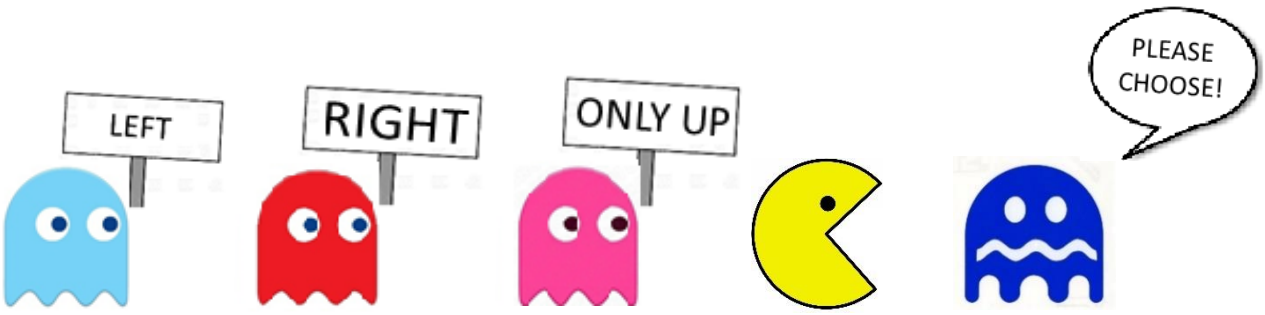
Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2019

RELEVANT EXPERTISE AGGREGATION



Student: **Chatzis M.**

CID: **01081134**

Course: **EEE4T**

Project Supervisor: **Dr Jeremy Pitt**

Second Marker: **Dr Daniel Goodman**

Acknowledgements

A special thanks to my supervisor, Dr Jeremy Pitt, for his patience and understanding during my final year project journey.

Contents

A. Abstract

Constant improvement of autonomous software systems, communication bandwidth and AI-specific hardware has created the opportunity for the use of self-organizing and multi-functional autonomous multi-agent system(MAS) solutions in tackling community-based future problems. Smart grids and traffic coordination of autonomous vehicles are just two examples of such problems where a MAS solution could severely outperform any other software paradigm. This report investigates a specific MAS reinforcement learning architecture which was inspired by a real life social structure, namely the Athenian Democracy of around 500 B.C. We borrow the idea of Relevant Expertise Aggregation from the 500B.C. scenario and apply it to a MAS system as a way for agents to vote and update their common ruleset which captures the knowledge gained in each iteration. For visual demonstration, the MAS developed is the game Pacman, where ghosts are the agents with common goal of capturing the Pac-Man. Three different voting methods are explored: equal-vote aggregation, relevant expertise aggregation(REA) and maximum expertise voting. The three methods' effectiveness in reaching the learning goal are compared. The project concludes with small but existent performance discrepancies between the algorithms with REA being slightly outperforming the rest.

A. Introduction

On its highest level of abstraction the project explores the concept of *Relevant Expertise Aggregation (REA)*, a voting method inspired by the epistemic democracy of classical Athens, as a means of learning the most effective common ruleset in a computationally simulated socio-technical, multi-agent system (*STMAS*).

Put in a less concise way, the project revolves around the development of a software algorithm. This algorithm exists inside a computer simulation of a multi-agent system. In this simulation, agents obey a common ruleset (One can draw parallels to the laws of society). Using the algorithm, the agents learn to form the most effective common ruleset. This process of learning the best ruleset is performed through voting. The agents vote for what they believe is the best ruleset according to their experience from interacting with their environment. The goal of this algorithm and hence of this project is to compare the effectiveness of 3 different ways of voting, simple vote aggregation, relevant expertise aggregation and total expertise voting.

These 3 different ways of voting are all based on the question of epistemic vs democratic voting. Who is right, the masses or the experts? Is the answer the same in real-life social environments and artificial computational systems? Can the one field borrow the solutions of the other? As in, can we design computational MAS solutions based on the success of REA in certain societies? To which problems do these solutions apply? All these are questions that the project aims to address and that will be explored in the following report.

The investigation on REA explained above will take place in a gaming environment. The game of choice is Pac-Man, a well-known game released in the 1980s with big success. A game was chosen as a platform for the investigation because it provides an easy way to determine success, an perfect visualization of the learning process and of the MAS system and ultimately a fun experience of winning the game with AI.

Section Breakdown

In the *Background* section the reader will find background information with regards to the field of research of the project and the relevant prior work needed to understand the rest of the report. Next, in the *Analysis and Design* section, an outline of the desing of the project in a high level of abstraction getting to a lower level of abstraction in the *Implementation* section which explains how individual parts and methods were implemented. Following, the *Results* section presents the results of the software simulation, which are then discussed in the *Discussion and Evaluation* section. Finally, conclusions are drawn in the *Conclusion and Future Work* section where also suggestions for further advancement of the project in the future are explored. For the interested reader the *User Guide*, *Bibliography and Appendices* follow, which are quite self-explanatory.

C. Background

C1. Political Science Perspective

When the voting method of a political regime is concerned two extremes can be defined, namely epistemic and democratic voting. The former is based on full deliberation on the proposal on hand by experts of the respective field of knowledge which ends with a decision. The latter is based on vote aggregation of all individuals participating in the democracy, either experts on the respective field or not.

Given the above, it goes without saying that the epistemic approach should in theory produce better results for the society at hand, due to the educated nature of the decisions taken. However, it bears a big price, namely the risk of de-democratization of the regime. In other words, since a portion of the democratic population makes the decisions, the core democratic values of justice, liberty and equality are directly threatened by the possibility of the formation of a tyrannical “elite”.

