AASMA Project Report

Adrian Manica 97076
André Leite 86383
Victor Ribeiro 97075

ABSTRACT

This report provides a study of the agent's cooperation and competition in a kitchen environment. We modelled two different scenarios, one corresponding to a homogeneous society, and the other a 'Master Chef' type of competition between two agents. For the first scenario, we have one brain that controls the distribution of tasks between all the different Chefs. For the second, we have two agents, one whose behaviour can be defined before runtime and is static for the duration of the execution, and another that adapts its behaviour in response to the other agent's achievements. Regarding the first scenario, our experiments displayed that the way in which we sort the importance of each *Dish* directly affects the final reward the restaurant obtains. Our sorting algorithm, which not only considers the customers' time waiting for the *Dish* but also how expensive (reward) the *Dish* is, proved to be slightly more efficient than the algorithm that mimics real life. Regarding the second scenario, our results show that the 'Copy-cat' style agent achieves good results in a probabilistic environment, but lacks some 'self-restraint' and some ambition to increase its winning advantage.

Keywords

Multi-Agents, Kitchen, Hostility, Competitive.

1. INTRODUCTION

This project was developed in the context of Autonomous Agents and Multi-Agent Systems course. Our goal for the project was the development of two different decision agents to study the decision making in a time-pressured, probabilistic environment and how some trust strategies apply in it.

We decided to take on the problem by scaling it down to represent the movement of Chefs in the kitchen and the decision making related to which Dish to cook.

For this project, we decided to focus on answering the following questions:

1. Scenario 1:
    a. Which parameters affect the reward rate the most?
    b. Is there an optimal sorting order for the Dishes, if so, what is it?
2. Scenario 2:
    a. Which parameters affect the reward the most.?
    b. How does copycat perform in a probabilistic environment?

2. PROBLEM

The main goal of our project consists of studying the agent's reward in a probabilistic and volatile environment. Thus, we modelled our problem by considering which real-life characteristics would be the most interesting to adapt to our system.

Environment

We begin by modelling the environment where our agents will cooperate. We have different stations spread out around the environment that is accessible by all agents but only support one at a time. Each Station connects to a specific action that the agents can perform (e.g., getting, cutting, cooking ingredients, etc..). They are stationary for the duration of the simulation and are related to no more than two different actions.

We can define our environment through the following properties:

1. First scenario
    a. Accessible – all agents have complete and up-to-date information about the network.
    b. Deterministic – each action taken by an agent has a guaranteed result.
    c. Static – random events take place before the agent's deliberation.
    d. Discrete – there is a finite set of actions the agents can perform.
    e. Episodic – simulations are independent of each other.
2. Second scenario
    a. Accessible – all agents have complete and up-to-date information about the network.
    b. Non-deterministic – actions performed by agents do not have a guaranteed outcome.
    c. Dynamic – random events take place after the agent's deliberation.
    d. Discrete – there is a finite set of actions the agents can perform.
    e. Episodic – simulations are independent of each other.

Agents

In our multi-agents system, we considered three types of agents: Chefs in a homogeneous society, a static Chef, and a 'Copy-cat' style Chef adapted to non-deterministic environments.

For the first scenario, the agents are represented by an 'all-seeing' brain that controls the actions of all agents. We will call this brain *'Gordon Ramsay'*. *Gordon* is responsible for:

1. Taking in the requests from the restaurant.
2. Sort the requests with the chosen sort function.
3. Send the sorted requests to the available Chefs.
4. Handle the reward received by the Chefs.

Gordon is parameterized by different properties that influence its behaviours, such as:

- The sorting order for the requests.
- The time interval in which the requests come into the queue.

For the second scenario, the agents are both 'single-minded' and can be of one of two types: *'Static'* or *'Dynamic'*. The risk behaviour property can parameterize both agents. This property dictates the agent's willingness to perform a higher-risk action. The difference between the two agents lies in the way this

property is managed. For *Static* agent's it is determined before runtime, while the *Dynamic* agents alter its value according to the other agent's success.

The following properties can also parameterize the *Dynamic* Agent:

- The Threshold from which it will start to change its risk behaviour.
- The amount in which it will change its risk behaviour.
- The ability to change its risk behaviour following its Threshold.

For both scenarios, the Chefs are capable of:

1. Navigating the environment.
2. Reserving a station for their use.
3. Performing actions about a specific Dish.

3. SOLUTION

For this project, we decided to build a Unity implementation where a class represents each agent. This allowed us to build an environment where each agent is a Unity game object.

