Faculty of Sciences
Department of Computer Science

UNIVERSITE LIBRE DE BRUXELLES,
UNIVERSITE D'EUROPE

Dynamic Task Allocation in a Turn-based Strategy Game

Gilles Schtickzelle

Promoter: Ann Nowe

Advisor: Peter Vrancx

ACADÉMIE UNIVERSITAIRE WALLONIE-BRUXELLES

Thesis submitted in partial fulfillment of the requirements for the degree of Master in Computer Science

Academic year 2011-2012

“We can only see a short distance ahead, but we can see plenty there that needs to be done.”

Alan Turing, 1950

Abstract

Artificial Intelligence techniques are many and some of the more inventive approaches take example from what can be observed in nature. In this work we demonstrate how to exploit the division of labor mechanisms observed in insect colonies to implement an artificial player capable of managing resources in a computer strategy game. Using the response-threshold model developed to explain dynamic task allocation in ants and wasps colonies, we build a player that can play a rather substantial subset of the game at near human level.

Acknowledgments

A master's thesis is never the work of one, but the result of the combined years of advises, mentoring and lessons we received during the course of our studies. This work is no exception, and I would like to express my thanks to everyone, professors, fellow students, family and friends, who made this possible. In particular:

Many thanks to my thesis advisor Dr. Peter Vrancx for his unending support. This work would never have taken shape without his thoughtful insights, correction suggestions and his perseverance in putting me back on the right track whenever I lost sight of the end result.

Many thanks also to my thesis advisor Dr. Ann Nowe for giving me the opportunity to work on a subject I am passionate about and leaving me the freedom to explore unexpected and unconventional path.

Thanks to Dr. Doina Precup, my professor of artificial intelligence during my stay at McGill University, who through her introductory course made me realize the potential of this branch of computer science.

Thanks to Dr. Mauro Birattari and Arne Brutschy at the IRIDIA laboratory for their insightful advises in swarm intelligence.

Many thanks also to all the ULB faculty and staff who helped me not only get through those five years of studies but also imparted me with some of their knowledge and made me that much smarter.

Thanks to my fellow students for their help, support and advises, in particular Joffrey Schmitz and Mikael Lenaertz for their outstanding work during our group assignments, and to François Santy for ever kindly passing out his notes whenever I needed guidance with a subject.

Finally, many thanks to my parents for supporting me in my decision to go back to University at an age where most people are happy to finally be done with it.

Contents

Chapter 1. Introduction

Starting with "The Turk", a fake chess-playing machine built by Wolfgang Von Kempelen in 1770, artificial players have long captivated the public's attention. Even before the invention of digital computers, American writer Edgar Allan Poe, comparing "The Turk" with computer ancestor, Babbage's *Difference Engine*, said the chess-player would be *"beyond all comparison, the most wonderful of the inventions of mankind"* (Poe, 1836), should it be a *pure machine* (which it was not, in fact). Indeed, there is something about being able to go toe-to-toe in a game against a human opponent that capture a man's attention beyond what the pure computing power a computer can.

In this work, we use a strategy game based on Sid Meier's "Colonization" (1994) to demonstrate how one can use different biology inspired techniques to make an efficient artificial player. In Colonization (or rather in FreeCol, an open-source clone of the original game), a player is tasked with replaying the development of the new World colonies, from the initial discovery of the Americas in 1492 until the declaration of independence (which may or may not be 1776 depending on the skill of the player). At each turn, the player has to choose where to allocate his units to grow and defend his colonies optimally. We show that an artificial player using the response threshold mechanism set forth by Wilson (Wilson, 1984) can prove a simple and elegant way to allocate units to the different tasks. We also show that this simple model can be augmented with planning-like functionalities and improves on the general result.

1.1 The computer game framework

Not to be mistaken with Von Neumann's Game Theory, computer games (so-called video games) can constitute a very interesting and practical framework for testing Artificial Intelligence concepts, as IBM showed again last year with their Jeopardy-playing Watson computer, now the center piece of IBM "Smarter Planet" initiative (IBM, 2012).

As a research platform, computer strategy games present several interesting qualities. First, they model problems that are intended to be challenging but within reach of a human mind. And while solving industrial-level complexity problems is by no means uninteresting, there is something to be gained in developing an artificial intelligence that can be more similar to human-level intelligence. Second, strategy games are designed to be more a challenge to human than say, planning a trip to the convenience store. Indeed, they propose situations with dozens or even hundreds of variables, and branching factors high enough to prevent most players from considering more than a fraction of the possible actions. In order to solve such a problem, a human player will apprehend the situation through different abstraction levels; which is fundamental to human intelligence. Developing a game AI can thus bring compelling insight into the workings of a human brain.

