search is completed we collect the Results. This means **we do not need to perform a new grid search each time we look for the closest robot trajectory to a given cluster of human trajectories**. In order to retrieve the best trajectory we check all features of all Results from a round of grid search. We consider one robot trajectory at a time, we compute its distance from each human trajectory, we average them and we compare that mean with the best one found so far. We select the robot trajectory whose average distance from the human trajectories in a cluster is the smallest. In addition, we also select the robot trajectory which has proved to be the closest to one of the given human Results; in this case we provide both robot Result number and human Result number, this way it is possible to compare robot and human.

As far as the human clusters (Section 4.6.5) are concerned, we can find an appropriate trajectory for clusters 1, 2, 5, 6, 7 of open1; the proposed robot trajectories for clusters 3 and 4 do not cover enough areas of the map. For example, if we consider cluster 6 (Page 61), we get the trajectories in Figure 5.21. Compared to the trajectories in the cluster, two are shown in Figure 5.22, we can see that they are all different, but they are all equally efficient in exploring the environment. **Considering uffici1 we can find appropriate robot Results only for clusters 2 and 5**, the proposed robot trajectories for the others do not cover enough areas, **this depends** not only **on the exploration policy**, but also **on the speed of the robot**. As highlighted in Section 5.8.1, there is a noticeable difference in the *average speed* between humans and robots, this difference is proving to be a severe limitation that prevents the robots from covering the same amount map areas as humans. Possible solutions to this problem are: increasing the speed of the robot, increasing the amount of time a robot is given before the exploration ends, and changing the robot's movement so that it can proceed while it rotates. These solutions might no longer be needed if the exploration policy of the

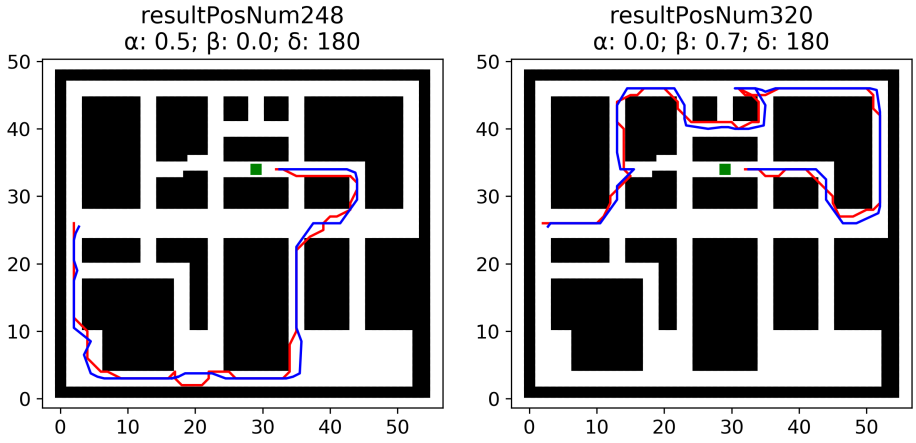

Figure 5.21: Robot trajectories for cluster 6, map open1.

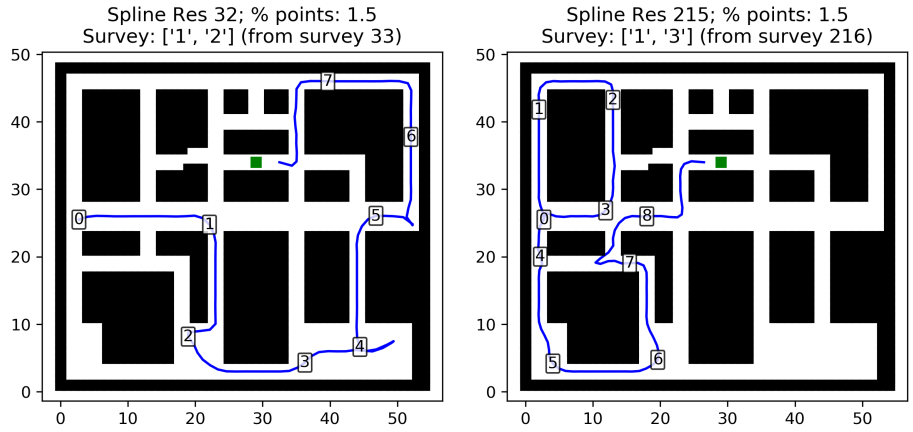

Figure 5.22: Human trajectories from cluster 6, map open1.

robot is changed.

5.10 Hidden Markov models

5.10.1 Hidden Markov model likelihood

In Section 5.9 we described a way of selecting one robot Result from all the collected ones, now **we want to know if the logLikelihood of the HMM can be used** for the same purpose, or even **to stop a robot exploration before it reaches its end**. The logLikelihood can only decrease as a trajectory gets longer, so we could stop a robot when its logLikelihood is below

the best value found so far. This can be done if the logLikelihood proves to be an effective metric to identify similar trajectories.

In order to test it, we compute one HMM for each human trajectory, we save them, then we load only the ones that take place in a specific map. We consider one robot Result, we compute its logLikelihoods w.r.t. the loaded HMMs and we order them from best, i.e., highest logLikelihood, to worst, i.e., lowest logLikelihood. Once this is done we check the HMMs. We find out that the HMMs learnt from the trajectories of the players who got lost and covered all map are the one providing the best logLikelihoods. This means that most of robot Results, both good and bad, are matched to the same HMMs learnt from bad human Results; **this is not what we want**. This happens because a bad Result explores the entire map, thus its HMM has states everywhere, and as such it is able to model a robot trajectory much better than an HMM that covers only a part of the map: as soon as the trajectory reaches an unexplored area, the HMM model cannot cope with it and the logLikelihood falls down.

5.10.2 Hidden Markov model samples

As mentioned at the end of Section 4.4.2, **hidden Markov models can be used to generate new trajectories**. The HMM original sampling function works, but we want more control over the final result, for this reason **we implement a custom sampling function**. We select the first state using the initial probability distribution, then we look at the transition probability matrix without self transitions to choose the next state. Each time a state is selected its position on the map is added to the final list to be returned. We do not want the positions of the observations, we want the positions of the states, which mean their x and y values identifying a precise location on the map. We stop adding state positions either when we reach a given number of samples or when we find all targets, we use the previously computed final states to check that. Furthermore, we implement an additional memory system; we are aware that this contrasts with the underlying assumptions of HMMs (Section 4.4.1), but it has proved to be an effective solution to solve our problem. If we sampled without this memory system, most of the samples would get stuck in a loop that prevents them from reaching all targets. We add a system such that each time we go from state i to state j we decrease the transition probability a_{ij} by a decrease percentage. The total amount that is removed is distributed to the remaining values in row i according to their weight, at the end of this operation $\sum_{j=1}^{N} a_{ij} = 1 \ \forall i$. In order for this system to work we must slowly restore the original values, we do this immediately after the next state j is selected. We use an increase

percentage to add/subtract what was previously removed/added. All these steps can be summed up this way:

1. Select an initial state using the initial probability distribution.
2. Use the transition probability matrix to select the next state.
3. Partially restore those values that were previously changed.
4. Decrease the content of cell $[i, j]$ in transition matrix and distribute it on the same row.
5. Go back to step 2 until all targets are found or the maximum number of samples is reached.

