Autonomous Software Agents Project Report

Corte Pause Manuela - 240183
manuela.cortepause@studenti.unitn.it

Gentile Francesco - 240186
francesco.gentile@studenti.unitn.it

1 Introduction

In artificial intelligence, an agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators (Russell and Norvig 2010). However, such a broad definition is not particularly useful. Much more interesting is the idea of an agent (or system of agents) that operates with the objective to maximize its expected performance measure given its available knowledge and resources. Still, intelligent agents need not only to be rational, but also autonomous, that is, they should act independently by external control and learn to compensate for their partial or incorrect prior knowledge of the world.

Based on these ideas, the hereby presented project involved creating an autonomous and rational (multi-) agent system to play the game of Deliveroo. The task is inspired by the real-world problem of courier services, where parcels have to be delivered to different locations in a timely and efficient manner. In more details, the game is played on a two-dimensional map, where parcels are randomly generated and spawn with a random reward value that may decay over time. The objective of the agents is to collect and deliver parcels to the designated delivery locations in order to maximize their total reward.

To make the problem more challenging, the agents have to deal with a number of constraints and limitations. First, while the underlying mechanics of the game are known, the environment is only partially observable to the agents which can only perceive their surroundings within a certain radius. Second, the agents have to deal with the stochastic and dynamic nature of the game, as the parameters of the game (e.g. the number of parcels, their reward distribution, the movement speed of the agents) can change from one game to another. As an additional challenge, other agents can be present in the environment and compete for the same resources. Such agents can be either adversarial or cooperative, thus requiring the agents to adapt their strategies accordingly.

In the following sections, we describe the design and implementation of the implemented agent system. While the original project required the implementation of both a single-agent and a multi-agent system, here we focus on the latter as the single-agent system is a special case of the multi-agent system (no communication and no Hungarian matching). We will also present the results of the experiments conducted to evaluate the performance of our system. Finally, we will discuss the challenges and limitations of our approach, and suggest possible improvements for future work.

2 Background

Before we delve into the details of our work, we will provide a brief overview of some of the key concepts that are relevant to our work.

2.1 Monte Carlo Tree Search

Monte Carlo Tree Search (Kocsis and Szepesvári 2006) is a heuristic search algorithm applied to decision processes, that has been successfully used in a variety of games, such as Go and Chess. The algorithm is based on the idea of performing a guided random search that focuses on exploring the more promising moves by expanding the search tree through randomly sampling the search space instead of a

brute force approach that completely visits the search space. More in details, the algorithm builds the game tree by iteratively repeating the following four steps (see Figure 1 for a visual representation):

1. **Selection**: starting from the root R, which represent the current game state, traverse the tree until a leaf L is reached. A key aspect for the proper functioning of the algorithm is being able to balance the *exploitation* of already promising paths and the *exploration* of children with fewer simulations. The most used formula to compute such trade-off is the UCT (Upper Confidence Bound 1 applied to Trees)

$$\texttt{UCT} = \underbrace{\frac{w_i}{n_i}}_{\text{exploitation term}} + c \cdot \underbrace{\sqrt{\frac{\ln N_i}{n_i}}}_{\text{exploration term}} \quad (1)$$

   where

   - w_i is the number of wins obtained in the subtree of i
   - n_i is the number of times node i has been visited
   - N_i is the number of time the parent of node i has been visited
   - c is the exploration parameter, usually equal to $\sqrt{2}$

2. **Expansion**: if L isn't a terminal node (i.e. there are valid moves that can be performed starting from the game state in L), pick a node C among its children that haven't been yet expanded

3. **Simulation**: starting from C, randomly choose valid moves until a terminal state is reached and the game is decided (i.e. win/loss/draw)

4. **Backpropagation**: the result of the simulation is used to update the statistics (number of wins and number of visits) for all nodes along the path from C to R that are then used to compute the UCTs for the following iterations

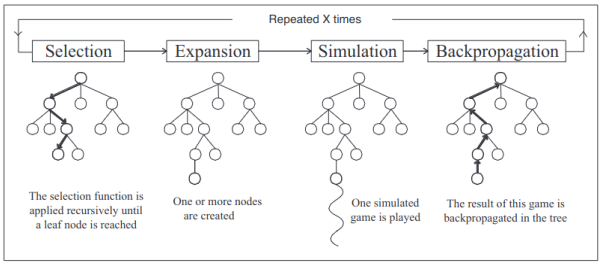

Figure 1: MCTS example for a two player game such as chess.