But also on a more basic level, one has to acknowledge that the video game industry is now a $66 billion industry worldwide and growing (Takahashi, 2011). And while there has been great focus to improve the graphical aspect of video games, artificial intelligence has been limited to mostly scripted rule sets. There have been exceptions here and there but it is clear that the industry could benefit a lot from using more state of the art techniques that have been developed by academia. Although games nowadays often circumvent the need for an intelligent artificial player by proposing human players to play against or with each other, the lack of a competent, non-cheating, always available artificial game opponent is still a major source of frustration.

1.2 The turn-based strategy game platform: FreeCol

Published in 1991 by MicroProse, Sid Meier's Civilization was the game that made turn-based strategy game popular and ranked among the top PC games of all time ("The Top 25 PC Games of All Time - PC Feature at IGN," n.d.). In that game, the player was asked to "build an empire to stand the test of time". The player started with a single unit, a settler, on a randomly generated map. The player would take turns with the AI opponents to explore his surrounding and choose an appropriate location to build his first city. From there, he would grow his burgeoning civilization by building more cities, developing both economically and militarily, researching new technologies and fighting against other civilization to eventually conquer the whole world.

The success of the game brought along many sequels and clones that would put a different spin on the genre. Sid Meier's Colonization was one of them. Published in 1994, also by MicroProse, Colonization borrowed the same basic mechanics than Civilization. Here, the player would start in 1492 with a ship and a couple colonists, sailing to the new world. He would then take turn with other colonial and Native American nations and vie for the control of the newly discovered territories, by expanding his colonies, building military units to defend himself of attack his neighbor, while ultimately trying to declare independence from his European motherland. A major part of the game is to decide what individual units should do. They can assume different roles in a colony, like growing food in the surrounding fields, chopping wood in the forest, or working in the different colony buildings to transform raw materials into more valuable processed goods. While the gameplay involves other aspects, this is the only one that is mandatory to win the game, so it is an ideal subpart of the game to build an AI player for.

In 2002, a small team of fans decided to create an open source clone of the original Colonization, and release it under the GPL. While officially still in beta (now at version 0.10.5), FreeCol (www.freeCol.org) is a fully functional (if not bug-free), nearly identical client-server implementation of Colonization. It is written in Java, multi-threaded; and with more than 650 classes and over 75 000 lines of code, it is a fairly complex piece of software.

Nonetheless, the freedom of the GPL license and the client/server model makes it appropriate for developing a stand-alone AI client, and the turn-based structure of the game allows focusing on concepts rather than on performance, since each player can take an arbitrarily long time to evaluate their options. The interested reader can find more detailed information about the game mechanics in Chapter 4.

Figure 1 Sid Meier's Colonization (1994). Colony management screen. From http://www.mobygames.com/game/windows/sid-meiers-colonization/screenshots/gameShotId,53861/

1.3 Bio-inspired Artificial Intelligence

Traditionally, artificial intelligence has tried to emulate the capabilities of the human brain. As such, one can argue that all artificial intelligence research takes its inspiration in biology: us. However, there have been some very interesting developments in a wider range of biological structure. Swarm intelligence, which is the foundation of this work, attempts to emulate the problem solving capabilities that can emerge from a group of simple agents, such as social insects. Algorithms based on ant colonies for example perform at state-of-the-art level for some important combinatorial optimization problem (Blum, 2004).

1.3.1 The Response Threshold Division of Labor Model

The groundwork for the response threshold model was laid out in 1984 by American biologist Edward O. Wilson, a Harvard entomology professor and two times Pulitzer-prize winner. Observing the tasks allocation of the different castes of ants, called *minors* and *majors*, in a colony, he noticed that when the ratio of minors/majors drops below a certain level, majors would

adapt and take it upon themselves to switch from their initial task (defend the colony) and fill in for the missing minors (taking care of the larvae) (Wilson, 1984). Since individual ants do not possess any global representation of the colony's need, this level of plasticity was unexpected. Bonabeau (Bonabeau, Theraulaz, & Deneubourg, 1996) then formalized the process as such: The colony, by way of its individuals, emit a certain number of pheromone stimuli. Larvae emit pheromone to signal their hunger, sentries emit pheromones to signal intruders, etc. Each caste has a set of thresholds associated to each stimulus. When a stimulus becomes greater than an individual ant's threshold, that ant will answer to that stimulus with great probability, and this will in turn lower the stimulus. The castes specialization can be explained by the fact that each caste has a lower response threshold to the stimulus they were intended to answer to. If not enough individuals work at reducing a certain stimulus, its value will grow quickly until it is high enough for a non-specialized individual to answer to it, and this is how the plasticity of task allocation in an ant colony is explained.

It was not lost on this author that both the ant social organization and the early pioneer settlements in the New World are called 'Colonies'. This was one of the main inspirations for adapting the response threshold model to this artificial FreeCol player.