The democratic method negates this potential threat but at the severe cost of non-educated decisions which threaten the well-being of the society. Instead, we may consider a middle-way between the two extremes, an epistemic **and** democratic voting method. Based on (J. Ober)[1] a real-life version of an epistemic democracy was implemented in classical Athens as described in the passages of Aristotle [2] and was generally very effective although not perfect as argued in [1].

Relevant Expertise Aggregation is a voting method which falls under the umbrella of voting methods that are both epistemic and democratic. As explained in [1], *“Relevant Expertise Aggregation (REA) [is] a “middle-way” system for making good decisions among two or more options on issues with multiple relevant criteria. In REA the best overall choice is a function of how the options score in terms of the criteria. Each criterion is defined as a relevant domain of expertise. Scoring of options is by ranking of experts in each domain, or by mass voting based on recommendations of multiple experts.”*.

REA is based on the assumption that for each different problem, the relevance of the decision maker’s knowledge to the problem’s nature has a direct correlation to the effectiveness of the solution chosen. In a simplified example, a farmer has more relevant knowledge to decide whether the society needs to install a better plant watering system than the doctor and is therefore more likely to make a better decision. This idea is captured by giving experts a more powerful vote or by allowing them to create the voting choices in the voting agenda. How this voting method is connected to computational multi-agent systems remains to be seen.

C2. Software Engineering

21st century societies are highly complex social structures which consist of many smaller substructures that work together. To give an example of a substructure, one could think of a business organization, where a set of individuals have to communicate and act in abidance to rules in order to meet common and personal goals. Although the structure

of a business is not as complex as the structure of a whole society, it is complex enough to be extremely difficult to fully simulate in a computational environment.

A socio-technical multi-agent system (MAS) is a software engineering approach to simulate such complex problems whose inherent nature involves autonomously acting individuals. By exploiting the inherent architecture of the problem, the MAS approach is considered one of the most promising ways to tackle the complexity of such complex social structures.

The importance of simulating such complex social structures is not limited to simulating currently existing real-life structures. It extends to being able to create multi-agent architectures which solve real-life problems and that are guaranteed to perform better than their current competitors, namely standard machine learning approaches and hard-coded algorithmic approaches. It is important to note that these solutions are targeted at specific problems, problems that have some sort of a multi-agent interacting environment. Smart electrical grids, traffic management of autonomous vehicles and video game bots are only some of the applications of the socio-technical multi-agent solutions in real life.

C3. Where political science meets software engineering

The creation of the software engineering solutions mentioned above is a difficult and complex task. The range of problems that come up during the building process is wide and multi-sided. This is due to the difficulty associated with translating social structures' concepts into code. Communication, power, control, delegation of power, ethics, abstract ideas such as justice are only a few examples of such concepts.

An extensive amount of solutions to the above "translation" problem have been proposed and they draw ideas from a wide range of scientific fields, such as game theory, communication theory, information theory etc. This project focuses on the decision-making aspect of a socio-technical MAS system. More specifically, it focuses on the different methods of choosing a suitable ruleset so that the system can learn and adapt inside a dynamic environment. This is where the two fields meet. To tackle the complexity of the problem, this project draws inspiration from the political sciences and borrows the concept of REA as a means for reinforcement learning based on the assumption that since it has been shown to be an effective voting method in classical Athens it will also be in our socio-technical computational system.

Having said that, it is important to stress one more time the assumption that a democratic decision-making method performs better than a tyrannical one, the latter being a system where the decision-making process is performed by a single individual (or a group of individuals in an oligarchy) who then delegates tasks to the rest. The need for this decentralization is obvious in a social environment since humans instinctively question freedom of choice, power and equality. However, it is not obvious in a computational environment, since computers are unaware of such concepts and therefore algorithms have always been mainly based on centralized architectures where one entity has total control and is delegating tasks to the rest.

This old approach however fails to meet the demands of socio-technical multi-agent systems. This is because of their high complexity which demands a high communication bandwidth for handling all sensor information, a high computing cost to compute all agents' information and delegate tasks based on it and finally fast state transition due to dynamically changing environments. A hard-coded "tyrannical" algorithm is not likely to cope with these demands and is likely to be outperformed by the decentralized ("democratized") architecture.

D. Analysis and Design

Before beginning the analysis and design section, I would like to state that this section goes from specific to generic and a subtle forward time arrow.

D1. "Puck-Man" or "Pac-man"?

Known as "Pac-man" internationally, the famous arcade game (Figure 1) was first released as "Puck-man" in Japan due to the character's hockey puck shape. However, later it was changed to Pac-Man for its international release due to the fear of name abuse.

Basic Goal

The yellow disk is called "Pac-Man" and the rest four coloured creatures are the ghosts. Simply put, Pac-Man is the good guy and the ghosts are the bad guys. Pac-Man tries to avoid the ghosts while the ghosts try to catch it. The player controls Pac-Man and navigates it around the maze trying to eat all the yellow pills. Doing that completes the level. As far as the maze is concerned, the blue tiles are walls and there exists an invisible warping tunnel at the middle of the maze where if Pac-Man continues left to the end of the maze it gets teleported to the right end side and vice versa.

Figure 1-Pac-Man

Special features

4 big yellow dots exist in each corner of the maze. Eating one of these converts the game to power-mode, where pac-man becomes powerful enough to eat the ghosts too. The ghosts are scared, turn blue and try to avoid it. If a ghost gets eaten it returns to the central cage (starting point too for the ghosts) where it becomes normal again and cannot be eaten. The more pills and power-pills Pac-Man eats the bigger the score (bottom-left corner).