Environment Representation

For both scenarios, the environment surrounding the agents is analogous. There are *Stations* which represent different spots in a kitchen (e.g., fridge, hotplate, cutting board, etc...). In the first scenario, each *Station* corresponds to a class that has a lock to use the *Station*. This ensures that only one Chef can use the *Station* at a time. In the second scenario, chefs do not need to reserve stations since they are always available.

First Scenario

This scenario represents a homogeneous society, where one brain (*Gordon)* controls the distribution of tasks between all the different chefs. *Gordon's* objective is to maximize the reward of the kitchen, which is analogous to what happens in real life (a restaurant having excellent service and customers leaving good reviews).

Gordon

The agent can contain multiple requests (Dishes in a queue) coming from the restaurant to assign to Chefs. Based on the current delay between requests arriving and the sort function applied, it will distribute the Dishes present in the queue. When a Dish is ready, *Gordon* will collect the reward, which is influence by the time it took to make it. There is a grace period for the Chefs to complete the Dishes in the queue, being equal for all Dishes, and it starts to count the moment the Dish enters the queue. If the time it took to deliver the Dish exceeds the grace period, the reward will be lessened with accordance to the following rules:

- *Time < GP => Reward = Reward*
- *GP < Time < GP * 2 =>*
  *Reward = (1 – ((Time-GP)/GP)) * Reward*
- *Time > GP * 2 => Reward = -10 * (Time/GP)*

Note: GP = grace period, Time = time since the Dish entered the queue until it was finished.

The sorting functions are the following:

- By the reward, they give (price) to the restaurant per unit of time.
- By the time left before the grace period ends
- If the difference between the time it takes to cook the Dish plus the time since it entered the queue is positive, the Dish gets sorted by that time difference. If the time is negative (there is not enough time until the reward starts deprecating), the Dish is given priority, and it gets sorted by the reward it will give per time unit. If the Dish is already in the time frame of giving a negative reward, the Dish is sorted by the time it arrived at the queue.

Chefs

The Chefs in the first scenario are responsible for executing the actions related to the Dish they were assigned. This includes getting the ingredients, cutting and preparing the ingredients, cooking, boiling and/or frying the ingredients, and assembling the Dish. The Dish is represented by a class that contains the ingredients needed for the Dish, the actions to perform, the *Stations* in which said actions should be performed, the time it takes to perform each of those actions, and the reward for delivering it. Concerning the ingredients, they are represented by a class with two main attributes: an *enum* that represents the ingredient itself, and its current state (whether its cut, cooked, and so on). Finally, this class also creates dictionaries of the type *<Ingredient, TimeToCompleteTask>* for each of the main tasks, which are cutting, boiling (cooking), and assembling.

Second Scenario

This is a competitive scenario where two chefs battle to see who gets the highest reward. *Gordon* is no longer present, instead, a class called *GameSession* is responsible for holding the information associated with all the *Dishes*, for updating the probabilities associated with the success rate of each *Dish*,

Advanced Chef

Advanced Chefs are only present in the second scenario and are represented by a class attached to a Unity game object. This class includes Risk Behaviour (RB), which describes the willingness of a chef to execute a specific probabilistic action (Dish). The parameter pertains to the inverse of the willingness to risk of the agent. The higher the RB, the lower the willingness to risk will be. This behaviour is defined before runtime and remains the same during the execution of the scenario, which means their demeanor is independent of the other Chef. Unlike the normal Chef, which executed tasks that were directly assigned by *Gordon*, this Chef will decide which *Dish* they consider will give them the best reward, while taking their respective Risk Behaviour and the *success probability* of the *Dish* into account. Every Dish has an initial *success probability* defined before runtime, but it will continuously change during the execution, depending on how often the chefs have success (or failure) in cooking it. This evolving *success probability* is called *session probability*, and it serves to encourage/discourage chefs from making specific *Dishes* based on their history. We made this design choice because, as seen in popular cooking shows like the aforementioned *Master Chef*, the psychology of the chefs during the competition impacts heavily what *Dishes* they decide to cook. For example, if a chef considers cooking a particular *Dish*, but remembers that their rival repeatedly failed at doing so, they will probably choose an alternative *Dish* to do. The opposite is also true when deciding to cook a *Dish* that has a high success rate.