1.3.2 Hierarchical Learning Architectures

In human and machines, intelligence is a matter or matching a set of input with a correct output. For any given situation (speech, vision, situational awareness), the input can be constituted by a great many number of variables. To extract the correct output from such a large input set, scientists ended up trying to train their intelligent systems rather than try to describe all the rules that could match a set of input variables to the correct output.

In the same time, and starting in the early 60's, it was argued that hierarchy is (one of) the central structure of complex systems (Simon, 1962). Indeed, if one looks at social or business organizations, biological and physical entities, or even simply natural language, hierarchy is everywhere.

Still, for many years artificial intelligence focused on developing flat or limited-layer techniques. In many cases, a depth of two (one input layer, one intermediate calculation layer, and one output layer) is enough to represent any function with a given target accuracy. This comes however at the price of having an intermediate layer that grows exponentially with the size of the input for complex functions (what Richard Bellman, the dynamic programming pioneer, called 'the curse of dimensionality'(Bellman, 1957)). This means the training of such a system would be exponentially slow. However, early attempts to train deep (i.e. more than a few layers) architectures failed. For example, before 2006, it was generally acknowledged that training a deep supervised feedforward neural network would yield worse training and testing error than a shallow one (with 1 or 2 hidden layers).

A series of papers changed that in 2006, starting with (Hinton, Osindero, & Teh, 2006), (Ranzato, Poultney, Chopra, & Lecun, 2007) and (Bengio, Lamblin, Popovici, & Larochelle, 2007). All three papers followed the same key principles (Bengio, 2010):

- Unsupervised learning of representations is used to (pre-)train each layer.
- Unsupervised training of one layer at a time, on top of the previously trained ones. The representation learned at each level is the input for the next layer.
- Use supervised training to fine-tune all the layers

While we will certainly not claim that our work constitutes a deep learning architecture, those three principles were the inspiration that guided the addition of an additional planning layer on top of the response threshold mechanism, as we will detail in 5.3

1.4 Contribution of this work

There has been some success in adapting the response-threshold model to real life industrial problem, where the difficulty lies in scheduling a series tasks in a highly stochastic environment. We show here that we can also use the response-threshold model in very practical applications (computer games) where the complexity comes not from the randomness of the input but from the intricate interactions between the different game mechanics. We present an AI player that can perform at near human level on average and would be perfectly suitable to be included in commercial games.

1.5 Glossary of terms

A certain number of abbreviations, technical terms, or commercial products are used throughout this thesis. Here is a brief explanation of the most recurring ones, in alphabetical order.

AI: Short for Artificial Intelligence.

ANN: Short for artificial neural network. An AI technique where a network of artificial neurons is trained to produce the correct output from a given input.

Colonization: A turn-based strategy game where the player replays the conquest of America.

Deep learning: A machine learning approach where several hierarchical layers are built on top of one another.

DTM: Dynamic Threshold Model. An evolution of the fixed threshold model (see FTM) where the thresholds are modified over time to reflect the learning of the task.

ϵ-greedy Q-learner: A reinforcement learning algorithm where the next action for each state is chosen at random ϵ % of the time and as the maximum value action the rest of the time.

Emergent behavior: A behavior that is a result arising from the action of many simple agents.

Fitness: A score representing the performance of an individual in a GA.

FreeCol: An open source clone of the Colonization game.

FTM: Fixed Threshold Model. A simple response threshold model where the thresholds for an individual do not vary over time.

GA: Genetic Algorithm. A meta-heuristic mimicking the 'survival of the fittest' evolutionary process to find an optimal solution to a given problem with a large search space.

Genome: A representation of an individual's characteristics to be used by a GA.

Minor: The basic worker in a colony of *Pheidole* ant, tasked with feeding the larvae

Major: The larger worker, also called 'soldier', in a colony *Pheidole* ant, tasked with breaking up larger food item and defending the colony.

Market-based approach: A multi-agent task allocation algorithm where agents bid for tasks according to their capacity to perform them efficiently. The task is then allocated to the higher bidder.

NPC: Non-playing character. A character in a video game that cannot be controlled by a human player.

Response Threshold: A dynamic task allocation mechanism inspired by insect colony organization. The center piece of the algorithm described in this work.

RTS: Real-time strategy game. A computer game genre where players have to make strategic decision in real time, by opposition with TBS.

SoL percentage : Sons of liberty percentage. A measure of the freedom of a colony in FreeCol. The SoL percentage has to be at least 50 % to declare independence and win the game.

Stigmergy: An indirect coordination mechanism used by social insects, where individuals communicate by changing their environment (leaving pheromones for example) rather than by direct communication.

TBS: Turn-based strategy game. A computer game genre using the turn-based game concept (see turn-based game)

Tile: a cell in the grid of a strategy game map

Turn-based game: A game where each player takes turn to plan and enact their actions, without any time limitation constraints on the length of a turn.