Pac-Man in this project

There exist a few differences from the real Pac-Man to this project's implementation. In this project the game has only one level, hence completing the level wins the game. Moreover, the "fruit" special feature

has not been implemented. In this feature, a fruit appears near the center of the maze once in a while which if eaten increments the score by more than a normal yellow pill. Given that the score does not matter in this implementation (only one level, hence maximum score = score of eating all pills to complete level), the “fruit” feature is unnecessary and was not implemented. Moreover, Pac-Man was controlled by an algorithm instead of a player and the movement of the ghosts by the learning algorithm, but this remains to be elaborated upon. Finally, Pac-Man normally has three lives but for the sake of fast learning I reduced it to one life.

Disclaimer

The code for the Pac-Man game was borrowed from (Fable)[3]. The code is split into functional and graphical parts. The graphical parts remained unchanged and were just used for visualisation. It is the functional parts that were amended and worked upon to implement the algorithm of the project. I do not claim to be the sole creator of the code. I worked with it instead of creating it from scratch in order to focus on the actual investigation of the report instead of focusing on the graphics and play functions of the Pac-Man game. Though, it goes without saying that I had to obtain full understanding of every piece of the code so that I could work with it and change it.

D2. Pac-Man Game Algorithm

The game is structured as a 256*256 two-dimensional (x,y) grid. Each element has a coordinate on the grid. To give an example, the red ghost’s starting position (central square) is on coordinate (105,85).

The game of Pac-Man is played using a “while” loop. Each iteration of the loop advances to the next state of the game, i.e. single movement for Pac-Man and ghosts. The state of the game, consists of all the information on the grid and the grid itself and is stored outside this while loop, both in simple variables with scope greater than that of the loop and in objects of class “ghosts”.

D3. The choice of language

The choice of programming language was an important decision which lead to a domino of subsequent decisions at the beginning of the project.

F# was chosen, a language that specializes in functional programming but supports functional, object-oriented and imperative programming too. F#, due to its functional style, allows for neat code, zero compiler bugs, minimal and easily resolvable execution bugs and a wide choice of programming paradigms to pick from. Given all the above and my past experience with the language, I concluded that F# was the right fit for my project. It is worth mentioning that the Pac-Man base code I borrowed was Object – Oriented focused. This forced me into learning OOP from scratch but it turned out really helpful in handling the state of the game (explained later). This illustrates to a certain extent the power of bringing multiple paradigms together in one language and hence the right choice of F#, although it goes without saying that it is not the only one.

D4. The graphics

The graphical user interface comes from a compiler which parses F# code and translates it into JavaScript code. This compiler is called “Fable” [3] and has been recently created. The maze, an ASCII two-dimensional array, is passed into functions which map it first into an array of bits and then into coloured images on the HTMLCanvas. In each iteration of the game-play loop, the graphics are rendered by translating the current state of the program to the corresponding image. In-depth explanation of the graphics and the functions used is outside the scope of the investigation and will not be given in this report. However, the interested reader can find information on them at Fable’s website [4].

Fable has been recently created. Therefore, documentation is poor if not virtually non-existent. This posed some serious limitations on exploiting the graphical interface and created some critical bugs that slowed down my progress significantly. In retrospect, I should have not assumed an acceptable documentation before choosing this path. However, with some creativity I managed to bypass most of the problems this created. Finally, it is worth mentioning, that it is *Webpack*, ‘‘a *static module bundler* for modern JavaScript applications. ’’ [4], that is used to handle HTML assets and create the local browser application.

D5. Pac-Man AI

Pac-Man is normally controlled by the player. Working towards an algorithm that learns through hundreds of iterations of the game, a human controlling the Pac-Man is not a time-viable option. Hence, Pac-Man is moved by an algorithm with the following pseudocode:

```
call function chooseDirection

    chooseDirections ():

    create list of all available directions
    filter out all wallDirections and gateDirections from list of availDirections
    if existsDanger then
                remove forwardDirection from list of availDirections
    else
                filter out all nonPillDirections from list of availDirections
                remove backwardsDirection from list of availDirections
    return availDirections

choose randomly a direction from list of availDirections
```

The above block of pseudocode makes sure that Pac-Man avoids walls and the gate of the central box where the ghosts reside. The *existsDanger* function avoids collisions with ghosts that are approaching. If no danger exists, Pac-Man picks a direction randomly from the directions remaining minus the directions that do not have pills and the backward direction.

This is to ensure that Pac-Man eats pills and that it does not oscillate around a fixed position respectively.

D6. Multiple Games and Speed

The learning process requires consecutive games of Pac-Man to be played. This creates the need for a second recursive layer encapsulating the basic recursive layer of the Pac-Man game. In fact, a third recursive layer was added for repeating the learning process.

Moreover, the need for speed is created too. Adjustments and re-writing of code of recursive functions applying on datasets has been made to speed up the process of playing one game of Pac-Man. Last but not least, graphics have been disabled with the exception of a score-keeping screen that keeps track of information, such as number of games that have been played and wins scored by Pac-Man. Figure 2 on the right demonstrates this.

Figure 2 - 339th iteration of game (happens to be a game Pac-Man lost)

D7. Learning

Stated again, the ultimate purpose of this project is to investigate the effectiveness of the REA voting method on a MAS learning environment. A detailed explanation of the design of the learning algorithm where REA is applied will be given here but first it is essential to define the software equivalent of REA that is used in this project which slightly differs from the political science definition given in the introduction.