3 Method

Different architectures can be adopted to design an agent program implementing the underlying agent function mapping from percept histories to actions. Given the necessity to operate in a partially observable and dynamic environment in a proactive fashion, we based our implementation on the Belief-Desire-Intention (BDI) framework. By explicitly representing the agent's beliefs, desires and intentions, the BDI model allows the agent to reason about its environment, its goals and its own internal state, and to plan and act accordingly.

The pseudo-code of Algorithm 1 provides an high-level overview of the main parts of the agent program. After initializing its belief base (denoted as B) and its goals (denoted as D) based on its initial percepts, the agent enters the main control loop. At each iteration, the agent updates its set of beliefs based on its current observation of the surrounding environment and on the observations made by the agents cooperating with it. The newly updated belief base is then used to revise the agent's desires that are promptly exchanged with the other member of its team. Based on its own internal state and on the desires of the other agents, the agent then select from its set of desires the one to pursue. The chosen intention together with the current knowledge of the environment is then used to formulate a plan (i.e. a sequence of actions) to perform in order to achieve the intention. Finally, the agent executes the first action of the plan, updates its belief base based on the effects of the ac-

tion. Given the highly dynamic nature of the environment, a cautious approach is adopted, where the agent never commits to execute a plan in its entirety, but instead re-evaluates its desires and intentions after each action.

Algorithm 1 Agent control loop

```
Input: B_0 (initial belief base)
B ← B_0
D ← options(B)
while true do
    perceive o
    o ← exchangeObservations(o)
    B ← update(B, o)
    D ← revise(D, B)
    D ← exchangeDesires(D)
    I ← select(B, D)
    P ← plan(B, I)
    α ← head(P)
    r ← execute(α)
    B ← update(B, r)
end while
```

Before delving into the details of the implementation, some clarifications need to be made. While for the sake of simplicity we have presented the agent control loop as a sequential process, in practice most of the operations are performed concurrently. Indeed, this is a paramount requirement in a highly dynamic environment such as the one of the Deliveroo game, where changes in the environment can happen even while the agent is reasoning, planning and acting. Thus, all perceptions, communications and updates are performed asynchronously and in an event-driven fashion such that updates to the belief base and the desires are performed as soon as new information is available.

3.1 Team Communication

As stated in the introduction, not all agents in the environment are adversarial. In fact, agents may form teams and cooperate with each other to better achieve their goals. To allow the formation of teams and the exchange of information among the agents (necessary for an effective and productive cooperation), a communication protocol has been implemented.

The protocol is built upon two communication primitives made available to the agents by the engine: `shout` and `say` (a third primitive, `ask`, is also available, but it has not been used in our implementation). The `shout` primitive allows the agent to broadcast a message to all the agents in the environment, while the `say` primitive allows the agent to send a message to a specific agent. Even though the provided communication channels are unreliable (i.e. there is no guarantee about when or if the message will be received), no reliability layer has been implemented on top of them, as the agents are designed to be robust to temporary communication failures. In fact, temporary inconsistencies in the belief bases of the team members should not impact the overall performance of the team in the long run.

Since agents do not know in advance which other agents are in the environment and which of them are adversarial or cooperative, the first step of the communication protocol is to establish the identity of the agents and to form teams. To this end, each agent broadcasts a `hello` message to all the agents in the environment. Since agents may appear at any time during the game, the `hello` message is periodically broadcasted. To allow the receiving agent to verify that the sender of the message is indeed a legitimate agent, the `hello` message contains the id of the sender encrypted using the AES-256 algorithm with a secret key known only to the agents of the same team. Upon receiving a `hello` message, the receiving agent can verify whether the decrypted id matches the id of the sender and, if so, it can add the sender to its set of cooperating agents. In this way, even if a malicious agent intercepts the `hello` message and tries to replay it, the receiving agent will notice that the id of the sender does not match the decrypted id and will discard the message.