When the *Advance Chef* tries to decide which *Dish* to cook, they will, for each *Dish*, calculate the *difference* between the *session probability* of the *Dish* and their Risk Behaviour, and choose the *Dish* with the lowest difference. For example, if their Risk Behaviour is 0.7, the *Gourmet Hamburger* has a *session*

probability of 0.65, and the *Tuna Pasta* one of 0.4, the Chef will ultimately choose the *Gourmet Burger*.

We update the session probability in accordance with a parameter defined in the Game Session, the *Learning Rate*.

SessionP = SessionP + Learning Rate(NewP - SessionP)*

Note: NewP is the new probability which comes from the current execution of the Dish by an agent.

Dynamic Chef

Dynamic Chefs are only present in the second scenario and are represented by a class attached to a Unity game object, which inherits from the Advanced Chef's class. The difference between the two is that Dynamic Chef changes its RB during the execution of the program following the difference in reward between the agents. The idea was based on making a 'Copy-cat' style agent but adapted it to a probabilistic environment. This poses various difficulties since just copying what the other agent does is not a good strategy if the outcome is not guaranteed. A better option would be to increase and decrease our willingness to risk based on the other player's performance. We do this by adding or subtracting to the RB when the difference between the other player's score and our own is more or less than the provided Threshold.

RB = RB + Threshold / 2000 if 'our score' > 'other's score' + Threshold

RB = RB – Threshold / 2000 if 'our score' + Threshold < 'other's score'

We also decided to include an option to pick our change rate (CR) of the RB, which gives:

RB = RB +CR if 'our score' > 'other's score' + Threshold

RB = RB – CR if 'our score' + Threshold < 'other's score'

This ensures our agent will risk more when it is behind the other agent and will have a steady, less risked behaviour when it is in front.

4. RESULTS

The simulations were run with a *timescale* of one and perform 3 runs for each of the given scenarios to guarantee the most accurate results possible. We ran all simulations for a period of two thousand seconds.

4.1 Scenario 1

To test the success of each sorting system, which decided to use the reward collected by the restaurant at the end of the simulation.

4.1.1 Parameters

The parameters used for the simulation were the following:

Gordon Parameters

Dish Probability to Spawn	Time to Generate a New Dish
8.3%	10 seconds

Chef Parameters

Movement Speed	Stopping Distance
3	2

4.1.2 Sorting by reward

The first sorting technic used was the sorting by reward, which returns the queue of Dishes sorted by the Dish's reward per unit of time.

We theorized that this sorting mechanism would go well if the time pressure in the environment is low (high value in the grace period parameter).

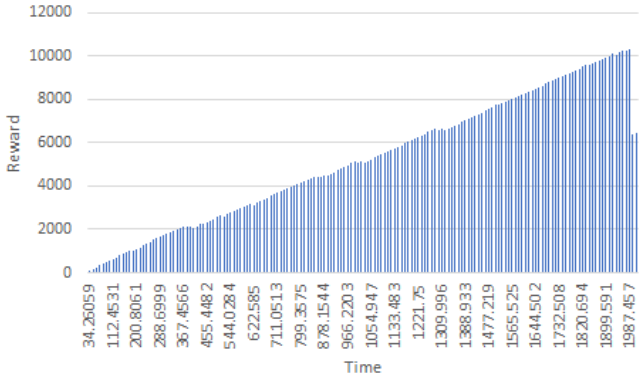

Figure 1

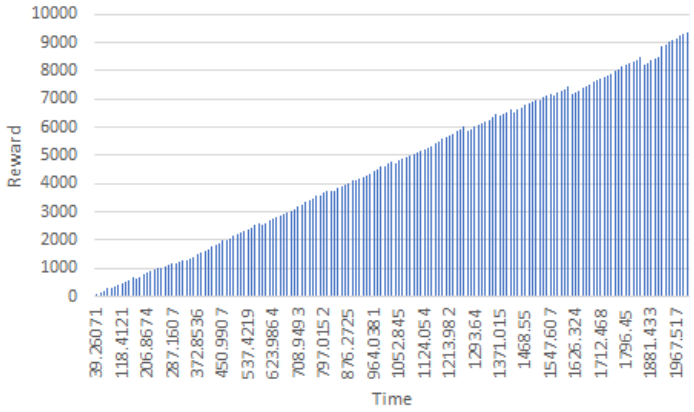

Figure 2

The parameters used for Figure 1 were:

- Ten seconds of an interval between requests arriving at the queue.
- Ninety seconds of grace period for each Dish.