1.6 Structure of this thesis

In chapter 2, we will detail the most up to date swarm intelligence algorithms that were the inspiration for this work. In particular, chapter 2.1 to 2.5 focus on the response threshold applications for the dynamic task allocation problem, while chapter 2.6 will briefly expose the a hierarchical approach that has been tried with this model. In chapter 3 will give an overview of

the AI in video games and show what has been done so far to implement a more advanced AI in that field.

In chapter 4 we will give a more in-depth overview of our framework game, FreeCol, and highlight the important game mechanics. In particular, 4.2 summarizes the different goals we have for our AI in the scope of the game.

Chapter 5 follows up with the implementation details of our algorithm and the different variations tested. Then Chapter 6 will give an in-depth analysis of our results.

Finally, we will give our conclusion in Chapter 7, including ideas for further development.

Chapter 2. Dynamic Task Allocation

Insect colonies, like ants, bees or wasps, have no centralized command. The queen inside the colony is only responsible for breeding and does not possess any capabilities to order the other colony inhabitants around, unlike the medieval royalties. Furthermore, a single individual is a very simple and primal agent so it seems nothing short of a miracle to see what the colony has a whole is capable of. Task allocation of the colony's inhabitants is both efficient and robust despite the fact that individuals do not have a language or a means to communicate directly with each other. Instead, they lay down pheromones that can be detected by other individuals. This process of modifying the environment to provide information is called stigmergy. Through stigmergy, ants, bees or wasps can communicate stimuli to one another and take the appropriate action. The beauty of it all is that an individual ant has no conscience of what it is doing but together, their collective actions result in emergent behavior, such as the reallocation of *majors* when the level of *minors* in the colony drops. It is to explain Wilson's findings about such dynamic task allocation (Wilson, 1984) that the response threshold model was proposed, first with the Fixed Threshold Model (FTM) in (Bonabeau et al., 1996).

2.1 FTM (Bonabeau et al., 1996)

The FTM provides a simple dynamic task allocation mechanism that partially matches what Wilson saw in his experiments (see 1.3.1). Under this model, each task is associated with a stimulus, each individual has a response threshold for each task, and they will engage in a task when the associated stimulus exceeds the threshold.

More formally, for a system with only one task and two castes:

Let X be the state of the individual ($X = 0$ corresponds to inactivity, $X = 1$ corresponds to performing the task). If S is the stimuli associated with the task, and θ_i is the threshold of an individual of caste i ($i = 1, 2$), an inactive individual belonging to caste i will start performing the task with a probability P_i per time unit:

$$P_i(X = 0 \rightarrow X = 1) = \frac{S^2}{S^2 + \theta^2} \qquad (1)$$

Thus, the probability that an individual will perform a task depends on the stimulus and the threshold. It will have exactly ½ chance of responding to it if the threshold and the task are equal, while the square will ensure that the probability will increase rapidly once the threshold is reached. Furthermore, we can explain the division of labor observed between minors and majors by postulating that the majors possess a higher threshold to that task.

On the other hand, an active individual will become inactive with a probability p (assumed to be fixed and equal for both castes) per unit of time:

$$P_i(X = 1 \rightarrow X = 0) = p \qquad (2)$$

This means an individual will on average spend $1/\mathrm{p}$ time units performing the task and could keep performing a task even after it is no longer necessary. While this behavior is not particularly efficient, it was observed in several species of ants and (Bonabeau et al., 1996) did not find it necessary to come up with a more elaborate model for the return to inactivity.

Finally, the variations in stimulus intensity were set to depend both from the task performance α, which reduces the stimulus intensity by meeting the demand for action, and from a natural increase of the demand δ, irrespective of whether or not the task is performed. The resulting equation for stimulus intensity S is therefore (in discrete time):

$$S(t+1) = S(t) + \delta - \frac{\alpha}{N}(N_1 + N_2) \qquad (3)$$

where N is the total number of potentially active individuals in the colony, and N_i is the number of individuals from caste i performing the task. It is assumed here that both castes are equally efficient at performing the task and that the strength δ of the stimulus scales up with the size of the colony.

With those three simple equations, (Bonabeau et al., 1996) obtained results very similar to Wilson's observations, using the following parameters: $\theta_1 = 8$, $\theta_2 = 1$ ($i = 1$ for majors, 2 for minors), $\alpha = 3$, $\delta = 1$ and $p = 0.2$ as shown in Figure 2