Once the list with all the state positions is ready we are halfway through our process. In order to have a complete and valid trajectory we must begin from the starting position, move through the space keeping obstacles into account, and ending close the last target if all targets are found. The starting point is extracted from the .txt file of the map layout (Section 2.2); each position in the list computed by our custom sampling function is converted to the closest Voronoi point (Section 3.2.3); if all targets are found then the closest target to the last position is selected. Once beginning, end, and all intermediate points are ready we use theta* to compute the shortest path between each pair.

This approach is able to compute new trajectories much **faster** than all the others, **but the generated trajectories follow a combinations of the paths learnt by the HMM**, so if a certain area was never explored then no states were defined in that area, and thus no trajectory can explore it. Furthermore, **we get a Result that contains only some of the data w.r.t. the Results obtained via Unity**. For example, we have no completion time, and even if in this thesis we do not use the saved orientation, it is another missing piece of data. Lastly, even if *has found all targets* is `True`, the generated trajectory may not get close to a target unless that target is the final one. A clear example is shown later in this section (Figure 5.24)

In Figure 5.23 we can see two trajectories generated from the same human Results in map open1; we place one trajectory per row in the figure. We highlight that since the HMMs are computed from scratch each time, the first HMM has 14 states, i.e., components, while the second one has 16. This happens because of slight variations in the learning process that change the BIC values, thus resulting in a different number of states, as explained in

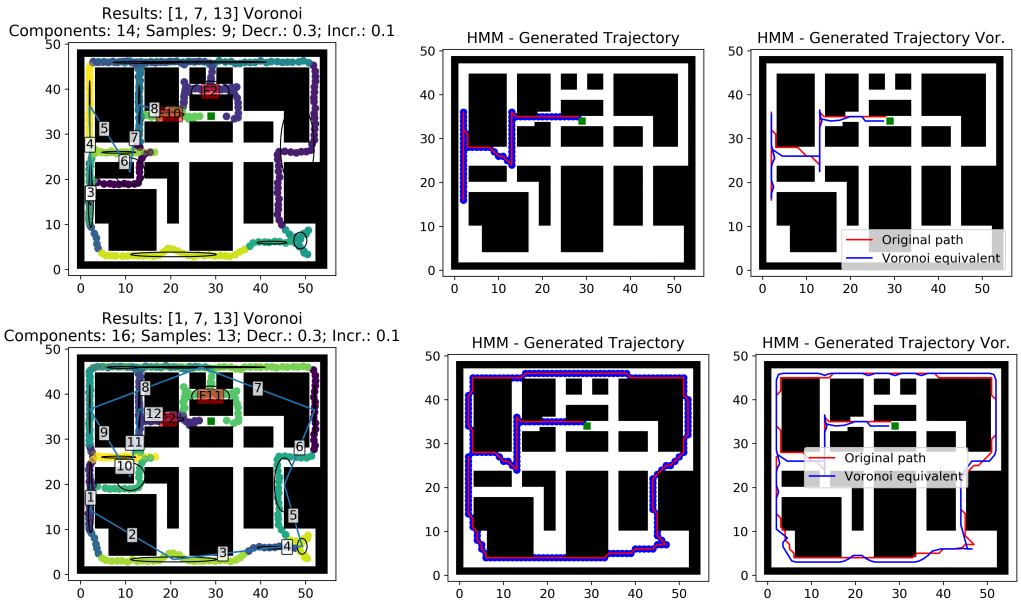

Figure 5.23: HMM generated trajectories based on human Results 1, 7, 13 - map open1.

Section 4.4.2. The central column shows the full trajectories after theta* has been used; the rightmost images show the theta* paths and their Voronoi equivalents.

In Figure 5.24 we report two trajectories for map uffici1. Again, we can see that the number of states is different between the two leftmost images, but we can also see that the decrease and increase percentages used by our custom sampling function are very different. In Figure 5.23 they were 0.3 and 0.1 respectively, they are the same in the first row of Figure 5.24, while in the second row they are 0.9 and 0.001. We were forced to use this extremes values due to the map layout. An increase percentage of 0.1 means that after 10 transitions the modified value is back to its original value, this increase was too fast for a map like uffici1 where dozens of transitions might be needed to get back to an already visited state. If we look at the first row we can see that we get 100 samples, but they where all in the same loop. To break that loop we must decrease the probability of continuously selecting the same transitions, and we must ensure that probability is not restored to its original value too fast. We can achieve this result using 0.9 and 0.001, as shown in the second row of Figure 5.24. In second row, central image, we can also see that the targets at the top and at the bottom are not touched by the trajectory, even if it has found all targets. If we look at the leftmost image

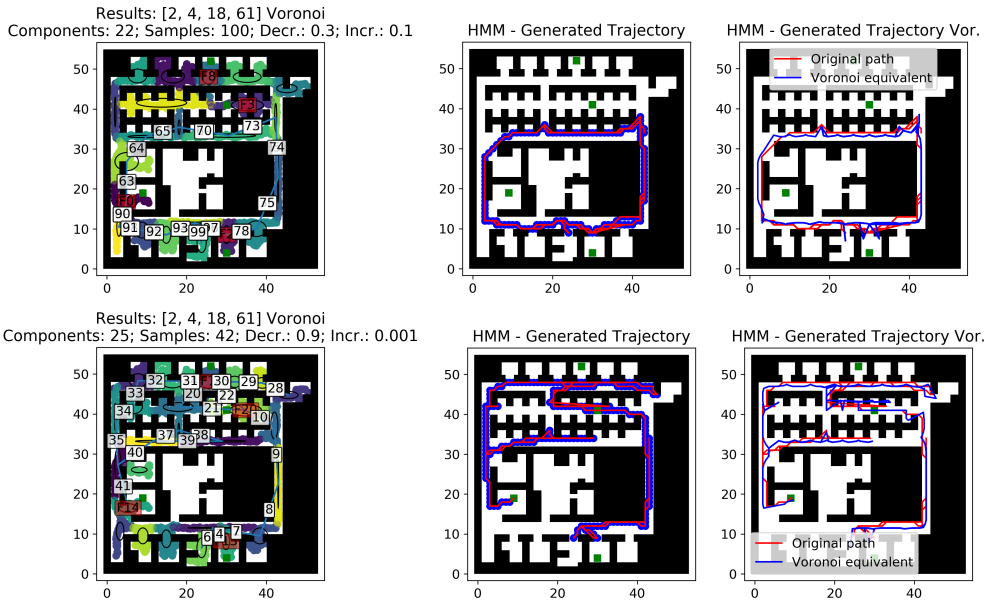

Figure 5.24: HMM generated trajectories based on human Results 2, 4, 18, 61 - map uffici1.

we can see that the trajectory reaches the centres of the final states (the small red boxes in Figure 5.24) instead of the actual targets. Lastly, we also highlight that we have to choose 4 trajectories for this HMM, where each trajectory has found all targets and ends in a different one, otherwise one or more final states could not have been computed, more on this in Section 4.4.2.