The parameters used for Figure 2 were:

- Eleven seconds of an interval between requests arriving at the queue.
- Sixty-five seconds of grace period for each Dish.

We can see that this sorting mechanism performance leaves a lot to be desired when considering a high-pressure environment where the decisions need to be made with the time left in mind. This approach is only good for low pressure environments, however as we will see next, the other approaches work in bot environments, leaving this sorting mechanic with no practical use.

4.1.3 Sorting by time left

The second sorting mechanism applied was to sort the Dishes by the time left to complete them, taking into account the grace period as well as how long the Dish itself takes to cook.

Time Left = GP – (Dish.Time + Dish.TimeToComplete)

Note: Dish.Time = time since the Dish entered the queue.

We theorize that this sorting function will work best when the time pressure is higher, since the time a specific Dish stays 'idle' will significantly influence its final reward. Making sure that we

cook the Dish by order of arrival is analogous to real life and good enough for the system to work well.

The next figures illustrate the reward per time throughout the simulation.

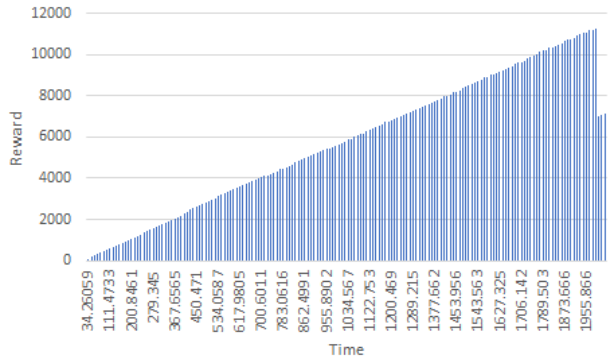

Figure 3

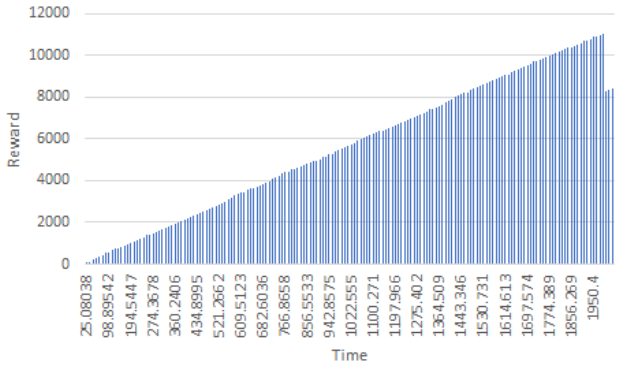

Figure 4

The parameters used for Figure 3 were:

- Ten seconds of an interval between requests arriving at the queue.
- Ninety seconds of grace period for each Dish.

The parameters used for Figure 4 were:

- Eleven seconds of an interval between requests arriving at the queue.
- Sixty-five seconds of grace period for each Dish.

We can observe that the reward gained is very similar. This is expected since the variability between runs of this sorting algorithm will come from the chance of having higher reward dishes arrive in the queue. Its behaviour is expected to be very similar in all situations when a restaurant is full of people or when it is empty.

4.1.4 Complex Sort

This sorting mechanism works by giving priority to the Dishes whose grace period is about to expire. It sorts by considering both the time left in the grace period as well as the reward obtained by executing the Dish.

Value = Reward / (GP – (Dish.Time + Dish.TimeToComplete))

If, however, the Dish's time exceeds the grace period, the corresponding Dish is given priority over the others.