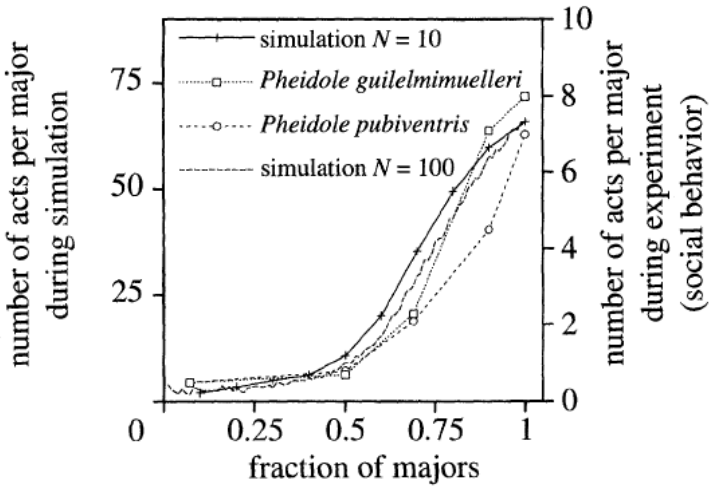

Figure 2 Comparison of Wilson's observations with FTM experimental results. From (Bonabeau et al., 1996)

2.2 The Dynamic Threshold model (G Theraulaz, Bonabeau, & Deneubourg, 1998)

Following on the FTM, (G Theraulaz et al., 1998) introduced an improved model incorporating the concept of specialization. The idea that task specialization of workers in an organization increases the efficiency of a system as a whole was already pioneered by political economist Adam Smith two hundred years ago (Smith, 1776). According to Smith, specialization in human industry had three benefits: (1) increased individual efficiency through learning, (2) reduction of switching costs, and (3) the invention of machines. As it turns out, points (1) and (2)

also apply to insect colonies. It was observed for example that ants and bees would start to display more and more specialization as they grow older (Robinson, Page, & Huang, 1994). Furthermore, (Withers, Fahrbach, & Robinson, 1993) showed that for honeybees, this specialization was the result of training and not simply a consequence of the ageing process.

Formally, the specialization can be modeled by a varying threshold. Since, according to the FTM, individuals with a lower threshold will have a higher probability to answer a task-related stimulus, (G Theraulaz et al., 1998) proposes that the more a worker performs a task, the lower its threshold for that task will be. Conversely, the thresholds for other tasks would increase.

For a system with N workers and M tasks, let θ_{ij} be the threshold of worker i ($i = 1..N$) for task j ($j = 1..M$). For each time step Δt that worker i performs tasks j:

$$\theta_{ij} \leftarrow \theta_{ij} - \xi\Delta t \tag{4.1}$$

and for each time step Δt that worker i does not perform task j:

$$\theta_{ij} \leftarrow \theta_{ij} - \varphi\Delta t \tag{4.2}$$

where ξ and φ are the learning and forgetting coefficients, respectively. The evolution of the threshold θ_{ij} is also assumed to be bounded in the interval $[\theta_{min}, \theta_{max}]$.

With this model, individual i becomes a specialist for task j when θ_{ij} becomes small. Figure 3 and 4 show experimental results of specialization in a colony of five individuals responding to two tasks. In Figure 3, all individuals start with the same threshold and we see that individual 3, 4 and 5 become task 1 specialists while individual 1 and 2 are task 2 specialists. In Figure 4, the initial threshold for each individual to each task is chosen from a uniform random distribution in the interval $[\theta_{min} = 1, \theta_{\max} = 1000]$. In this case, individual 3 and 5 become task 1 specialists while individual 2 and 4 are task 2 specialists. Because it started with a low threshold to both tasks, individual 1 ends up being both task 1 and task 2 specialist.

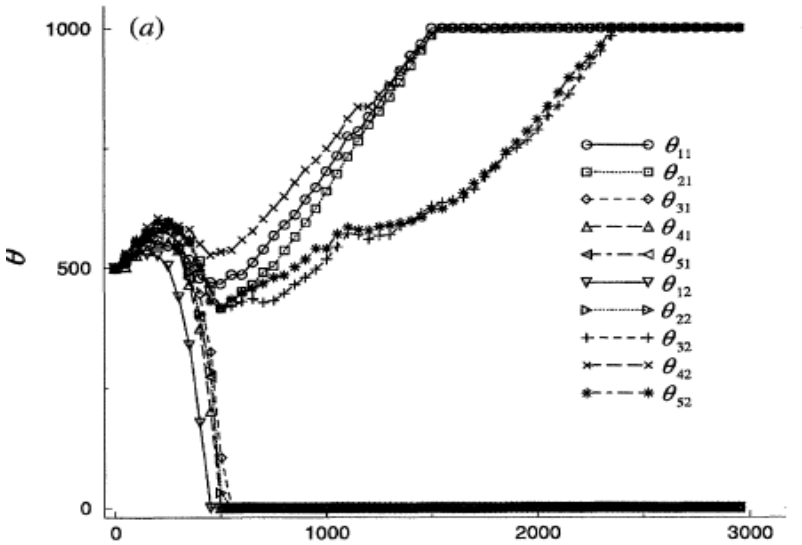

Figure 3 Dynamic of thresholds for N = 5 individuals and M = 2 tasks. α = 3, δ = 1, p = 0.2, ξ = 10, φ = 1. Θ_{ij} (t = 0) = 500. From (G Theraulaz et al., 1998)