Relevant Expertise Aggregation(REA) in a nutshell

A pool of agents is created inside a multiple-agent system. In this system, agents' actions are governed by a common ruleset. The ruleset contains either individual rules or combinations of rules. Each rule or combination of rules is a function that maps input to output. Input is the information the agent gets from its environment. Output corresponds to the agent's actions. Given that the system can be represented by a finite amount of states, applying the function/rule once advances to the next state of the system.

At the start of the system, the agents have no experience of existing in the environment and are therefore agnostic of strategies that can be implemented to exploit it. The agents have a common goal. In order to achieve it they need to learn from their actions in order to better understand and exploit their environment. They learn by performing actions and observing feedback coming from the environment. The way they establish their knowledge is by updating their common ruleset. To give a simple example, if the system was earth and agents humans, a rule could be "to not jump from heigh places". The knowledge to not do that has been acquired possibly since someone jumped. The negative feedback from the environment has been captured by the rule so that future generations do not repeat the mistake.

Different agents learn different things as exploring their environement. Therefore, they have different opinions on the formation of the common ruleset. The democratic way of handling these conflicts of opinion is if all agents voted based on their opinion on the next best ruleset and their votes carried equal weights. As argued in the introduction, in certain problems this method might suffer from non-epistemic and hence ineffective decisions since agents with no knowledge relevant to the problem vote with equal weight. Hence, trying to resolve this malfunction of the voting system, relevant expertise aggregation weigs more votes coming from agents with more relevant expertise. Finally, the last method, reminding us of a tyrranical regime, is when only a few experts take the matters on their hands and decide for everyone on the grounds of their expertise advantage.

Implementing REA in a software environment

The above process is captured in the following pseudocode.

```
instantiate a pool of agents
for 1..trainingIterations
        pick a subset of the pool
        initiate training environment
        vote (subset only) on common ruleset using one of the three voting
        methods
        while goal not reached, repeat:
                Receive sensor information from environment,
                Perform action abiding to common ruleset but otherwise free to
                choose
                Receive feedback from environment according to action performed
                Record feedback received binded with the input given and action
                taken
        end
        put trained subset back to pool
end
```

In the above process the *voting method* will determine how fast the agents will learn. Depending on the environment each method will perform better or worse than the others. The basic reason behind this performance variety is due to the trade-off between exploration and exploitation that the voting methods achieve. A tyrranical approach where the experts decide is likely to lead to decisions based on past experience, hence more exploitation of past knowledge. However, in this way the agents do not explore other solutions. In a fully democratic approach the opposite happens. The experts' voice gets attenuated by the masses and the ruleset depends less on previous knowledge, therefore achieving more exploration. The different approach of the methods to this trade-off creates the differences in performance since some environments require more exploration and some more exploitation. This is the theoretical expected outcome of such an algorithm according to literature [5].

Investigating REA in Pac-Man

It goes without saying that in Pac-Man the multiple agents are the ghosts and the system is the game of Pac-Man. The goal of the agents is to win the game by capturing Pac-Man. My implementation of REA is very similar to the above.

When the program starts, a pool of ghosts is created. 4 ghosts are picked randomly from the pool. A set of all possible rulesets exists in the program. Ghosts' choice of ruleset is therefore limited to voting on the user-defined set of rulesets. The ghosts vote using one of the 3 voting methods, depending on which one the programmer wants to experiment upon. The ghosts hold an opinion about each ruleset's performance and vote accordingly. A common ruleset is adopted and the game starts.

Ghosts apply the input of the environment to the ruleset function in order to get the directions which they are allowed to take according to the commonly agreed ruleset (shown below).

Then, they choose randomly from the allowed directions. After applying the ruleset for duration of the whole game, the ghosts update their record of the performance of the ruleset that was used. If the game was won (Pac-Man) lost, they improve their opinion about the specific ruleset's performance and if lost, they lessen it. The ghosts are put back into the pool and a new subset is picked. The process is repeated until the limit set by the programmer.

To give an example, one of the rulesets does not allow ghosts to go towards walls and get stuck. If the ghosts have voted on that ruleset and is currently in use, an input of *wallOnLeft* would result in the *left* direction being absent from the allowed directions. Hence the ghost will not run into the wall.

E. Implementation

E1. F# |> GUI

As previously explained, the graphical interface is handled by Fable. Given my non-existent previous experience with either Fable or graphical interfaces in general, let alone with Javascript and HTML, it was a struggle setting up the starterpack environment to get the project going.

The Pac-Man code which I borrowed from the "Fable" website was running on an online platform constructed by Fable, namely "repl" [1]. Technically, the whole project could have been programmed in that website. However, I made the choice of installing Fable, Webpack and all their dependencies (e.g. Node.js) on my computer with the ultimate goal of

[1] https://fable.io/repl/

developing the application locally in my browser. That was a time consuming process due to the poor instructions and documentation from Fable as explained in the Design section.

This decision has probably been the most important of the whole project, since it facilitated and sped up both the coding and the debugging processes of my program. The online editor provided poor code-intelligence, such as code completion or member look-up suggestions, and no debugging mode. Developing locally, on the other hand, using Visual Studio Community and my local browser's Javascript debugging tool, I was able to resolve bugs and code errors in a tenth of the time it would take me to do so without them.

Fable Graphics

Fable comes with a big graphics library which allows the programmer to define HTML and Javascripts assets in F#, which then parses and compiles. As stated before, an in-depth explanation of the graphics' modules is out of the scope of the report. However, I will explain some basic graphics' implementations.

The whole application is a "window" asset in the local browser. Customization of the window features is easily peformed if the corresponding libraries have been added to the project, namely the "Browser._____" libraries. The maze is then painted on the white *HTMLCanvasElement* using functions that treat the canvas as a two-dimensional grid, as expected.