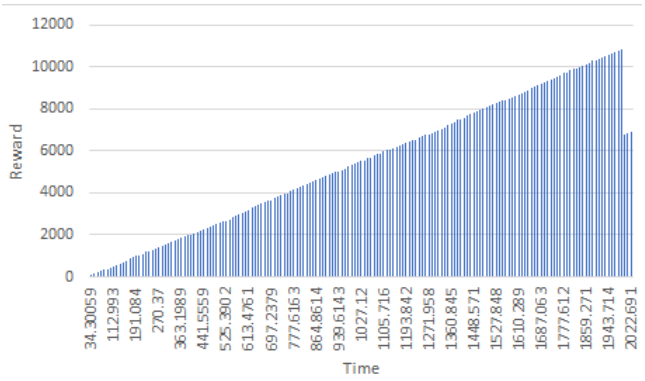

Figure 5

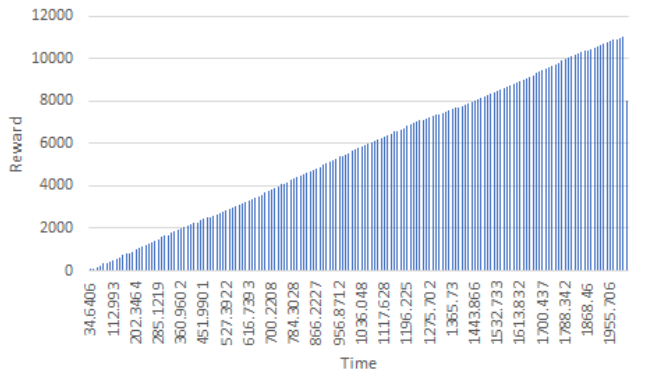

Figure 6

The parameters used for Figure 5 were:

- Ten seconds of an interval between requests arriving at the queue.
- Ninety seconds of grace period for each Dish.

The parameters used for Figure 6 were:

- Eleven seconds of an interval between requests arriving at the queue.
- Sixty-five seconds of grace period for each Dish.

The results are within the expectation. The agent achieves slightly better results in the second run, but it does not vary too much. We expect this sorting algorithm to perform around the same as the last one but have slightly better performance when the time pressure is more significant. We observed that the difference between the two algorithms is minimal with *Sorting by time left*, achieving a slight advantage. Nevertheless, we attribute this result to not being able to run the simulation enough times to achieve proper distribution and diminish outliers.

4.2 **Scenario 2**

To test the results of our agents, we decided to consider the reward gained by each agent at the end of the session.

4.2.1 Parameters

Game Session

Learning Rate	Session Time
0.216	2000 seconds

Advanced Agent

Moving Speed	Stop Distance
3	2

Dynamic Agent

Moving Speed	Stop Distance	Starting Risk Behaviour	Threshold	Dynamic Risk Change Rate
3	2	0.8	50	5

4.2.2 Reward for different Static Agent's Risks

This simulation was executed to test the hypothesis that the Dynamic Agent will at least perform like the other agent. Its reward will rarely fall too much behind the Static Agent's reward if we assume a sufficiently large number of samples. The Dynamic Agent should be able to outperform the Static Agent by a massive difference, but the opposite happening should be an outlier in the grand scheme of things. The most common result should be between the two agents ending up with similar rewards and the Dynamic Agent occasionally outperforming the Static Agent.

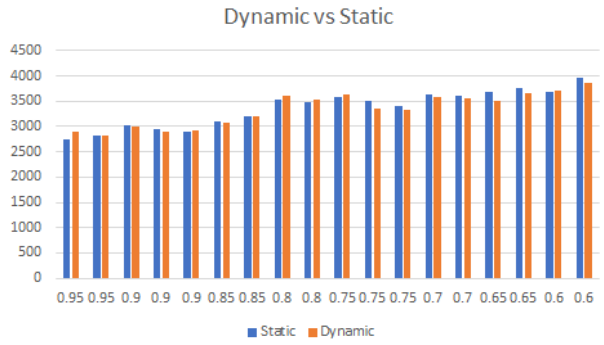

Figure 7

In the y-axis is the reward, while in the x-axis is the Static Agent's Risk Behaviour. We can observe that as the willingness to risk increases (lower Risk Behaviour value), the overall score increases. We can also confirm that our Dynamic Agent does an excellent job of keeping up with the Static Agent. Both scores vary in an almost identical matter.

4.2.3 Dynamic vs. High-Risk

In this simulation, we assume the agents' rewards will mostly be inconsistent and diverge. The Dynamic will start with a more stable reward (since its starting Risk Behaviour is higher in comparison to the High-Risk Agent). However, if the High-Risk agent hits a streak of good luck and performs several high-risk high-reward Dishes in a row with few or no failed tasks, the difference between the two agents' scores will increase and the Dynamic Agent will be forced to increase its willingness to risk (lower the Risk Behaviour).