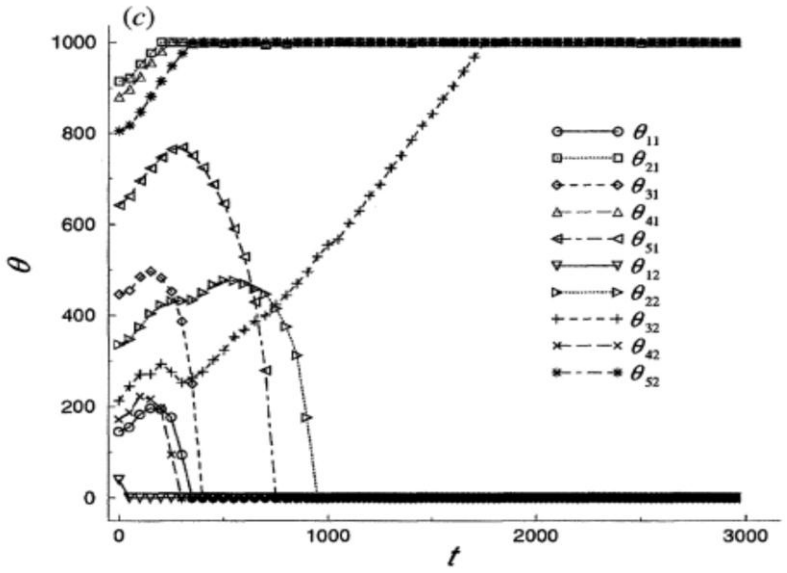

Figure 4 Dynamic of thresholds. Same parameters as Figure 3 except θ_{ij} (t=0) which is taken from a uniform random distribution in [θ_{min} = 1, θ_{max} = 1000]. From (G Theraulaz et al., 1998)

Immediately following those results, (Bonabeau, Sobkowski, Theraulaz, & Deneubourg, 1998) showed how this new knowledge about social insects can transfer to the field of intelligent system design by applying the DTM to the problem of mail distribution.

Imagine that a group of mailmen (we will call them agents) have to pick up letters in a large city. Customers expect their mail to be picked up and delivered within a limited amount of time. The mail company is therefore faced with the problem of allocating the agents to the various demands that appear within the course of the day, so as to keep the global demand as low as possible.

Let the city be divided into M zones, each with their corresponding stimulus S_j ($j = 1..\mathrm{M}$). For a mail company with N agents, the probability that individual i ($i = 1..\mathrm{N}$), located in zone $z(i)$, responds to a demand S_j in zone j is given by

$$P_{ij} = \frac{S_j^2}{S_j^2 + \alpha\theta_{ij}^2 + \beta d_{z(i),j}^2} \quad (5)$$

Where θ_{ij} is the response threshold of agent i to demand from zone j, $d_{z(i),j}$ is the distance between zone $z(i)$ and zone j (which can be Euclidian or include factors such as one-way streets, traffic lights, etc…), and α and β are two positives coefficients that modulate the respective influences of θ and d.

Each time an agent allocates himself to retrieve mail from zone j, with $\{n(j)\}$ being the set of zones neighboring j, his response thresholds are updated in as follow:

$$\theta_{ij} \leftarrow \theta_{ij} - \xi_0 \quad (6)$$

$$\theta_{i,n(j)} \leftarrow \theta_{i,n(j)} - \xi_1 \quad (7)$$

$$\theta_{ik} \leftarrow \theta_{ik} + \varphi \; for \; k \neq j, k \notin \{n(j)\} \qquad (8)$$

ξ_0 and ξ_1 are the learning coefficients associated to zone j and its neighboring zones, respectively, with $\xi_0 > \xi_1$, and φ is the forgetting coefficient associated with all the other zone. This procedure makes the agents more sensitive to demand in their current zone and the zones around it.

A simulation was performed for a city split in a grid of 5x5 zones and a pool of five agents. At every iteration, the demand increases by 50 in five randomly selected zones. Parameters for equation (5) to (8) were $\alpha = 0.5$, $\beta = 500$, $\theta_{min} = 0$, $\theta_{max} = 1000$, $\xi_0 = 150$, $\xi_1 = 70$, $\varphi = 10$. Agents are swept in random order and decide to respond to the demand from a particular zone according to equation (5). If no agent responds after 5 sweeps, the next iteration starts. If an agent decides to respond, it will be unavailable for an amount of time that is taken to be equal to the distance separating his current location from the zone where the demand comes from. Once in a zone, the agent is considered handling all the mail business of that zone and therefore the demand is kept at 0 in zones occupied by an agent. Figures 5(a) and 5(b) show the results of the simulation.