The two classes, *Browser.Types.CanvasRenderingContext2D* and *Browser.Types.HTMLCanvasElement* contain the functions needed to create the and paint the canvas. Subfunctions are defined within the scope of the Pacman module to to handle the exact way of painting the maze according to the color and coordinate requirements of the Pacman maze.

E2. Preparing for Learning – Part 1:Pac-Man AI

Pac-Man

In the beginning I considered AI algorithms with forward planning. However, due to the underlying, rather subtle, complexities of the game, I dropped this way of thinking. The root of all these complexities is the maze itself. To clarify the above statement one only needs to consider that in many points of the maze, travelling to a new point which is generally towards one direction, say south, might involve travelling to several other directions first. A human mind can easily plan that far ahead especially in such simple visual problems. However, calculating 50 states ahead using any AI algorithm is very computationally expensive.

Figure 3 illustrates this concept where the pacman tries to go 15 pixels south but is forced by a wall to first go right, then left and then again left, if it takes the green route. The red route is even more complicated. Yellow pills are 8 game states far away from each other, so Pac-Man in this scenario would need 72 states forward planning to manage to reach the 15 pixels south coordinate. To conclude, even if in a simple game of Pac-Man this computational overhead cost can be tolerated, in learning where hundreds of games need to be played fast, the cost is prohibitive and I therefore dropped the perfect gameplay idea.

Figure 3 - Pacman routes

Instead, I leaned towards a simpler algorithm, based on random movement and avoiding ghost collisions.

Randomness

A Pac-Man picking the next direction completely random oscillates around its position as expected. Therefore complete randomness is non-effective. To resolve the backwards direction is removed. This immediately makes the Pac-Man movement smooth.

Ghost Collisions

Although the Pac-Man moves smoothly, it still loses the game due to collisions with ghosts. To account for such scenarios, I created a function that operates as a radar. It checks in a radius of 30 pixels around the Pac-Man for ghosts. If a ghost enters this dangerZone the algorithm checks if the ghost's trajectory will cut Pac-Man's trajectory, i.e. if they will collide. If this is true, Pac-Man immediately picks any other direction but forward to avoid the collision. Figures 4 and 5 illustrate this concept.

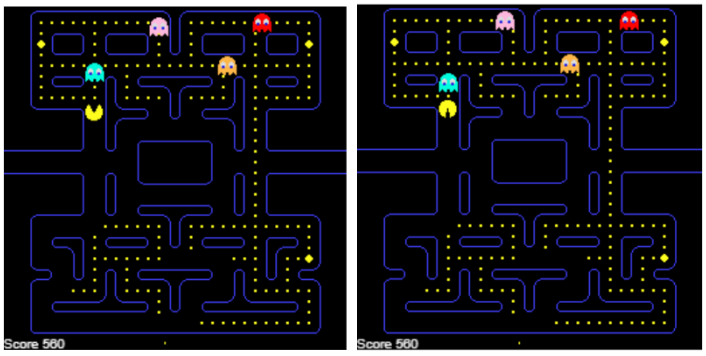


Figure 4&5 - Collision Detection and Avoidance

Prefer Pills

Due to the randomness in Pac-Man's motion, I observed that Pac-Man multiple times failed to choose a direction with more pills, hence direction closer to winning, and chose instead randomly a direction already explored. To account for this, I also added a preference for directions with pills. The result was a shorter winning time.

Gate

The central box, where the ghosts come out from in the start of the game, has a gate on the top, which although appearing as a wall is traversable. Pac-Man when entering the box gets trapped and loses. Hence, I added a block of code to never enter the gate. This improved the winning ratio too.

Results

The Pac-Man wins 1 out of 4 times on average. When it loses it loses 60% of the time because ghosts surround it and 15% of the times because the collision avoidance algorithm fails in some scenarios. Figures 5 and 6 demonstrate these two cases.

Figure 7 - Pac-Man loses to ambush

Figure 6 - Collision Avoidance fails

Result Discussion

The focus of this investigation lies on the learning algorithm of the ghosts. Hence, given time constraints, I made the choice to not improve the Pac-Man algorithm any further. With this Pac-Man algorithm the ghosts have a 75% winning ratio and most of this comes from all going towards the Pac-Man from different directions. This will serve as a benchmark for the learning algorithms. In future work I would eliminate the disturbing losses due to the collision avoidance fails.

E3. Preparing for Learning – Part 2:Multiple Games and Speed

Triple Layer Recursion

The learning is performed using a triple layer recursion, as shown in the code block below.

```
let rec game ()
    let rec start ()
            if gamesPlayed > n then return ()
            else
            while (not pillsZero) && (not energyZero) do
                    logic()
                    return gameWon?
            start()
    start()
window.addEventListener("click", fun _ -> game () )
```

Before the *game()* function begines, code not shown above initializes the *HTMLCanvas* and renders the background. It then waits for the user's "click", using an *EventListener*. When the user clicks, the execution enters the *game()* function whose last element is calling the *start()* function.

In reality the start function calls the *playLevel* () function which encapsulates the loop shown above and also contains several helping functions. However, for simplicity it has been omitted in the above code

block in order to capture only the essence. I will continue analyzing the code block ignoring the small difference to the real code.

The *start()* function consists of a *while* loop which runs one game of Pac-Man using the *logic()* function. When the game ends, either because *pillsZero = true* (Pac-Man ate all pills) or because *energyZero=true* (Pac-Man has zero lives), the result of the game is returned as a Boolean value to *start()*. Then *start()* is called again to continue with the next game of Pac-Man. When the number of games played exceeds the user's desired value (*n* here), *start()* returns unit to *game()* and the code waits again for a "click" to initiate the next learning session.