The other part of the communication protocol regards the exchange of information among the agents in the same team. To avoid adversarial agents from exploiting the information exchanged among the team members, the following messages are not broadcasted to all the agents in the environment, but are

instead sent only to the agents in the same team. Moreover, to avoid malicious agents from injecting false information in the belief bases of the team members, messages sent by non-cooperating agents are ignored. To this messages no encryption is applied since the `say` primitive should already guarantee that the message is received only by the intended recipient.

The exchange of information regards three main aspects: the observations of the environment, the desires of the agents, and the position of the agents.

The observation of the environment is the main source of information for the agents to update their belief bases (see Section 3.2 for more details). Each time an agent receives a new observation of the environment, it relays the observation to the other agents in the team. By doing so, the belief bases of the agents in the same team are kept in sync and their knowledge of the environment is both more accurate and more complete. One could argue that it would be better to only relay updates to the belief base, rather than each observation (which are much more frequent). However, we noticed that this leads to a higher risk of inconsistencies since temporary inaccuracies in the belief base of one agent will be propagated to all the other agents in the team.

As for the observations, each time an agent updates its desires, it relays the new desires to the other agents in the team. In this way, the agents in the team can coordinate their actions and not interfere with each other (e.g. by not picking up the same parcels) to better achieve their goals. Finally, each time the position of an agent changes, it relays its new position to the other agents in the team.

3.2 Belief base

In the BDI model, the belief base represents the information that the agent has about the world. In the context of the Deliveroo game, the informational state of the agent includes both the ground truth facts about the environment and the agent's own beliefs about the current state of the world. The former includes the topology of the map, the location of the pickup and delivery locations, and other parameters controlling the game dynamics. The latter includes the agent's beliefs about the state of the parcels and the other agents, as well as the agent's own state (e.g. its current location and its membership to a team). While the ground truth facts are fixed and given as input to the agent when the game starts, the agent's beliefs require to be constantly updated based on the received observations so as to maintain the most accurate representation of the world on which to reason and plan.

Hereafter, we describe the main assumptions and design choices made to keep track of the changes in the environment and to update the belief base accordingly.

Parcels One of the main components of the belief base is the set of parcels that the agent can pickup (that is, the parcels that are not yet picked up by any agent). Each parcel is represented as a tuple (p, l, t, v), where p is a unique identifier, l is the location of the parcel, t is the time at which the parcel was first observed, and v is the value of the parcel at time t (by knowing the time at which the parcel was first observed, the agent can estimate the current value of the parcel based on its decay rate).

Each time the agent receives a new observation (from its sensors or from the sensors of the other team members), it updates the set of parcels in its belief base accordingly. In particular, if the agent observes a free parcel that it was not aware of, it adds the parcel to its belief base. If the agent observes a parcel that it was already aware of but in a different location or now picked up by another agent, it updates its location or removes it from its belief base accordingly. Given that the agent can only perceive its surroundings within a certain radius, the state of the parcels that are not within the team's perception range can not be determined with certainty. In such case, we decided to mark as no longer free only those parcels whose last observed location is inside the perception range but that are not currently observed by any agent in the team.

Agents Tracking the state of the other agents is also a crucial part as their positions and intentions can greatly affect the agent's own plans. Here a dis-

tinction must be made between the agents in the same team and the adversarial agents. Tracking the state of the cooperating agents is straightforward, as their position and desires are directly communicated by the agents themselves. The only additional information that the agent needs to keep track of is the last time at which the agent has sent a message to the team. This is necessary to detect when an agent has been inactive for too long and remove it from the team.

On the other hand, keeping the belief base up-to-date with the state of the adversarial agents is more challenging as we can only rely on the observations made by the agent itself and its team. Each of the adversarial agents is represented as a tuple (a, l, t, r), where a is a unique identifier, l is the last observed location of the agent, t is the time at which the agent was last observed, and r is a boolean flag indicating whether the agent is considered a random agent. Note that by random agent we do not necessarily mean an agent that moves randomly (i.e. without any specific goal), but an agent whose behavior does not seem to be rational (i.e. whose actions do not seem to be aimed at maximizing its reward).