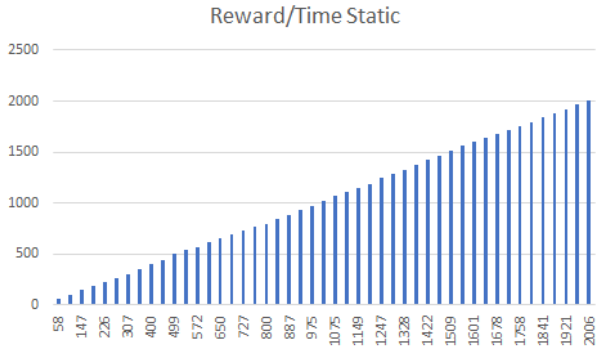

Figure 8

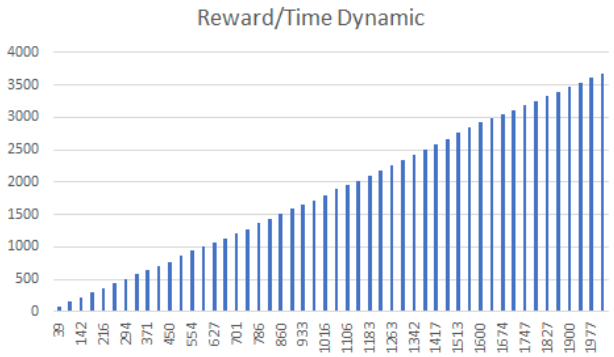

Figure 9

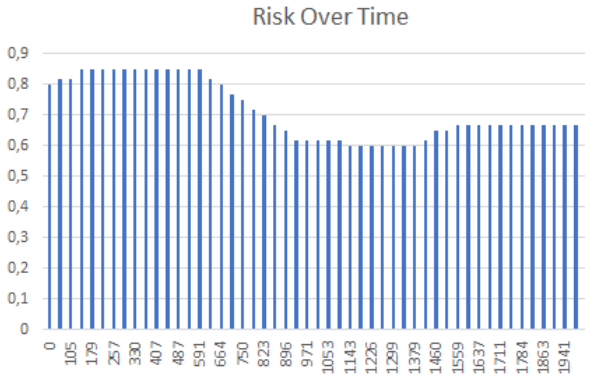

Figure 10

Figure 10 is the average of the *Dynamic Agent vs. The High-Risk Agent* scenario, which illustrates how most encounters went down. The *Dynamic Agent* started by going in front, which gave it a higher reward, but then it lost the advantage to the High-Risk Agent. Near the end, it caught up with it again and stayed stable until the simulation finished.

After 21 simulations, we quickly realized that it was more common for the *High Risk* agent to lose by a wide margin (in simulations where they had "bad luck") than to win by a vast margin against the *Dynamic Agent*. Nevertheless, in most cases, the final score of both agents was quite close.

4.2.4 Dynamic vs. Low-Risk

In this simulation, we assume Dynamic Agent will either outperform the Low-Risk Agent by a fair amount, or both agents scores will be very similar.

The Dynamic Agent will start with a higher willingness to risk (lower Risk Behaviour) in comparison to the Low-Risk Agent. This gives the Dynamic Agent, the opportunity to get ahead of the other agent, in which case, it will start to decrease its willingness to risk (increase Risk Behaviour) which will make its rewards more stable. This allows the Dynamic Agent to keep its score advantage. If, however, the Dynamic Agent starts with a streak of bad luck and the Low-Risk Agent gets ahead of it, the Dynamic Agent will increase or maintain its lower Risk Behaviour (higher willingness to risk) until it finally hits a streak of good luck and catches up to the Low-Risk Agent. At this moment, it will lower the willingness to risk (increase Risk Behaviour) and maintain a similar score to the Low-Risk Agent.

For the following data, we used a Static Agent with a Risk Behaviour of one (1.0).