Exceeding Maximum Call Stack

My initial approach to designing the tripled-layered recursion did not involve a *while* loop but rather the more natural (functional-oriented) triple recursive function desing shown below.

```
let rec game ()
    let rec start ()
            if gamesPlayed > n
            then return ()
            else
            let rec update()
                    if (not pillsZero) && (not energyZero) then
                    logic()
                    update()

                    else
                    return gameWon?
            start()
    start()
window.addEventListener("click", fun _ -> game () )
```

Again, in the code block above I am showing a similar version to avoid superfluous information. This way of implementation was perfectly fine for about up to 50 games of Pac-Man. After that threshold the program became non-responsive and exceptions were thrown. The exception pointed out to a "Maximum Call Stack violation". The update() function was recursively been called continuously until a game of Pac-Man was terminated. Hence, in long games of Pac-Man the maximum call stack size was exceeded and the game chrashed. I corrected this bug using the *while* loop wich returns the call after each iteration.

The difficult part of spotting this bug was that using common sense one would expect the game to crash also at the first game, if the game was long enough. However, this was never observed. I later realized that the game could indeed crash also when executing 1 game of Pac-Man but it was much less probable (might have even happened once out of 100 and I probably disregarded as a browser crash error). However, when playing more games the probability of finding one game that takes too long increased and also the call stack increased by 50 because of the outside

layer function (start()) which is recursive too. Hence, the bug always existed but it only became visible due to the law of large numbers in a sense.

E4. Data Structures

Ghosts

The ghost data structure is a class, shown on the right in collapsed function mode. When an object of type *Ghost* is created, the image of the ghost, the starting position on the maze, the starting velocity and a unique identification number are passed to the constructor.

The member functions are fairly simple and are used to modify the encapsulated private mutable variables of the class. Further underlining this idea, the use of the class makes sure that ghost variables are not mutated by any other function other than the member functions of the class. Bugs due to mistaken mutation of ghost variables were considered very likely due to the whole program revolving around the coordinates of the ghosts and the pacman. Hence, the use of the class was deemed the appropreate measure to obtain a peace of mind.

```
/// Ghost Class...
type Ghost(image: HTMLImageElement,x,y,v,id) =
  let mutable x' = x
  let mutable y' = y
  let mutable v' = v
  let mutable id' = id
  let mutable games' =0
  ///(won,total)
  let mutable performance' =...
  member val Image = image
  member val IsReturning = false with get, set
  member __.X = x'
  member __.Y = y'
  member __.V = v'
  member __.ID = id'
  member __.Games = games'
  member __.Perf = performance'
  /// Move back to initial location
  member __.Reset() =...
  /// Move in the current direction
  member __.Move(v) =...
  member __.Identify(id) =...
  member __.GameCompleted() =...
  member __.UpdPerf(victorious,ruleIndex) =...
  //return percentage of games won
  member __.Opinion(ruleIndex) = ...
```

Ilustración 1- Ghost Class

Function *This.Reset()* is used to move the ghosts to their initial location when they are eaten by the Pacman. Function *Move(v)* is self-explanatory and it takes the desired direction to move and applies it to the ghost, mutating the *This.X* and *This.Y* variables representing the position fo the ghost.

Member function *GameCompleted()* increments the *Games* field by one after the termination of each game. This is used during learning to quantify the expertise of each agent (ghost). The more games played, the more voting power using the REA algorithm. Moreover, the *UpdPerf(victorious,ruleIndex)* function updates the ghost's perceived performance of a rule. In other words, if the ghost received positive feedback from the environment using rule 'x', at the end of the game it records it using this function to update its field *This.Perf*. Last but not least, the function *Opinion(ruleIndex)* returns the opinion of the ghost about the performance of a certain rule (pointed out by *ruleIndex*).

Pac-Man

The pacman state is nothing more than a simple (x,y) position and a current velocity, captured by mutable variables of wide scope.

Ruleset, Pool of Ghosts and Graphics

The ruleset adopted during learning, the pool of ghosts to choose from in each iteration and the graphics (e.g. maze) were all handled using Arrays. F# is famous for its Lists, which I personally prefer, but the program I started with was using Arrays and I therefore chose to stick to it to avoid confusion. Arrays were used for small data structures that stored temporary values. In this way iterative functions could easily be implemented upon them, a domain where F# provides a huge freedom of

```
let ghosts =
      [|1..4|]
      |> Array.map (fun _ -> let randomNum = System.Random().NextDouble()
                             let j = randomNum * float ghostPool.Length
                             let i = int (floor j)
                             let g = ghostPool.[i]
                             ghostPool <-
                                  ghostPool
                                  |> Array.filter(fun a -> (g.ID<>a.ID))
                             g

                  )
```

Ilustración 2

choice. Indeed, it saved me in a lot of instances from laborious and verbose implementations of simple data structure manipulation. To give an example, on the code above, *Array.map* and *Array.filter* are used to pick randomly 4 ghosts from the pool of ghosts. The filter function is used to remove the picked ghost from the remaing pool of ghosts. Clearly, the F# built-in functions facilitate such kind of operations on Arrays/Lists.

Capturing the State

To conclude, all the above data structures were used to capture the state. The choice was careful so that the state would be well protected and easily manipulated by functions.

E5. Learning

Basic Algorithm

The basics of the learning algorithm have been explained in the design section. In this section I will focus on the technical implementation.