5.11 Trajectory reproduction conclusions

After all the considerations in this chapter we can conclude that we do not have a best trajectory reproduction technique that overcomes all the others. Each one has its pros and cons, and depending on the situation a different approach should be preferred.

If the situation allows the time for the computation of 484 Results, and the goal is to get a robot trajectory for a continuously changing cluster of human trajectories, then **grid search** is the best solution (Section 5.3). Each time the cluster changes we look for the best already obtained robot Result and we provide it as answer. However, since the 484 robot trajectories do not change, after a while it is better to perform a new grid search. In the meantime, the old 484 trajectories can be used.

If we do not have a lot of time, and the goal is to get a new trajectory for a cluster of human trajectories, then **robot distribution exploration** (Section 5.6) can compute a new trajectory for the given cluster faster than a grid search. Furthermore, each time it is used the trajectory is different even if the human cluster does not change.

If the goal is to focus on the areas of the map that players have already explored, then samples generated from a **hidden Markov model** provides the fastest way of getting new trajectories (Section 5.10.2); however, an automated system to find the best parameters, decrease and increase percentages, and filter the generated trajectories based on the features of a given human cluster, is not in place.

Grid search and robot distribution exploration are based on **the** existing **policy that drives the robot** exploration (Section 2.4.3). As we have reported in this chapter that policy **is not appropriate** for two reasons: it is not deterministic, and it is not good enough to properly explore the map uffici1.

Some of the causes of the non-determinism are deeply intertwined in how Unity works, we reported on this in Section 5.4.

Chapter 6

Conclusion

In this thesis we considered various approaches to trajectory clustering and reproduction. Differently from the most widespread cases, we did not focus only on geometric-based clustering, but rather on higher level features capable of describing trajectories based on several qualities of the exploration paths. After some tests we found a good solution using the features described in Chapter 4.

Based on these features we looked for effective methods to reproduce trajectories in a cluster. We found different alternatives, each one with its own pros and cons. We summed them up at the end of Chapter 5. In the same chapter we analysed the behaviour of the robot in order to understand the extent of its non-determinism, and we provided a list of possible causes.

In the end we achieved our goal of clustering trajectories in an effective manner, but we did not find a one-size-fits-all solution to the problem of reproducing the trajectories belonging to a cluster.

6.1 Known issues and possible criticism

The 9 features that are currently used in the clustering procedure may not be the best ones, in fact, we saw that in map uffici1 one binary feature splits the collected results in two groups.

Other issues are related to the hidden Markov models: Markov and output independence assumptions let us work with simpler models, but they do not represent how a human being thinks. Human beings' decisions are impacted by past actions, so we implemented a memory system in our custom sampling function for the hidden Markov models. This system requires two parameters to operate, the computation of those parameters is not automated, but they are manually chosen after some tests.

Robot distribution exploration provides many Results, however the final ordered list of Results contains bad trajectories before others that appear to be much better. This problem does not arise for the first dozens of Results, but it is there.

Lastly, the non-determinism of the robot behaviour proves problematic, it prevents us from associating a stable good or bad trajectory to a given parameter combination. Furthermore, the current policy does not allow the robot to effectively cover the more complex map.

6.2 Future developments

Future improvements could look for different features, or different weights for the existing ones, in order to improve the clustering capabilities. Collecting more data for the maps in the second group (Section 2.4.1) could be useful to apply clustering on those maps as well.

Another improvement could be related to the map layouts. As stated in Chapter 4 our features work only on flat maps. Extending the code that computes the features could potentially make them work in multi-floor maps.

At the moment hierarchical clustering cuts the dendrogram tree at a fixed height. Given the non uniform distribution of the collected Results over the various clusters, it could be useful to consider applying dynamic tree cut [23], in order to change the height of the cut based on some factors.

In the first paragraph after the list on Page 61, we highlighted that according to our results being a gamer or not has an impact over the final performance. Verifying whether this statement holds true in other conditions or with more samples could prove interesting.

The policy used by the robot during exploration could be changed or improved. Alternatively, the entire robot architecture could be changed, shifting away from the current one based on a real robot, in order to be able to explore the virtual environment much faster.

Appendix A

Acronyms & definitions

In this appendix we collect acronyms and terminology used throughout this thesis.

A.1 Acronyms

- **BFR:** Bradley-Fayyad-Reina.
- **BSS:** Between-clusters Sum of Squares.
- **BIC:**Bayesian Information Criterion.
- **DB:** DataBase.
- **DBSCAN:** Density-Based Spatial Clustering of Applications with Noise.
- **DM:** Decision Making.
- **DTW:** Dynamic Time Warping.
- **EU:** Euclidean.
- **FPS:** First Person Shooter or Frames Per Second.
- **HMM:** Hidden Markov Model.
- **IP:** Internet Protocol.
- **KE:** Knee/Elbow.
- **L&Z:** Simone Lazzaretti and Yuan Zhan. [2]
- **LCSS:** Longest Common SubSequence.

- **OS:** Operating System.
- **PCA:** Principal Component Analysis.
- **PF:** Piciarelli-Foresti. [5]
- **t-SNE:** t-distributed Stochastic Neighbour Embedding.
- **UI:** User Interface.
- **UX:** User Experience.
- **WebGL:** Web-based Graphics Library.
- **WSS:** Within-cluster Sum of Squares.

A.2 Definitions

The following definitions are given with the sole purpose of clarifying the terminology used in this thesis. For this reason the following definitions are not exhaustive and do not contain meanings beyond the needs of this work.

- **Ad-block:** software or browser extension that checks which content is loaded on a webpage and prevents all advertisements from being shown. On some occasions this type of program may interfere with other tasks, like a data upload process.
  `https://en.wikipedia.org/wiki/Ad_blocking`
- **Centroid:** n-dimensional point representing the center of a group of n-dimensional data points. In Euclidean spaces it is the average among all data points. [6]
- **Clustroid:** existing data point that is taken as cluster representative. It can be the data point that minimises the sum of all the distances to the other data points in the cluster. [6]
- **Control scheme:** the way a user is expected to interact with the game input device in order to control a character or other game features.
- **Dendrogram:** tree structure where each node can be either a termination leaf or the origin of two branches that terminate in two new nodes.
  `https://en.wikipedia.org/wiki/Dendrogram`

- **Firebase:** development platform that offers several tools to suit the needs of a wide variety of use cases.
  `https://firebase.google.com/`

- **First person shooter:** a game where the player's point of view is the same experienced by the character they are controlling. The player cannot see the entire character they are controlling, only the character's hands are shown on screen, usually holding one or more guns that the player can use to shoot different types of projectiles.
  `https://en.wikipedia.org/wiki/First-person_shooter`

- **Fog of war:** technique used in some video games to hide parts of the level that are far away from the player or from the player controlled characters. The goal is to prevent the player from seeing what is happening. Sometime the fog of war can let the player sees the orography of the terrain but not enemies.
  `https://en.wikipedia.org/wiki/Fog_of_war#In_video_games`

- **Frames per second:** number of distinct images, or frames, shown on a display in a time interval of one second.
  It can be computed as $\frac{1}{frame\,time}$.