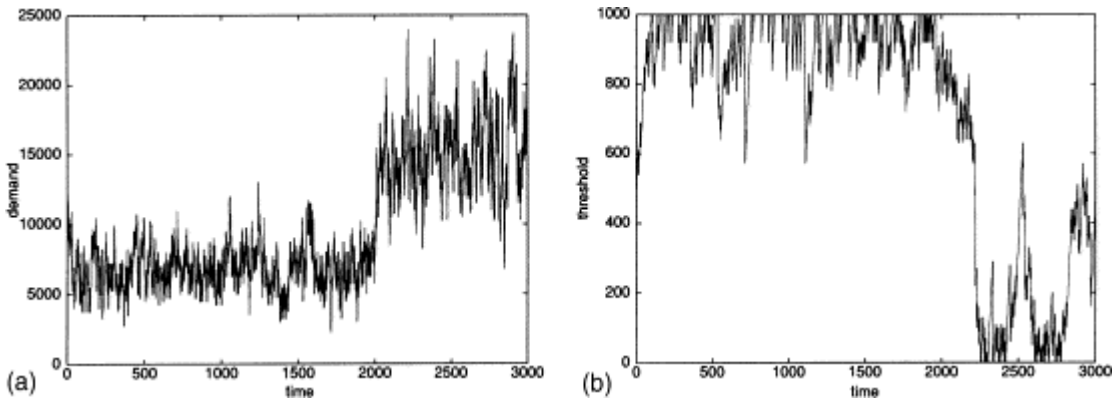

Figure 5 (a) Fluctuations of demand around a finite value in "normal conditions" and perturbed (at time = 2000). (b) Dynamics of one threshold under the demand in (a) From (Bonabeau et al., 1998)

In Figure 5(a), we see that the system can keep the global demand under control in normal conditions. An agent was then removed to observe the robustness of the system (at t=2000). We see the loss of an agent does not affect significantly the stability of the system, as the demand doesn't spike out of control. Figure 5(b) shows the evolution of one of the remaining agent's threshold. The agent that was removed was a specialist of the zone associated to that threshold. We see that after removal, the other agent becomes a new specialist of that zone to meet the demand. Threshold oscillations can be explained by a workload too high to allow all agents to settle into a given specialization, but nonetheless, the behavior of the system points to the flexibility and robustness of the algorithm.

The authors conclude with guidelines for adapting the algorithm to other task allocation problems but they also point out one common issue of adaptive algorithms, which is the lack of standard way to evaluate the performance of such algorithms, as commonly used benchmark problems are static problems (Bonabeau et al., 1998).

2.3 A benchmark problem for adaptive algorithms: the paint booth problem

To be able to gauge the performance of the DTM, the algorithm was applied to a dynamic flow shop scheduling problem by (Campos, Bonabeau, Théraulaz, & Deneubourg, 2000). The problem is to assign trucks coming out of a factory to paint booths, in order to minimize the total duration, and the number of paint flushes when a booth needs to change colors between trucks. The static problem, in which all colors are known in advance and booths never break down, is known to be a hard combinatorial optimization problem. In reality, the customer orders are treated in real time and booths may go down unexpectedly due to mechanical failure, which makes preplanning unreliable and the dynamic problem at least equally hard to solve. A market-based approach had been implemented in a real truck factory operated by General Motors, and it allowed significant improvement both in timespan and paint usage over the previous method, built around a centralize scheduler (Morley, 1996). In this approach, paint booths place bids for trucks based on their potential to paint the truck quickly. The booth with the highest bid is allocated the truck. Since that approach is actively used by the industry, its performance serves as a reference for evaluating the ant-based algorithm.

To test the different algorithms, the problem was formalized as follow: trucks get out of the assembly line at a regular pace of one per minute and have to be painted, some of them with higher priority than other. The color of the truck is predetermined by the customer order and it takes three minutes to paint a truck. However if the color is different from the truck that was just painted, the booth first needs to flush the previous color. This also takes three minutes and the old paint will be wasted, which adds to the overall paint cost. Therefore, the number of such changeover should be minimized.

The simulation is done using 20 possible colors, five to fifteen booths and a maximum queue of five trucks per booth. Booths with a full queue are unable to accept more trucks. If all booths are full, trucks are assigned to temporary storage until there is room for them. In addition, there's a 1 in 20 chance that a random booth will break down and be inoperable for a duration uniformly distributed between 0 and 20 minutes. Trucks already assigned to a broken down booth cannot be reassigned and therefore have to wait until the booth is operating again. The simulation runs for a maximum of 1000 time steps, but stops after the last truck is painted. The last truck comes out of the factory line at step 419th, which corresponds to a truck coming out of the assembly line every minute for seven hours.