Each time the agent receives a new observation, it updates the set of adversarial agents in its belief base accordingly. In particular, if the agent observes a new agent, it adds the agent to its belief base, otherwise it updates the last observed location and the last observed time of the agent. Given that the agent can only perceive its surroundings within a certain radius, the state of the adversarial agents that are not within the team's perception range can not be determined with certainty. In such case, since agents are much more dynamic than parcels, we decided to mark as undefined the position of all agents that have not been observed for a certain amount of time. This is done to avoid the agent to make decisions based on outdated information.

To assess whether an agent is randomly moving, a simple heuristic has been defined. If from the first time the agent was observed to the last time it was observed its score is below a certain threshold, the agent is considered a random agent. The threshold is defined as the expected reward a greedy agent would have obtained in the same time span. This is computed in the following way:

$$\mathbb{E}_{\text{greedy}}[r] = \frac{\mathbb{E}[dist]}{\mu_{\texttt{dist}}} \cdot \frac{\mu_{\texttt{reward}}}{n_{\texttt{smart}}}$$

where $\mathbb{E}[dist]$ is the expected distance covered by the agent in the time span, $\mu_{\texttt{dist}}$ is the average distance between parcels in the map, $\mu_{\texttt{reward}}$ is the average reward of the parcels, and $n_{\texttt{smart}}$ is the number of agents in the environment that are not random agents.

Finally, note that here we do not make any assumption about the possible intentions of other agents. Indeed, modelling the behaviour of other agents is a complex and challenging task that may require learning-based approaches. Therefore, we preferred not to implement any partial model of the other agents' intentions, given that wrong assumptions about the actions of the other agents can lead to suboptimal decisions.

3.3 Search Tree

In the BDI model, the desires represent the goals that the agent wants to achieve. In the context of the Deliveroo game, at each time instant the goals of the agent are to move to a location where there are parcels or to deliver the currently held parcels (if any). Therefore, each time there is a change in the set of available parcels, the agent needs to update its desires accordingly.

Since the long-term objective of the agent is to maximize its reward, not all desires are equally important and worth pursuing. Therefore, it is necessary to assign them a score that should reflect their goodness for the agent's long-term goal. Hence, we deemed not sufficient to assign them a score based on their immediate reward, as this would lead the agent to be short-sighted and not to take into account the future consequences of its actions. To avoid this, we decided to implement a forward-looking scoring function based on Monte Carlo Tree Search. By using the MCTS algorithm, the score of each desire depends not only on the immediate reward, but also on the expected future reward that the agent can obtain after pursuing the desire.

Hereby we describe the main design choices and modifications made to the MCTS algorithm to make it more suitable for a highly dynamic and real-time environment such as the Deliveroo game.

Instead of searching over all possible move actions (up, down, left, right) that the agent can perform, to reduce the search space and to make the search more efficient, we decided to search over the possible desires that the agent can pursue at a given time instant. Thus, moving from one node of the search tree to another corresponds to pursuing a desire and each node corresponds to the state of the game if the agent were to execute all the desires along the path from the root node to the current node. Thus, we have that the root node corresponds to the current state of the game, its children correspond to the possible desires that the agent can pursue at the current time instant (among which the agent will choose the best one to pursue), the children of each child correspond to the remaining possible desires that the agent can pursue after pursuing the desire corresponding to the parent node, and so on.

With respect to standard applications of MCTS, we decided not to model the state of the game that is not under our control, except for the decaying of the parcels. In fact, while MCTS is often used to model processes where the only uncertainty is given by the opponent's actions, in the Deliveroo game the uncertainty is much higher and computing the full state would require modelling the behaviour of other agents and the dynamics of the environment.