- **Frame time:** amount of time an image, or frame, persists on screen before being replaced by another image.
  It can be computed as $\frac{1}{frames\,per\,second}$.

- **Game object:** fundamental object in Unity, it is a container where components like characters, lights, code,... are placed.
  `https://docs.unity3d.com/Manual/class-GameObject.html`

- **Greedy algorithm:** algorithm that selects the best local solution at each step of its execution; for this reason it might find a local optimum instead of the global one.

- **Grid search:** parameter optimisation technique that tests any possible, or specified, parameter combinations and selects the one providing the best result.

- **Heroku:** platform as a service, it can be used to run the back end for a data collection process.
  `https://www.heroku.com/`

- **Inspector:** see Unity inspector.

- **Objective function:** code in charge of computing the utility value to be returned each time an input is given.
- **Playtest:** meeting in which a group of players play a game and provide their feedback to the development team which organised the session.
- **Result:** when written with capital "R" it means the collected data for one or more agents (humans and/or robots). It is often used to denote a specific trajectory.
- **Scene:** see Unity scene.
- **Spawn:** as verb it means to appear or to be created and it is usually associated with a *spawn point* or a *spawn position*, which is the location where the object of interest is spawned.
  As noun it means the location where the spawning action takes place.
  `https://en.wikipedia.org/wiki/Spawning_(video_games)`
- **Spline:** piecewise defined function that is used to create a smooth line connecting multiple assigned points. Once the spline is computed it can be sampled to extract any number of points.
  `https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.splprep.html` & `https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.splev.html`
- **Stuttering:** condition that occurs when the time interval between two consecutive frames is not constant. To the human eye this translates into an inconsistent fluidity that detracts from the experience.
  `https://en.wikipedia.org/wiki/Micro_stuttering`
- **Theta*:** algorithm able to compute a near-optimal path between two given locations.
- **Unity:** game engine providing all the tools needed for development, test and release of a vast array of game genres.
  It can be download for free from `https://unity.com`.
- **Unity inspector:** window inside the Unity editor where many variables can be changed before or during the execution of a game.
  `https://docs.unity3d.com/Manual/UsingTheInspector.html`
- **Unity scene:** container of environments, menus,... Usually a scene contains one game level, but it is not mandatory.
  `https://docs.unity3d.com/Manual/CreatingScenes.html`

- **Unity unit:** unit of distance within Unity.
- **WebGL:** web-based technology used to deploy high visual quality games in a compatible web browser.
  `https://www.khronos.org/webgl/`

Appendix B

Unity documentation

As stated in Chapter 2 and Section 3.2.2, we expanded a pre-existing Unity project called *Project Arena*. In this appendix we document the additions implemented during this thesis, more information about the robot implementation, or other elements that were not changed, can be found in Appendix A in L&Z's thesis [2].

The *Asset* folder of the project contains several subfolders, the most important ones are *Scenes*, *Scripts*, and *Maps*.

B.1 Scenes

Scenes contains all the scenes used by both this thesis and the previous ones. It contains a folder called *OLD* with the scenes that are not used in our work; a folder *Robot Experimenting* with the scenes used in our experiments, and the scenes *Start*, *Menu*, and *Error*. These 3 scenes are used, respectively, to start the human experiment, to show the main menu (i.e., title screen), and to show an error screen if an error occurs. Typically, an error occurs when the connection between client and server cannot be established.

The content of *Robot Experimenting* is split between human-related scenes, robot-related scenes, and other test scenes not used in our experiments, for this reason they are not listed in this documentation.

B.1.1 Human-related scenes

In addition to *Start*, *Menu*, and *Error*, the human experiment is composed of the following scenes located inside the *Robot Experimenting* folder:

- *Experimenting - Control 1*: it provides instructions to the players (e.g., Figure 3.1 Page 19)

- *Experimenting - Tutorial*: it is the tutorial map mentioned at the end of Section 2.4.1.
- *Experimenting - Test1*: it is the single target map scene, whether open1 or uffici2 depends on which map group the player chose (Section 2.4.1).
- *Experimenting - Test2*: it is the multi target map scene, whether uffici1 or open2 depends on which map group the player chose (Section 2.4.1).
- *Experimenting - Survey*: it is the scene where the player answers the survey (Section 2.4.1).

These scenes are the ones selected at build time in order to create the appropriate game build. Note that even if in the Unity editor it is possible to launch any of the previous scenes, in order to correctly load them either *Start* or *Experimenting - Control 1* should be the starting scene.

B.1.2 Robot single target & multi target scenes

Robot experiments take place inside the Unity editor, so we do not compile a game build to run those experiments. Inside the *Robot Experimenting* folder we have two robot-related scenes: *Experimenting - Robot Single Target* and *Experimenting - Robot Multi Target*, the former is for single target maps, the latter is for multi target maps. In order to select which map to load we must provide its .txt file in the property *Text File Path* of *SL Map Manager* component in *SL Map Manager* game object under *SL Divise Map* game object.

In both scenes the two most important game objects are *ExplorationIterator* and *Robot*. They contain several scripts that will be explained in Section B.2:

- *ExplorationIterator* game object:
  - *ExplorationIterator* script.
  - *InputReader* script.
- *Robot* game object:
  - *Robot* script.
  - *RobotMovement* script.
  - *RobotProgress* script.
  - *RobotPlanning Theta Star* script.
  - *RobotDMUtilityCloseWall* script.

B.2 Scripts

Scripts contains the folders with the C# code; some of the scripts are no longer used, but they are still in these folders. In particular, we focus on on the following scripts (each script is preceded by the folder containing it):

- *Connectivity/IPManager*: it is in charge of getting the IP address of the client running the game. It relies on `https://api.ipify.org`, if it does not work it tries `http://icanhazip.com`. If none of these works, then we assume there is no internet connection, thus the *Error* scene is loaded.

- *Connectivity/RobotConnection*: it is in charge of uploading the human Result to Firebase Realtime Database, in order to do so, its property *DB Url* must be set to *https://projectName-randomValues.firebaseio .com/Results/*, more on this in Section C.1. It works using a free third party HTTP and REST client downloaded from `https://assetstore. unity.com/packages/tools/network/102501`.

- *Connectivity/SurveyUploader*: it is in charge of uploading the survey answers to Firebase Realtime Database, in order to do so, its property *DB Url* must be set to *https://projectName-randomValues.firebaseio .com/Surveys/*, more on this in Section C.1. It works using a free third party HTTP and REST client downloaded from `https://assetstore. unity.com/packages/tools/network/102501`.

- *Entities/Player*: it is the main script in charge of the player interactions. Some of the properties are inherited from a previous work (Section 2.2), we focus only on those there are relevant for our work. *Seconds To Wait* specifies the amount of time, in seconds, before the player exploration is automatically terminated, we set it to 480 as said in Section 3.2.2. In Section 3.2.2 we described the distance-based sampling, it is implemented in this script and it requires two parameters: *Distance Type*, which specifies if the distance should be considered as the direct air distance from the latest saved position (*Air Distance*), or as the actual walked distance from the latest saved position (*Walked Distance*); *Distance Interval Between Saves*, which is the distance threshold from latest save point above which the current position and rotation of the player are saved, it was called n in Section 3.2.2. We used *Distance Type = Walked Distance* and *Distance Interval Between Saves = 3*, but since *3* has proved to be too high (Section 4.3.3) in future works it might be changed.