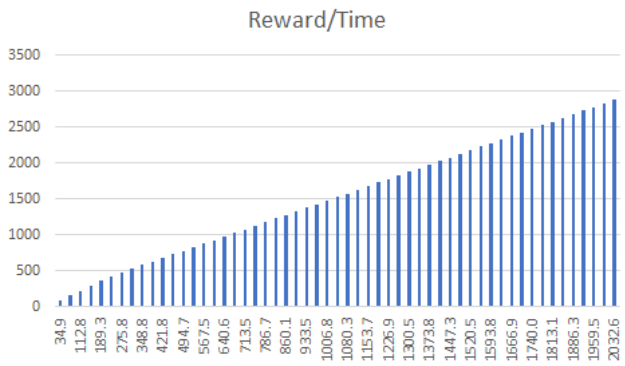

Figure 11

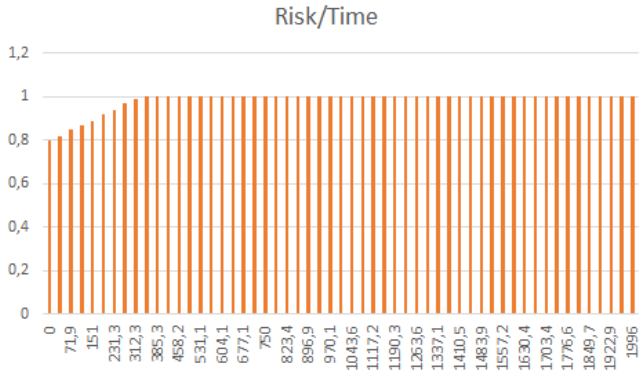

Figure 12

As we can observe, our hypothesis is confirmed by the results. The Dynamic Agent overcomes the Static Agent at the beginning with a streak of average/good luck and then proceeds to decrease its willingness to risk to maintain the said advantage. Since the Static Agent is Low-Risk, the reward received per Dish is not affected guaranteed and so it is never able to overtake the Dynamic Agent.

4.2.5 Risk Change Rate

The Risk Change Rate parameter allows the Dynamic Agent to react quickly when the other agent starts winning. The Risk Behaviour value will decrease at a greater rate, allowing the agent to take on more High-Risk tasks quicker. It does, however, have a downside since the changes might become too abrupt, both for when the player is behind or ahead and make it, so the agent exhibits fewer Behaviour's. The changes make the agent less stable overall, which can be a bad or a good thing, depending on luck.

4.2.6 Threshold

Changing the Threshold of the Dynamic Agent allows the agent to react quicker to the changes in the score. If the Static Agent starts getting ahead, the Dynamic Agent will react quicker, and increase its willingness to risk (lower the Risk Behaviour) faster. It does, however, have a downside, since when the Dynamic Agent gets ahead of the Static Agent, the lower Threshold will make it risk less and provide lower advantage, which can allow the other agent to overturn it quickly.

5. CONCLUSION

5.1 Scenario 1

In conclusion, the best sorting mechanism is, as we expected, the third, but in comparison with the second one, its advantages are very small. The first sorting mechanism has no practical use, since it would only be useful in an almost empty restaurant, where any sorting mechanism would work well, since the queue would be negligible. The second sorting mechanism, being the one that resembles real-life the most, we can conclude that this approach works well enough for a restaurant. As for the last sorting mechanism, it will only show itself useful when a certain restaurant has too many requests to handle all at once. It works by choosing the client whose expected future purchase would give the restaurant the biggest income. If we have two people waiting for their food, this sorting algorithm will take into account the time the client has been waiting, but also how much they are willing to spend to get food, and the type of dishes they order. This gives an overall slightly better reward for the restaurant, since we take advantage of the clients willingness to spend money and make sure they are happy.

5.2 Scenario 2

The Dynamic Agent is a perfect match against a Low-Risk agent since when it starts to get an advantage, it will be tough, if not impossible, to lose it.

On the other hand, when the Dynamic Agent plays against the High-Risk agent, it performs well if we want the agent to never lose by a high margin, but its performance is not good enough if we only consider winning as a success. This comes from the fact that the agent might be too slow to react to changes, as well as not consolidating its winning position. A Dynamic Agent with a High Threshold will consolidate the winning position but will be too slow to react to changes in the score. If we consider Risk Behaviour, the same applies, since when in a winning position, the agent will decrease its willingness to risk too fast, which can result in a low score advantage.

With this in mind, we propose for future work, the following changes in the agent architecture (considering winning as the only success):

- Give the agent two different Threshold values, one for when it is losing and another for when it is winning. This allows adequate control over the agent's Risk Behaviour changes over time. If the Dynamic Agent is losing, we want it to react quicker, while if it is winning, we want the agent to consolidate its advantage before starting to slow down.
- Give the agent two different Risk Behaviour values, one for when it is losing and another for when it is winning. This allows adequate control over the agent's Risk behaviour changes over time. Another possible solution would be a 'logarithmic' based change in Risk Behaviour so that the initial change in the value allows the agent to react quicker, but so that it slows down and does not 'overshoot'.
- Introduce a cap in the high a low possible value of both the Threshold and the Risk Behaviour.