Since we no longer model the randomness of the environment, the utility is no longer an expectation over the possible outcomes of the game, computed as the average of the rewards obtained by simulating the game from the node state. Instead, the utility of a node is the exact reward that the agent can obtain from the given state of the game. Since we are interested in the best desire to perform, the utility of a node is equal to the maximum utility of its children plus the immediate reward the agent can obtain by pursuing the desire corresponding to the node. In particular, if the desire is a pick-up then its immediate reward is 0 (since the agent does not obtain any reward by picking up a parcel), while if the desire is a delivery then its immediate reward is the value of the delivered parcels (at the time of the delivery).

Since the search must satisfy real-time constraints, some optimizations have been made to make it more efficient. First of all, in the play-out phase (i.e. the phase in which the game is simulated from the current state to the end), instead of throwing away the simulated subtree at the end of the play-out, we decided to keep it. While this increases the memory usage, it allows the agent to reuse the subtree in the next iterations of the search without having to recompute it from scratch. Furthermore, since before selecting the best desire to pursue the number of played iterations may be limited, we decided to expand the children of a node based on their greedy score (i.e. the reward obtained by pursuing that desire followed by a put-down) as to focus the first iterations of the search on the most promising desires.

Finally, in traditional applications of MCTS, the search is restarted from scratch after each move. Doing the same in our case would be too inefficient and would not allow the agent to reuse the information gathered in the previous iterations of the search. Therefore, we decided to adopt a dynamic search strategy, in which the tree is modified in place as the game progresses. In particular, each time the agent moves the state of the root node is updated with the new position, while, when the agent completes a desire, its corresponding child is promoted to the root node and the other children are discarded. Furthermore, as new parcels appear or disappear, the nodes corresponding to the pick-up desires are added or removed from the tree.

3.4 Selection

The selection phase is the process of choosing which desire to pursue next. As stated above, the possible desires that the agent can pursue at the current time instant correspond to the children of the root node of search tree. Thus, the most intuitive way to select the desire to pursue would be to choose the desire with the highest score. However, given the presence of other agents in the environment, the choice of the next intention must necessarily take into account the state of the other agents and the desires of the other team mates.

To take into account the state of the non-cooperating agents, the score of each desire is adjusted based on how likely the agent can achieve it without being interfered by the other agents. To this end, the score of each pick-up desire is decreased if there are other agents in the vicinity that are more likely to pick up the same parcels first. This is done by decreasing the score of the desire by a factor inversely proportional to the distance of the closent adversarial agent to the pick-up location. More formally, the score of each desire is adjusted as follows:

$$\texttt{factor} = 1 - \frac{1}{1 + \min_{a \in A} \texttt{dist}(a, l)}$$
$$\texttt{score} = \texttt{score} \cdot \texttt{factor}^2$$

where A is the set of non-random adversarial agents, l is the location of the parcel, and $\texttt{dist}(a, l)$ is the distance between the agent a and the location l.

To make the cooperation among the team members more effective, we need to ensure that no team members are pursuing the same desire at the same time. Even better, we should try to distribute the work among them such as to maximize the overall reward. To find the best one-to-one assignment of the desires to each team member Hungarian Matching (Kuhn 1955) has been used. Since the intention to pursue is recomputed after each action, to avoid a frequent reassignment of desires among team members, the score of a desire (for the next time instant) is adjusted based on the current assignment of desires to the team members. In particular, the score of each desire is decreased if it has been previously assigned to another team member, while it is increased if it has been previously assigned to the agent itself.

A particular case that needs to be handled is when the agent is carrying some parcels but it cannot reach any delivery location. In such case, the agent sends a `ignore-me` message to the other agents in the team to inform them that it should not be taken into account when assigning the desires to the team members. Meanwhile, the agent will start moving towards its nearest team mate to deliver the parcels to them. As soon as the agent delivers the parcels, it sends a `resume-me` message to the other agents in the team to inform them that it should be taken into account again when assigning the desires to the team members.