- *Entities/Robot*: it is the main robot script, it coordinates all the other robot-related scripts. Some of its properties are not used in our work, but others are: *Numb Ray* is the number of rays shot by the robot's proximity sensors (Section 2.4.3); *Time For Scan* is the time interval between two perception processes (Section 2.4.3), *Time For Decision* is the time interval between two decision processes (Section 2.4.3), these two values are used by the timers mentioned in Section 5.4. As far as *Distance Type* and *Distance Interval Between Saves* are concerned, they are the same described in *Entities/Player*.

- *Entities/RobotMovement*: it is in charge of moving the robot. The move forward or rotate behaviour described in Section 3.2.2 is implemented here. The values of *speed* and *rotationSpeed* can be set only in code, they are not exposed in the inspector.

- *Entities/RobotProgress*: it is in charge of saving the progress as the agent explores. If the agent is the robot, data are saved locally in a JSON file, then they are converted to a .txt file once the exploration of the corresponding robot is over. If the agent is the player, once the exploration is over, data are uploaded to the Firebase Realtime Database by *RobotConnection*.

- *Libraries/InputReader*: it checks the folder *Assets/Inputs* looking for files containing the parameters to be used by *ExplorationIterator*; the names of the files must be the numbers of the iterations, starting from 1. It works under the assumption that each file contains only three numbers, two floats and one int, that represent the values of α, β, and δ index, which is the index use to choose the forgetting factor in the corresponding list in *ExplorationIterator*, δ index is not the actual value of the forgetting factor δ.

- *RobotDecisionMaking/RobotDMUtilityCloseWall*: it implements the policy described in Section 2.4.3.

- *RobotPlannings/RobotPlanningThetaStar*: it implements the theta* algorithm used by the robot to reach its destination (Section 2.4.3).

- *Singletons/ExplorationIterator*: it is in charge of handling the robot exploration. If its property *Parameters Selection* is set to *Grid Search*, then it performs the grid search using the values specified in the properties *Min Alpha*, *Min Beta*, *Max Alpha*, *Max Beta*, *Alpha Increment* and *Beta Increment*. The possible values for the forgetting factor δ can be found in the C# code. Otherwise, if its property *Parameters Selection*

is set to *Read From File*, it asks *InputReader* for the values of α, β, and δ. If no values is given, then it waits *secondsBetweenChecksForNewValues* before asking again. The remaining properties are: *Number Of Robots*, which specifies the maximum number of robots that can be deployed simultaneously, its value cannot exceed the total amount of combinations in the grid search or the amount of values *InputReader* can provide when the simulation starts; *Exp Time Scale*, which specifies the time scale of the exploration, we set it to 1 as said in Section 3.2.2; *Seconds To Wait*, which specifies the amount of time, in seconds, before a robot exploration is automatically terminated, we set it to 480 as said in Section 3.2.2.

B.3 Maps

Maps contains the .txt files with the map layouts (Section 2.2). The subfolder *City Style Map* contains the files *open1.map.txt* and *open2.map.txt*, while the subfolder *Star Style Map* contains the files *uffici1.map.txt* and *uffici2.map.txt*. Note that the file extension .txt is not shown in some of Unity UI. These .txt files are the ones assigned to *Text File Path* of *SL Map Manager* component in *SL Map Manager*, as stated in Section B.1.2.

Appendix C

Firebase documentation

Firebase is a Google-owned platform that offers several tools to suit the needs of a wide variety of use cases. The official Firebase website can be reached at `https://firebase.google.com`.

After creating a Google account, we can access the Firebase console. In the console it is possible to create a new project, once the project is created all Firebase tools become available. For our purposes, we focus on two tools: Realtime Database and Hosting.

C.1 Firebase realtime database

Realtime Database can be found in the Database menu. Once a new Realtime Database is created, an associated url is created too; the format should be similar to *https://projectName-randomValues.firebaseio.com*. This url is the one that must be used in *RobotConnection* and *SurveyUploader* in Unity in order to make the upload process work, more on this in Section B.2.

New information can be added using PUT requests to the database url followed by the subdomain where the information should be placed. The information is sent as JSON, a commonly used file format when exchanging data between a client and a server. In our case the database has 2 main branches: Results and Surveys. The former contains the data related to the explorations, the latter the answers to the surveys. Each Result has a unique identifier: *YYYY-MM-DD--HH:MM:SS:mmm--6RandomDigits*. If two Results with the same identifier are uploaded, then the older one is overwritten by the newer one. We created the identifier in order to minimise this possibility, furthermore, we do no have traffic spikes that could result in several Results being uploaded at the same time. The date in the identifier is the precise time, up to the milliseconds, at which the exploration ended.

Survey follows the same principles, but its identifier is the same used by the Result of the latest played map. Instead of PUT, POST requests can be used to avoid overwriting, but by doing so, the identifiers would be completely randomised.

A Realtime Database has an associated set of rules that can be used to regulate the access to the database. The following code represents the rules in place during our collection process:

```
"rules": {
    ".read": false,
    ".write": false,
    "Results": {
        ".read": false,
        ".write": true
    },
    "Surveys": {
        ".read": false,
        ".write": true
    }
}
```

These rules prevent any read from the database, and allow new information to be written only in the subdomains *Results* and *Surveys* of our database-associated url. This system is not secure, any input on those two subdomains is accepted without any check. Since our data collection process was kept relatively private, we decided to focus our time on other aspects, assuming no player would try to attack us. The official Firebase Realtime Database documentation provides more information about how to implement stricter security measures, it is available at `https://firebase.google.com/docs/database/security`. Once the data collection process is over we change the rules, replacing the `true` with `false`.

In order to download the collected Results and Surveys we go to the Firebase console, then Database, then we click on the icon with three dots in the upper right corner of the box, then select "Export JSON". This can be done at any time, in fact, we periodically downloaded them before the collection process was over, in order to evaluate the Results collected so far.

C.2 Firebase hosting

We created a Hosting instance for our project, then we used Unity to generate the files needed in order to run a WebGL version of our game. We put those files inside the *public* folder created using the Firebase CLI (Com-

mand Line Interface). Updated instructions about how to install the CLI, connect it to Firebase Hosting, and deploy the game, can be found in the official documentation available at `https://firebase.google.com/docs/hosting`. Inside the Hosting web interface we can find the website of that specific Hosting instance, it is something similar to *https://projectName-randomValues.web.app*. This is the url that must be provided to the testers. In case of need, it is possible to restore older versions of the files in the Hosting *public* folder via Firebase Hosting web interface. Once the data collection process is over, we use the CLI to deploy an updated *public* folder containing only the file *404.html*, which is used to show a 404 error page to anyone who opens the aforementioned link. This file is automatically generated by Firebase when the project is created. In case of need, it can be modified, for example to change the message shown when the 404 error page is reached. When the CLI asks if we want to use a Firebase Database we must answer affirmatively, this way a file *database.rules.json* containing the same rules specified for the Realtime Database is created.