Ghost Pool Formation

The creation of the pool of ghosts is simply done by instantiating objects of the *Ghost* class and appending an array with them. This can be shown on the right. The image, initial position and the initial velocity are all given to the class constructor, which can be seen in the last line of the code block. The constructor returns the *Ghost* object which is then added to the array using the *Array.map* function.

```
let createGhosts =
  [| Images.redd, (16, 11), (1,0)
     Images.cyand, (14, 15), (1,0)
     Images.pinkd, (16, 13), (0,-1)
     Images.oranged, (17, 13), (-1,0)
     Images.redd, (15, 11), (1,0)
     Images.cyand, (17, 15), (1,0)
     Images.pinkd, (18, 13), (0,-1)
     Images.oranged, (15, 12), (-1,0)
     Images.redd, (14, 11), (1,0)
     Images.cyand, (18, 15), (1,0)
     Images.pinkd, (16, 12), (0,-1)
     Images.oranged, (17, 14), (-1,0)
     Images.redd, (14, 13), (1,0)
     Images.cyand, (14, 14), (1,0)
     Images.pinkd, (16, 15), (0,-1)
     Images.oranged, (18, 14), (-1,0)|]
  |> Array.map (fun (data,(x,y),v) ->
       Ghost(Images.createImage data, (x*8)-7, (y*8)-3, v, 0) )
```

Ilustración 3

Ruleset Array

The set of rulesets creation is also simply an array creation with rulesets generated by the author. This can be seen below where the specific function performed by each ruleset is commented on the code block.

```
let allRules : ((int * int) [] -> Ghost -> (int * int) []) [] =
    [|
  rule1 //No rules, random motion
  rule2 //Walls directions removed
  rule3 //Backwards directions removed
  rule4 //left and right only
  rule5 //Rule 2 and Rule 3 combined
    |]
allRules
```

Ilustración 4

Picking subset from Ghost Pool

Picking the subset from the ghost pool has been explained in the E4 -> “Ruleset, Pool of Ghosts and Graphics” section. To reiterate what was stated, a ghost is picked randomly from the pool and not replaced until the end of the current game using the built-in functions of the *Array* module.

Voting Method

1) Simple Vote Aggregation

The block of code that implements the equal-weights vote aggregation method is shown below.

```
let formOpinions =
      [0..ruleset.Length-1]
      |> List.toArray
      // SIMPLE VOTE AGGREGATION
      |> Array.map(fun index -> let mutable sum = 0;
                                ghosts
                                |> Array.iter(fun ghost ->
                                                    sum <- sum + ghost.Opinion(index))
                                |>ignore
                                opinions.[index] <- (sum/ghosts.Length)
                                let opin = opinions.[index]
                                ((ruleset.[index],index),opin)
```

Ilustración 5

It goes through the whole ruleset and gets all ghost's opinions on each rule. It sums up the opinions and then divides by the number of ghosts to obtain the average opinion for each rule. The opinions are later used to pick the rule with the highest performance using a simple *Array.max* function.

2) Experts-Only Vote

The experts-only vote method is similar to the equal-vote except that now the array of ghosts is sorted according to their experience, i.e. how many games they have played. Then it is reversed so that the maximum is first and then the first element is picked, i.e. the ghost with the most expertise. Finally, this ghost's opinions are adopted and all other ghosts' opinions are irrelevant. Just to reiterate, the *opinions* is the accumulated feedback of the ghost for each rule from past games.

```
let formOpinions =
      [0..ruleset.Length-1]
      |> List.toArray
      //EXPERTS ONLY VOTE
      |> Array.map(fun index -> let mutable op' = 0;
                                let expertGhost =
                                  ghosts
                                  |> Array.sortBy (fun ghost -> ghost.Games)
                                  |> Array.rev
                                opinions.[index] <- expertGhost.[0].Opinion(index)
                                let opin = opinions.[index]
                                ((ruleset.[index],index),opin)
```

Ilustración 6

3) Relevant Expertise Aggregation

Again, the code is similar to the other methods but in REA the votes are weighted by the times the ghosts' have played the game, i.e. by their expertise.

```
let formOpinions =
      [0..ruleset.Length-1]
      |> List.toArray
      //RELEVANT EXPERTISE AGGREGATION
      |> Array.map(fun index -> let mutable sum = 0;
                                ghosts
                                |> Array.iter(fun ghost ->
                                                    sum <- sum + ghost.Opinion(index)*ghost.Games)
                                |>ignore
                                opinions.[index] <- (sum/ghosts.Length)
                                let opin = opinions.[index]
                                ((ruleset.[index],index),opin)
```