Finally, it may happen that the reward the agent can obtain by pursuing the chosen intention is not worth the effort. In such cases, it would be better to move to another location of the map hoping to find better opportunities. To this end, when the score of the chosen intention is equal to 0, the agent starts moving to the most promising position of the map. To find the most promising position a simple heuristic has been defined. Intuitively, a position on the map is more promising if it is close to a spawning point but far from other agents. More formally, the score of each position is computed as follows:

$$\texttt{score} = \left(\sum_{p \in P} e^{-\frac{1}{2}\left(\frac{\texttt{dist}(p,l)}{\sigma}\right)^2}\right) \cdot \prod_{a \in A} \left(1 - e^{-\frac{1}{2}\left(\frac{\texttt{dist}(a,l)}{\sigma}\right)^2}\right)$$

where P is the set of spawning points, A is the set of all agents, l is the location of the position, $\texttt{dist}(p, l)$ is the Manhattan distance between the spawning point p and the location l, and σ is a scaling factor that determines the radius of the influence. As can be seen, the score of each position is the product of two terms. The first term is the sum of the positive Gaussian functions centered at the spawning points, while the second term is the product of the negative Gaussian functions centered at the other agents.

3.5 Planning

Once the intention to pursue has been selected, the agent needs to formulate a plan to achieve the intention. In the case of the Deliveroo game, the plan consists of a sequence of move actions (up, down, left, right) to perform to reach the desired location. In other words, the planning problem can be seen as a shortest path problem on a two-dimensional grid.

Given that the computation of the shortest path is often required (after each action and during the whole search phase), recomputing the shortest path from scratch each time would be both inefficient and unnecessary. Therefore, considering also the fact that

the map layout does not change except for the position of the agents, we decided to precompute the shortest path from each tile to each other tile of the map and to store the results in a look-up table. To this end, we decided to adopt the Seidel's algorithm (Seidel 1995) to compute the shortest path between each pair of tiles. By treating the map as an undirected and unweighted graph (where each walkable tile is a node and each edge connects two adjacent tiles), the algorithm computes the shortest distance (that can be easily used to compute the shortest path) between each pair of nodes in $O(V^{\omega}log(V))$ time, where V is the number of nodes and ω is the exponent of matrix multiplication. Given that the number of nodes in the map is relatively small, the precomputation of the lookup table can be performed in a reasonable amount of time that will be repaid by the speedup in the planning phase.

As stated above, the precomputed paths do not take into account the current state of the environment, that is the occupied positions. Thus, when the agent perform the first move of the plan, it may find the destination tile to be occupied by another agent. In such case, the agent needs to recompute the shortest path to the destination tile, taking into account the current state of the environment. To this end, we decided to leverage a PDDL planner to give us a viable path. In our case, the PDDL domain is relatively simple as the only possible actions are up, down, left, right while the PDDL problem is created taking into account the agents' positions every time there is the need to recompute the path. As the planning computation needs to be performed in real-time, rather that using the provided online planner (that would introduce further latency), we decided to use the Planutils tool (Muise et al. 2022) to locally run the FF PDDL planner.

To further speed up the planning phase, we also implemented a solution based on the A* search algorithm (Hart, Nilsson, and Raphael 1968). By further exploiting the precomputed distances as the heuristic function (an admissible and consistent heuristic), the A* search is guaranteed to find the shortest path in the most efficient way.

The recomputed paths are then stored in a cache to be reused in the future. However, given the highly dynamic nature of the environment, an invalidation mechanism has been implemented to remove from the cache the paths no longer valid. To this end, the concept of path bottleneck is introduced. When moving from the current position to the destination tile, the set of all tiles that the agent must necessarily traverse to reach the destination in the shortest time is called the path bottleneck (see Figure 2). In other words, the path bottleneck is equal to the intersection of all the shortest paths from the current position to the destination. Given such definition, it is easy to see that the path recomputation is necessary only when the path bottleneck is no longer free. Thus, as soon as the path bottleneck becomes free again, the recomputed path can be discarded and the Seidel's cache can be used again.

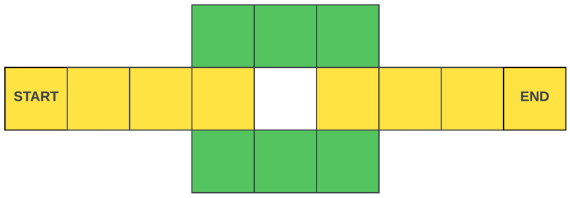

Figure 2: An example of path bottleneck. The tiles in yellow belong to the path bottleneck from the start position to the end tile.