Appendix D

Python documentation

We have a dozen Python files, so we explain each one in its own section. Each file contains a first part with the functions that are expected to be used directly, these functions start with "My" so that they can be easily distinguished; in the remaining part of the file the actual implementation and some utility functions can be found. In this appendix we comment only on the first part; in any case, inside the files, almost all functions are preceded by a comment explaining their purpose.

Italic is used to identify parameters or functions when they are not written in **bold**.

D.1 Metrics_updated.py

This file is an updated version of the file written by L&Z. It contains the following functions:

- **MyComputeVoronoiDistMatrixForPathsInSelectedMap()**: it computes the Voronoi distance matrix for all the paths in *mapToAnalyze*, which is the parameter where we specify which of the four maps should be taken into consideration.
- **MyAmountOfResultsForEachMap()**: it provides the amount of available Results for each map.

D.2 FirebaseAdapter.py

This file only purpose is to convert the JSON file downloaded from the Firebase Realtime Database into several .txt files, one for each Result and survey. It contains the following function:

- **MyFirebaseToTxtConverter()**: it performs the conversion from JSON to .txt assuming *pathOfFileDownloadedFromFirebase* specifies the path to the JSON file, and *resultsDirectoryPathOriginals* specifies where the output .txt files should be placed.

D.3 Voronoi.py

This file contains the code to compute the Voronoi points and the Voronoi equivalent paths. It contains the following functions:

- **MyVoronoiFilteredPoints()**: it provides the Voronoi points for a map after removing those that are inside walls. *mapName* is the map under consideration.

- **MyVorEquivalentAllPaths()**: it computes the Voronoi equivalent paths for all the paths in the folder specified by *resultsDirectoryPathOriginals*, i.e., where the .txt files generated by *MyFirebaseToTxtConverter()* are located.

- **MyVorEquivalentIntervalPaths()**: it computes the Voronoi equivalent paths for all the paths whose Result numbers are greater than the *resultNumber* (*resultNumber = 0* means all Results).

- **MyVorEquivalentMultiplePaths()**: it computes the Voronoi equivalent paths for all paths in the specified list *files*.

- **MyVorEquivalentSinglePath()**: it computes the Voronoi equivalent path only for the path in *file*. The legend position in the final figure can be shown by enabling *showLegend*, and its position can be changed using *lengendPosition*.

- **MyRobotVorEquivalentIntervalPaths()**: it computes the Voronoi equivalent paths for all the robot paths whose Result numbers are greater than the given *resultNumber* (*resultNumber = 0* means all Results). No legend is added to the figure. The Result files are assumed to be in the folder specified before this function declaration in Voronoi.py.

D.4 MetricsClustering.py

This file contains the logic to perform the metric-based clustering (Section 4.3). It contains the following functions:

- **MyMetricClustering()**: it performs the clustering; it contains several functions calls, one for each metric (Section 4.3.1). Each call performs the clustering using all 7 methods (Section 4.2).
- **MyKneeElbowAnalyzer()**: it performs the Knee/Elbow analysis on *clusters*.

D.5 ClustersDrawer.py

This file contains the code to draw several trajectories, each one with its own colour, in one unique figure. It did not prove useful to effectively visualise the clusters, so it was discarded. It contains the following function:

- **MyDraw()**: it automatically draws the paths in the given *clusterToDraw*.

D.6 CriticalPoints.py

This file is used to compute the critical Voronoi points, the critical Voronoi groups, and the critical Voronoi equivalent paths. It contains the following functions:

- **MyGetVoronoiCriticalPointsEquivalent()**: it computes the critical Voronoi equivalent paths for each human Result whose number is greater or equal to *startingValue*. *shouldShowLegend* specifies if the legend should be add to the final figure, *lengendPosition* specifies where the legend should be located.
- **MyGetGroupedCriticalPoints()**: it computes the critical points and groups for the map specified in *mapUnderConsideration*. *distance* is the distance within which two critical points are considered part of the same group. *shouldGetADifferentFigureForEachCriticalGroup* can be set to `True` in order to generate a separate figure for each group.

D.7 HMM.py

This is the file containing the code to perform the HMM computations described in Sections 4.4.2 and 5.10.2. It contains the following functions:

- **MyGetSplineFiguresWithSurvey()**: it generates the figures of the splines of the requested *inputFileNumbers*, with the answers to the

survey added in the title. If no survey is found for a Result number, then that Result is ignored. *splineColor* specifies the colour of the spline, *divValue* specifies the interval of points after which a label is placed on the spline, *inputFileType* specifies to which type of Results *inputFileNumbers* refers to, e.g., *Original*, *Voronoi*, *VorCriticalPoints*, *RobotVoronoi*.

- **MyGetHMMForResults()**: it computes the HMM for the trajectories specified in *inputFileNumbers*. *inputFileType* specifies to which type of Results *inputFileNumbers* refers to.

- **MySaveHMMForSingleResult()**: it saves an HMM for each of the requested Result in *inputFileNumbers*. *inputFileType* specifies to which type of Results *inputFileNumbers* refers to.

- **MyGetSampleTrajectoryFromHMM()**: after learning an HMM using the trajectories specified in *inputFileNumbers*, it uses that HMM to get samples. *inputFileType* specifies to which type of Results *inputFileNumbers* refers to.

- **MyGetLogLikelihoodForRobotResult()**: it computes the logLikelihood of the specified *robotResultsToBeConsidered* w.r.t. all the HMM available for map *mapUnderConsideration*.

Other values, like the *decreasePercentage* and *increasePercentage* mentioned in Section 5.10.2, can be changed in *ActualHMMCode([...])*.

D.8 HMMClustering.py

This file was supposed to be used to perform the clustering based on the HMM logLikelihoods, as described in Sections 4.4.3 and 5.10.1. It contains the following functions:

- **MyScoreResultUsingSavedHMM()**: it computes the logLikelihood for the specified *resultToBeConsidered* w.r.t. each HMM model whose number is in *HMMFileNumbers*. *inputFileType* specifies to which type of Results *resultToBeConsidered* refers to.

- **MyGetHMMBasedClusters()**: it computes the result of the hierarchical clustering of the given list of Results *resultsList*. *inputFileType* specifies to which type of Results *resultsList* refers to.

D.9 SurveyGraphs.py

This file contains the code to draw the graphs for the various survey questions (Section 2.4.1), which means it contains the code to compute the measures of a group of Results (Section 4.5.2). Its functions are the following:

- **MyDrawGraphs()**: it contains all the necessary function calls to draw all the graphs of the survey. It is possible to use it to ease the graph drawing.
- **MyResultFilter()**: it filters, e.g., all Result numbers from map open1, based on the groups of Results obtained via *MyDrawGraphs()*; these groups must be put in *listOfList*.
- **MyResultSorter()**: it returns one sorted list containing all the elements from the four separated input lists.
- **MyGetRobotResultMeasures()**: it computes the measure file of robots Results for the map specified in *mapName*.
- **MyGetMixedResultMeasures()**: it computes the measure file of the robots in *indicesOfRobotNumbersOfInterest* and the humans in map *mapName*.