Ilustración 7

Capturing Knowledge and Forming Opinion

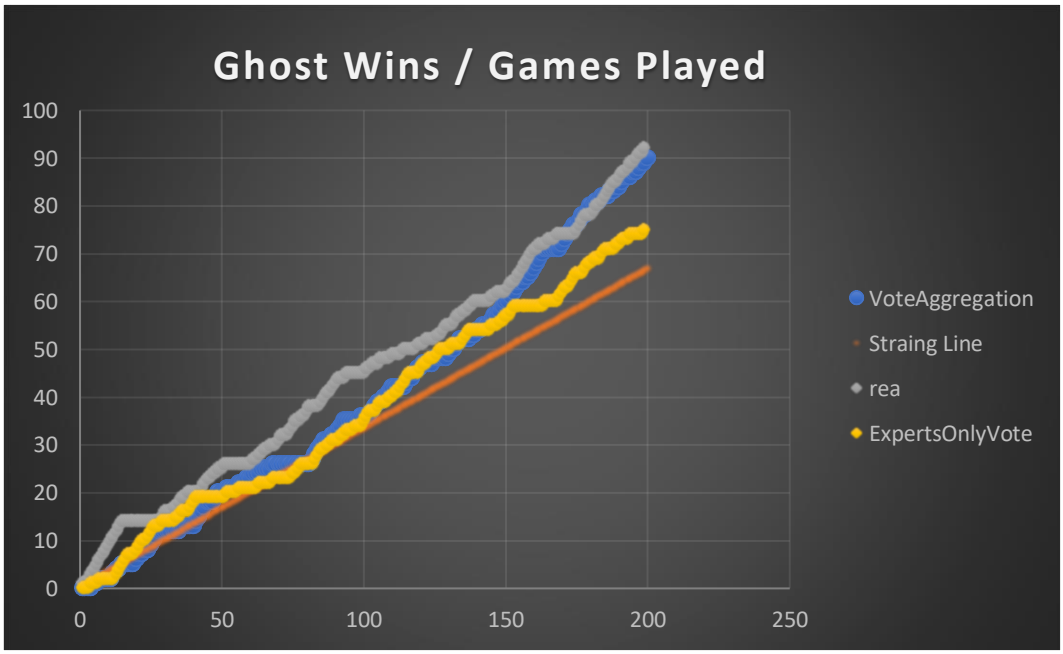
At the end of the game, if ghosts won is known. Hence, ghosts update their perception/opinions about the ruleset they used accordingly. They do this using the member function *This.UpdPerf(victorious,ruleIndex)*. They also record that they gained more expertise by noting down that they played one more game with the member function *This.GameCompleted().* This is shown below.

```
ghosts
|> Array.iter (fun ghost ->
                              allGhosts
                              |> Array.map(fun g ->
                                              if ghost.ID = g.ID
                                                 then
                                                     g.UpdPerf(won,rIndex)|>ignore
                                                     g.GameCompleted()|>ignore
                                                     g
                                              else
                                                     g.GameCompleted()|>ignore
                                                     g
                                           )
                              |>ignore
              )
```

Ilustración 8

F. Results

The result of my investigation can be summarized in the following graph.
In the graph I have plotted the number of wins of the ghosts against the Pac-Man at each iteration. In other words, an increase in the slope of the graph corresponds to an improvement in the ghost algorithm (**learning = change in slope**). The grey line is the result of applying REA while the yellow is for experts-only voting and the blue is for simple vote aggregation. I also plotted a simple straight line to facilitate the visualization of the increase of the slope of the learning algorithms compared to a constant slope line.



G. Discussion and Evaluation

To my great disappointment the results show a small difference between the learning performance of the 3 algorithms. The increase in distance from the constant slope straight line is slow and constant. This implies a small learning progress after the first few iterations where most of the learning occurs.

REA

REA begins by learning very fast, giving hope and anticipation, but it then slows down only to get equal to the vote aggregation method. The traid-off between exploration and exploitation seems to be effective in the short term, where learning is fast, but as long as most of the learning has occurred it slows down.

Equal-Weight Vote Aggregation

The equal-weight vote aggregation method starts slower than all 3 methods but through exploration manages to surpass the Experts-only

method and finally catch up with REA (although in the last iterations REA has a slightly more promising slope).

Experts-Only Vote

This method fluctuates a lot, probably because of different experts taking the lead and failing after some iterations. The long-term performance is the worst of all three as the exploration is minimal and the learning hits an early cap.

Pac-Man

Unfortunately, the results obtained are not enough to declare the correct trade-off(exploitation vs exploration) for ghosts to win in Pac-Man. However, the vague conclusion can be drawn that certainly both are needed.

H. Conclusion and Future Work

Concerning Results

The results show small differences between the three algorithms. The trade-off between the underlying ideas of exploration and exploitation is barely visible but still worth mentioning.

Having gone through the whole process of creating this program it is my educated guess and firm belief that:

> The algorithm is severly constrained by the pure fact that the possible rulesets were generated by the author and were therefore biased. It is my educated guess that this resulted in rapid learning in the first several iterations (as shown in graph) which then quickly capped due to the inherent cap in my ruleset. The inherent cap is created by the fact that in my generated ruleset, one rule produces results that completely outperform the others. Hence, if picked once it caps the learning.

Time limitations prohibit me from changing the rulesets. However, in future works the possible rulesets should be changed to either ghost generated rulesets using randomness or to user-generated ruleset with wide variety and minimized bias. In that case, it is my educated guess that the rest of the environment and the Pac-Man game in general can foster the production of valuable results.

Another very important aspect of future work is to focus on **relevance.** Having read most of Ober's paper [1] I realized that half of the success of the REA method was relying on relevant expertise, hence not quantity of expertise but quality. My algorithm does not distinguish between the two. My initial plan was to take the road of investigating how relevance of the expertise affects the performance of the algorithm but due to time constraints I dropped that path. Hence, for future work that path should definitely be futher explored.

I. User Guide

An important part of my project has been the visualization aspect of the algorithm. For this reason, I have kept the process of running my code as simple as possible. Simple instructions and code can be found on my GitHub profile **(mchatzis)** under the *Relevant-Expertise-Aggregation-A-simple-reinforcement-learning-approach.* It is actually as simple as copy-pasting code to an online compiler.

A. Bibliography

[1] J. Ober, «Relevant Expertise Aggregation: An Aristotelian middle way for epistemic democracy,» Stanford University, 2012.

[2] R. R. a. D. Keyt, Aristotle Politics, Books III and IV, Oxford and New York: Oxford University Press, 1995.

[3] R. C. E. H. J. O. Jeremy Pitt, «Relevant Expertise Aggregation for Policy Selection in Collective Adaptive Systems».