4 Results

To evaluate the performances of our implementation, six different maps were used and each of them focuses on a different facet of the problem (e.g. map structure, parcel rewards, etc.). The tests were conducted by running our agent on each map for five minutes and reporting the final score. It is important to note that this score is only an estimate as both the condition of the map and the behaviour of the agent have a stochastic component to them and thus are not perfectly reproducible.

4.1 Single Agent

The evaluation for the single agent was carried out on the following four maps. The results on all of them are shown in Table 1 together with the scores achieved by the `BenchmarkAgent` provided by the course lecturer.

	Benchmark	Our Solution
Challenge 21	90	350
Challenge 22	6	613
Challenge 23	698	3219
Challenge 24	637	1470

Table 1: Scores for the single agent maps

Challenge_21 Challenge with the simplest map, consisting of a fully connected square map with non-decaying parcel rewards. Moreover, the map is characterized by a high number of randomly moving agents and a low agents observation distance.

Due to the number of available tiles, the initial precomputation of the distances between any two tiles is quite noticeable and takes several seconds. Nevertheless, the speedup in the planning phase more than makes up for this initial time loss.

Sometimes it may also happen that the agent performs sub-optimal actions such as ignoring parcels even if they are very close to the path it is already taking. This stems from the fact that the parcels' scores don't decay and thus the agent has no incentive to change the action it is currently performing to include a new parcel when it can simply collect it later and incur in the same reward.

Challenge_22 Challenge characterized by a large number of parcels with a small average reward and a fast moving agent.

This map is quite challenging as the rate at which parcels spawn and then die is very high. Moreover the agent is able to see up to a large distance and move fairly fast. This leads to having a lot of parcels that have to be taken into account at every point of the game and the MCTS is not always able to keep up with the frequent changes and come up with the best plan that considers all parcels. In order to at least partially mitigated this problem, each time a parcel expires or is picked up by another agent, the search tree is pruned and that parcel removed from all paths where it was previously considered to reduce the size of the tree.

Challenge_23 The map is characterized by narrow paths with many other agents moving in them, a limited number of available parcels at any time but with high rewards and an high parcel observation distance.

This tests how well an agent is able to navigate its surrounding environment and either modify its path to take into account the obstacles that are other agents or drop an intention all together to pursue a more promising one.

Challenge_24 This map differs from the other previous challenges because parcels are able to spawn only on some of the tiles and can be delivered in a single far away position.

Since the parcel observation distance is quite small, this challenge puts to the test the agent's ability to explore a map even when it isn't able to see any parcel. However, our solution exploits the spawning tiles to compute the promising positions to move towards as described in Section 3.4 which means that it performs well in this setting.

Due to the nature of our implementation, the main drawback is that once a parcel has been collected the agent promptly takes the shortest path towards the delivery position and therefore isn't able to observe any other spawning tile until after the parcel has been delivered. This continuous back and forth between one of the spawning tile and the delivery position could lead to a smaller reward than taking a longer path to reach a parcel but with the chance of seeing some other spawning tiles and collecting more parcels in the meantime.

4.2 Multi Agent

The evaluation for the multi-agent implementation was carried out on the following three maps. The results on all of them are shown in Table 2.

Challenge_31 The map is designed with vertical lines connected by an horizontal corridor. Both delivery and spawning tiles can be found at the ends of the vertical lines.

This challenge is complex due to the high number of parcels seen at any time, similarly to challenge_22, but with the added difficulty that the parcels are viewed by two different agents and therefore the information about them is shared asynchronously.

Because of the dynamic nature of the environment, the agents often change their intentions and the parcels they are going to pick up. This sometimes leads to agents starting to move towards their intention only to then receive a message from another teammate stating that it has a better score for that same intention forcing the agent to drop its intention. This may result in the agents moving back and forth, constantly switching their intention, while not being able to make any progress.

Challenge_32 This challenge has a unique configuration with vertical, separate lines with a spawning tile on one end and a delivery tile on the other and it tests the ability of the agents to effectively coordinate and collaborate to exchange parcels.