D.10 MeasuresClustering.py

This file contains the code used in Sections 4.6, 5.3, 5.5, and 5.9. It contains the following functions:

- **MyMeasureClustering()**: it performs the measure-based clustering and draws the corresponding PCA and t-SNE graphs. *robotResultsToBeConsideredInMixedCase* is the list, often made up of only 1 element, of robot Results to be clustered with human Results if *measureSource* is set to *MeasureSource.Mixed*, otherwise it is ignored; *mapToBeConsidered* specifies the map under analysis; *alternativeToBeUsed* specifies which set of features to consider among the ones defined in Section 4.6.2; *measureSource* specifies which is the source of the data among *Human*, *Robot*, and *Mixed*, i.e., humans and robots; *roundNumber* specifies which round of grid search should be considered; *useRobotParametersAsLabel* can be enabled to replace the numbers of the robot Results with the corresponding values of the parameters α, β, δ (it works only when *measureSource* is set to *MeasureSource.Robot*). With *measureSource* set to *MeasureSource.Robot*, the code performs the measure

clustering of all robot Results without any human Result. If necessary *MyGetRobotResultMeasures()* in *SurveyGraphs.py* can be used to generate the robot measure files.

- **MyMeasureAllSingleRobotClustering()**: it performs the clustering of all robot Results, one at a time, with all human Results. It is the same as calling *MyMeasureClustering()* with *measureSource* set to *MeasureSource.Mixed*, while assigning one robot Result number at a time to *robotResultsToBeConsideredInMixedCase*. *mapToBeConsidered* specifies the map under analysis; *alternativeToBeUsed* specifies which set of features to consider among the ones defined in Section 4.6.2; *roundNumber* specifies which round of grid search should be considered.

- **MyBestRobotTrajectoryForCluster()**: it checks all grid search robot Results, and returns the ones that are the closest to *humanTraj*, which is the list of human Results under consideration. It implements the procedure used in Section 5.9. *mapToBeConsidered* specifies the map under analysis; *alternativeToBeUsed* specifies which set of features to consider among the ones defined in Section 4.6.2; *roundNumber* specifies which round of grid search should be considered.

- **MyPCAHumanClusters()**: it performs PCA with humans, and it colours each human cluster with a different colour (Figures 4.32 and 4.33 Page 59). The Results contained in each cluster are automatically gathered based on the specified information. *numberOfClusters* is the cardinality of the cluster; *hierachicalClusteringMethod* is the method (Section 4.2) whose clusters are the ones to be considered; *shouldDrawSingleHumanHighlightFigures* can be enabled so that, in addition to the overall PCA, the code will drawn one PCA for each human Result while highlighting its Result number. *mapToBeConsidered* specifies the map under analysis; *alternativeToBeUsed* specifies which set of features to consider among the ones defined in Section 4.6.2.

- **MyPCAHumansAndRobots()**: it performs PCA with humans and then adds robots to the figures. *mapToBeConsidered* specifies the map under analysis; *alternativeToBeUsed* specifies which set of features to consider among the ones defined in Section 4.6.2; *roundNumber* specifies which round of grid search should be considered.

- **MyPCARandomValues()**: it performs PCA with humans and then adds random feature values to the figures. *mapToBeConsidered* specifies the map under analysis; *alternativeToBeUsed* specifies which set of

features to consider among the ones defined in Section 4.6.2; *amountOfRandomSamplesToBeConsidered* specifies how many samples should be generated, each sample has as many random values as the features in *alternativeToBeUsed*.

- **MyDrawHistogramsForHumanClusters()**: it draws histograms for each cluster, showing the answers to the survey for all the humans in each cluster (this code assumes all humans Results in the clusters have a corresponding survey). *numberOfClusters* is the cardinality of the cluster; *hierachicalClusteringMethod* is the method (Section 4.2) whose clusters are the ones to be considered; *mapToBeConsidered* specifies the map under analysis; *alternativeToBeUsed* specifies which set of features to consider among the ones defined in Section 4.6.2.

- **MyGetHasFoundAllTargetsForHumans()**: it returns a list of `True` and `False` for each human cluster, indicating whether each human has found all targets or not. *numberOfClusters* is the cardinality of the cluster; *hierachicalClusteringMethod* is the method (Section 4.2) whose clusters are the ones to be considered; *mapToBeConsidered* specifies the map under analysis; *alternativeToBeUsed* specifies which set of features to consider among the ones defined in Section 4.6.2.

- **MyGetHasFoundAllTargetsForRobots()**: it returns a list of `True` and `False` for each robot in *listOfRobots* in map *mapToBeConsidered* in grid search round *roundNumber*. The Boolean value indicates whether the robot has found all targets or not.

For all previous functions in this file, more settings can be changed in the "# Settings" section of the file; e.g., *repetitionWindowSize*, which specifies the window size n used by the measure *number of repeated positions within repetition window* n (Section 4.5.2); or *percentageToBeUsedWhenReplacingUnavailableValue*, which is used when computing *average distance between repeated positions normalised*, it is the *1.25* used in Section 4.6.1.

D.11 TrainingAlgorithms.py

This file contains the code of the procedures described and/or used in Sections 5.2, 5.6, and 5.7. It contains the following functions:

- **MyAnnealing()**: it performs the parameter values exploration based on annealing algorithm (Section 5.2). *humanResultsToBeConsidered* specifies the Results the robot should imitate; *alternativeToBeUsed*

specifies which set of features to consider among the ones defined in Section 4.6.2; *maxObjFuncCalls* is the maximum number of objective function calls mentioned in Section 5.2.

- **MyRobotDistributionExploration()**: it implements the robot distribution exploration described in Section 5.6. *humanResultsToBeConsidered* are the Results the robot should imitate; *maxNumberOfValues* is the amount of combinations generated by the uniform sampling process, and it is also the maximum number of robots allowed in *Exploration Iterator*, more on this in Appendix B; *amountOfValueToBeKept* specifies how many of the best combinations must be kept for the next iteration; *amountOfNewValuesForEachCombination* specifies how many new combinations of α, β, and δ must be generated for each best combination; *amountOfGaussianIterations* specifies how many iterations using Gaussian distribution samples should take place; *alternativeToBeUsed* specifies which set of features to consider among the ones defined in Section 4.6.2. There is no need to specify a *mapUnderConsideration* since it is inferred from *humanResultsToBeConsidered*, assuming all values in *humanResultsToBeConsidered* come from the same map as the first one.

- **MyPCAFromOutcomeFileOfRobotDistributionExploration()**: it performs PCA using as input the file *OutcomeOfRobotDistributionExploration.txt* generated by *RobotDistributionExploration()*. This file is expected to be in the current working folder, i.e., the folder containing this code. *alternativeToBeUsed* specifies which set of features to consider among the ones defined in Section 4.6.2.

- **MyRobotVarianceAnalysis()**: it performs *explorationToBeDoneForEachParameterCombination* amount of explorations for each robot parameter combination identified by the robot numbers in *robotNumbersWhoseParameterCombinationsShouldBeConsidered*. Then, it collects the data and draws the figures showing the variance of each measure. These figures are the same that can be obtained using *MyRobotVarianceAnalysisGraphsDrawer()* and *MyPCAFromOutcomeFileOfRobotVarianceAnalysis()*. *alternativeToBeUsed* specifies which set of features to consider among the ones defined in Section 4.6.2; *roundNumber* specifies which round of grid search should be considered; *mapUnderConsideration* specifies which map should be considered.

- **MyRobotVarianceAnalysisGraphsDrawer()**: it draws box plot and PCA figures for the Results of *MyRobotVarianceAnalysis()*. Note:

files *Measures-robot_on_demandN.txt* must be the same generates by *MyRobotVarianceAnalysis()*, and *robotNumbersWhoseParameterCombinationsShouldBeConsidered* must be the same used by *MyRobotVarianceAnalysis()* to generate the *Measures-robot_on_demandN.txt* files. *alternativeToBeUsed* specifies which set of features to consider among the ones defined in Section 4.6.2; *roundNumber* specifies which round of grid search should be considered. There is no need to specify *mapUnderConsideration* since it is inferred from *mapNames* in *Measures-robot_on_demand1.txt*.

- **MyPCAFromOutcomeFileOfRobotVarianceAnalysis()**: it draws PCA figures for the Results of *MyRobotVarianceAnalysis()* or *MyRobotVarianceAnalysisGraphsDrawer()*, depending on which one was executed last. *alternativeToBeUsed* specifies which set of features to consider among the ones defined in Section 4.6.2; *roundNumber* specifies which round of grid search should be considered. There is no need to specify a *mapUnderConsideration* since it is inferred from *humanResultsToBeConsidered* in *OutcomeOfRobotVarianceAnalysis.txt*, assuming all values in *humanResultsToBeConsidered* come from the same map as the first one.

Note about how to use the code to transfer data between Python and Unity (from the beginning of *TrainingAlgorithms.py* file): "This code provides Unity with the parameters for each iteration by placing files in *parametersFilesDestination*, then it uses the trajectories generated by Unity, and stored in *unityResultsFilesFolder*, to evaluate the quality of each parameter combination. This code must start first, while in Unity the *Parameters Selection* in the inspector of *Exploration Iterator* must be set to *Read From File*. The maximum number of robots allowed in Unity is 1 if annealing is used; it is *maxNumberOfValues* otherwise." More information can be found in Appendix B.

D.12 FeaturesAnalysis.py

This file contains the code to draw all the graphs described in Section 5.8. It contains the following functions:

- **MyDrawAllGraphs()**: it contains the function calls to draw all variations of all graphs that the following functions can draw.
- **MyFeaturesNormalisedRobotsOnlyGraph()**: it draws box plots showing the distribution of all possible feature values after the nor-

malisation (Section 5.8.1). It works only for grid search robots or for both grid search robots and humans in map *mapUnderConsideration*. *shouldAddHumans* specifies if the human Results should be considered; *roundNumber* specifies which round of grid search should be considered.

- **MySingleParameterGraphs()**: it draws 3 box plots for each feature, showing the distribution of the values of that feature for each possible value of robot parameters α, β, δ (Section 5.8.2). *mapUnderConsideration* specifies which map should be considered, *roundNumber* specifies which round of grid search should be considered.

- **MyCubeHelixGraphs()**: it draws one figure for each feature; each figure contains 4 graphs showing the distribution of the values of that feature over the robot parameter space (Section 5.8.3).

Bibliography

[1] Jiang Bian, Dayong Tian, Yuanyan Tang, and Dacheng Tao, "A survey on trajectory clustering analysis", *arXiv e-prints*, p. arXiv:1802.06971, Feb 2018.

[2] Simone Lazzaretti and Yuan Zhan, "Simulating human behaviour in environment exploration in video games", Master's thesis, Scuola di Ingegneria Industriale e dell'Informazione, Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Academic Year 2017/2018.

[3] Tharindu Fernando, Simon Denman, Sridha Sridharan, and Clinton Fookes, "*Soft + Hardwired* attention: An LSTM framework for human trajectory prediction and abnormal event detection", *Neural Networks*, vol. 108, pp. 466–478, 2018.

[4] Brendan Morris and Mohan Trivedi, "Learning trajectory patterns by clustering: Experimental studies and comparative evaluation", *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 312–319, 2009.

[5] Claudio Piciarelli and Gian Luca Foresti, "On-line trajectory clustering for anomalous events detection", *Pattern Recognition Letters*, vol. 27, no. 15, pp. 1835–1842, 2006.

[6] Mohammed J. Zaki and Wagner Meira Jr., *Data Mining and Analysis: Fundamental Concepts and Algorithms*, Cambridge University Press, 2014.

[7] *SciPy*, `https://scipy.org`.

[8] *SciPy cluster hierarchy linkage*, `https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.linkage.html`.

[9] Dan Jurafsky and James H. Martin, *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition*, Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, 2009, Appendix A available online at `https://web.stanford.edu/~jurafsky/slp3/A.pdf`.

[10] Brendan Tran Morris and Mohan Manubhai Trivedi, "Trajectory learning for activity understanding: Unsupervised, multilevel, and long-term adaptive approach", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, pp. 2287–2301, 2011.

[11] Tomáš Vintr, Zhi Yan, Tom Duckett, and Tomáš Krajník, "Spatio-temporal representation for long-term anticipation of human presence in service robotics", *International Conference on Robotics and Automation (ICRA)*, pp. 2620–2626, 2019.

[12] Tomáš Krajník, Tomáš Vintr, Sergi Molina, Jaime Pulido Fentanes, Grzegorz Cielniak, Oscar Martinez Mozos, George Broughton, and Tom Duckett, "Warped hypertime representations for long-term autonomy of mobile robots", *IEEE Robotics and Automation Letters*, vol. 4, pp. 3310–3317, 2019.

[13] Mikel Vuka, Erik Schaffernicht, Michael Schmuker, Victor Hernandez Bennetts, Francesco Amigoni, and Achim J. Lilienthal, "Exploration and localization of a gas source with mox gas sensors on a mobile robot — a gaussian regression bout amplitude approach", *ISOCS/IEEE International Symposium on Olfaction and Electronic Nose (ISOEN)*, pp. 1–3, 2017.

[14] *hmmlearn*, `https://hmmlearn.readthedocs.io`.

[15] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*, Springer Series in Statistics. Springer, 2009.

[16] *Matplotlib pyplot boxplot*, `https://matplotlib.org/3.2.0/api/_as_gen/matplotlib.pyplot.boxplot.html`.

[17] *SciPy stats norm*, `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html`.

[18] Laurens van der Maaten and Geoffrey Hinton, "Visualizing data using t-SNE", *Journal of machine learning research*, vol. 9, pp. 2579–2605, Nov 2008.

[19] *Scikit-learn PCA*, `https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html`.

[20] *Scikit-learn t-SNE*, `https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html`.

[21] *SciPy optimize dual annealing*, `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.dual_annealing.html`.

[22] *seaborn cube helix*, `https://seaborn.pydata.org/examples/cubehelix_palette.html`.

[23] Peter Langfelder, Bin Zhang, and Steve Horvath, "Defining clusters from a hierarchical cluster tree: the dynamic tree cut package for R", *Bioinform.*, vol. 24, no. 5, pp. 719–720, 2008.