This map is where the `ignore-me` message described above is mainly used in order for the agent closer to the spawning tile to drop its parcels so that the agent closer to the delivery tile can successfully deliver them.

Another possible scenario is for the two agents to spawn in separate lines but this is less interesting because, while still sharing information about the position of parcels, there is no explicit need for coordination.

Challenge_33 The map's design is similar to that of challenge_31 but this map is divided in two, non-communicating, halves and the horizontal corridor connecting them is tighter.

While no direct mechanism to prevent the agents to block each other when moving in narrow corridors was implemented, we observed that this didn't happen frequently enough to hinder the performances of our solution. Even when this did happen the agent would simply select another parcel to pickup, if any was available, or exchange parcels with the other agent if no other action was possible.

	Our Solution
Challenge 31	1352
Challenge 32	5473
Challenge 33	1938

Table 2: Scores for the multi-agent maps

5 Conclusions

We have presented a multi-agent system that uses Monte Carlo Tree Search to solve the problem of collecting and delivering parcels in a dynamic environment. As can be seen from the results reported (see Section 4), our solution provides satisfactory results on all the maps we have tested it on. Still, there are some limitations that should be addressed in future work.

First of all, the current implementation of MCTS cannot keep up in case of many and frequent changes in the environment. This is due to the fact that the tree must be constantly modified and pruned and too few iterations are performed to come up with a good plan. Thus, future focus should be on improving the efficiency of the MCTS algorithm, for example by parallelizing the search.

Another issue regards the coordination between agents. In the current implementation, the search tree is built independently by each agent limiting the coordination to the exchange of messages and for the intention selection. This means that the search does not take into account possible interference between the agents at a depth greater than one (that is, after the immediate next move).

A possible solution to this problem could be to move from a distributed to a centralized approach, where a leader is responsible for coordinating the agents and building a global search tree. Still such solution is not without its drawbacks. Other than being a single point of failure, a global search tree that take into consideration all possible combinations of actions for all agents would be too large to be

practical. While there exists decoupled implementations for cooperative MCTS (Asik, Aydemir, and Akın 2023), such solutions do not allow to update the tree in real time but require to start from scratch every time the environment changes.

Therefore, future work should focus on finding a better balance between the two approaches, possibly by keeping a local search tree for each agent with a modified version of the UCT algorithm that takes into account the future actions of the other agents. This would allow to keep the advantages of a distributed approach while still being able to coordinate the agents and take into account the interference between them.

References

Asik, Okan, Fatma Başak Aydemir, and Hüseyin Levent Akın (2023). "Decoupled Monte Carlo Tree Search for Cooperative Multi-Agent Planning". In: *Applied Sciences* 13.3, p. 1936.

Hart, Peter E., Nils J. Nilsson, and Bertram Raphael (July 1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2, pp. 100–107. ISSN: 2168-2887. DOI: 10.1109/TSSC.1968.300136. URL: https://ieeexplore.ieee.org/document/4082128.

Kocsis, Levente and Csaba Szepesvári (2006). "Bandit Based Monte-Carlo Planning". en. In: *Machine Learning: ECML 2006*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 282–293. ISBN: 978-3-540-46056-5. DOI: 10.1007/11871842_29.

Kuhn, H. W. (1955). "The Hungarian method for the assignment problem". en. In: *Naval Research Logistics Quarterly* 2.1-2, pp. 83–97. ISSN: 1931-9193. DOI: 10.1002/nav.3800020109. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800020109 (visited on 02/03/2024).

Muise, Christian et al. (2022). "PLANUTILS: Bringing Planning to the Masses". In: *32nd International Conference on Automated Planning and Scheduling (ICAPS 2022). System Demonstrations*. AAAI Press.

Russell, Stuart and Peter Norvig (2010). *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall.

Seidel, R. (Dec. 1995). "On the All-Pairs-Shortest-Path Problem in Unweighted Undirected Graphs". In: *Journal of Computer and System Sciences* 51.3, pp. 400–403. ISSN: 0022-0000. DOI: 10.1006/jcss.1995.1078. URL: https://www.sciencedirect.com/science/article/pii/S0022000085710781.