The ant algorithm in (Campos et al., 2000) is an adaptation of the DTM algorithm described in the previous section. First the stimuli, which correspond to the demand for each of the j colors (j = 1..M), are computed.

$$S_j = \sum_{i}^{N} w_i * \delta(c_i - j) \qquad (9)$$

where w_i is the priority of truck i (i = 1..N), c_i is the color of truck i, and $\delta(\bullet)$ is the Dirac function ensuring that only trucks of color j are taken into account in the sum. By convention, trucks already assigned to a booth have $w_i = 0$.

Next, the tendency of a booth k to accept unassigned truck i is quantified by

$$P_k = \frac{S_{c_i}^2}{S_{c_i}^2 + \alpha\theta_{k,c_i}^2 + \Delta T^{2\beta}} \tag{10}$$

with α and β being parameters weighing the relative importance of their respective terms, and ΔT measuring the time it would take before truck i would be painted in booth k, measured by

$$\Delta T = q_k t_p + n_t^f t_f + t_k^r \tag{11}$$

In (11), q_k is the number of trucks in the queue for booth k, t_p = 3 min, which is the time it takes to paint a truck, n_t^f is the number of time the booth will need to be flushed because a truck in the queue requires a change of color, and t_f = 3 min is the time needed to make the change of color. Finally, t_k^r is the remaining time to finish painting the truck that is currently in the booth. Once all the P_k have been computed, the booth with the highest value gets assigned the trucks. Booths with a full queue are not considered. Ties are broken by looking at which paint booth would not have to flush between its last truck and the new one, then by comparing the length of the queues, and finally at random if no booth gets the upper hand.

After truck i is assigned to booth k, the thresholds of all booth are updated. For booth k:

$$\theta_{k,c_i} \leftarrow \theta_{k,c_i} - \xi \tag{12}$$

while for booth $m \neq k$

$$\theta_{m,c_i} \leftarrow \theta_{m,c_i} + \varphi \tag{13}$$

This specialization will increase the chance that a given booth gets assigned the trucks of a given color and thus minimize the number of paint flushes.

One critical point was then to determine the best combination of values for all six parameters presents in equation (10) to (13) ($\alpha, \beta, \xi, \varphi, \theta_{min}$ and θ_{max}). (Campos et al., 2000) used a genetic algorithm (GA) to fine tune those values. First a range was defined for each parameter to restrain the search space of the GA:

$$0.0 \leq \alpha \leq 1000.0$$
$$0.0 \leq \beta \leq 5.0$$
$$0.0 \leq \xi \leq 10.0$$
$$0.0 \leq \varphi \leq 20.0$$
$$0.0 \leq \theta_{min} \leq 10.0$$
$$0.0 \leq \theta_{max} \leq 200.0$$

Within the range of those parameters, a first population of 50 individuals is generated at random. Each parameter, or gene, is then encoded as a binary 8bit representation. Finally, the six genes are concatenated to form a 48bit *genome*. To evaluate the *fitness* of a given individual, its set of parameters is fed into the program for 30 runs so that the results, dependent on the randomly generated succession of colors, can be averaged out. For the first 50 generations, the fitness is defined as 1000 minus the number of steps over 420 (step 419 being the last one during which a new truck is produced) needed until the last truck is painted. For the remainder of the generations the fitness is that number minus the number of paint flushes needed by the booths. The idea is that it is most necessary to finish painting the trucks quickly, and that goal being accomplished, we want to minimize the amount of wasted paint.

Once the fitness of each individual within a generation has been evaluated, a tournament is staged to select which individuals will pass on their genes to the next generation. Two individuals are compared at random. The one with the highest fitness is chosen with a 75% chance, and the one with the lower fitness is selected in the other 25%. This process is repeated 50 times to obtain a new generation of the same size than the previous one.
Following the tournament, the new population is modified. Again, two individuals are chosen at random. This time there is a 75% chance that they will exchange part of their genomes. If a crossover happens, a random crossover point between 0 and the length of the genome is chosen and the two individuals exchange bits with each other up to the crossover point. Finally, a small mutation probability is introduced, and there's a 0.3% chance that any bit in any individual will switch from 1 to 0 or vice versa. After 100 generations, the genome from the fittest individual was chosen to run the simulations:

$$\alpha = 617.188$$
$$\beta = 4.66797$$
$$\xi = 7.85156$$
$$\varphi = 17.7344$$
$$\theta_{min} = 5.50781$$
$$\theta_{max} = 39.6875$$

Several test scenarios were defined. Only the first and most generic case is presented here but results for other scenarios were similar. In this scenario, each of the 20 colors has the same probability of occurring. One truck comes out of the assembly line every time step and the probability that a booth goes down is 0.05.

Compared to the state of the art Market-Based approach from (Morley, 1996), the dynamic thresholds method performs at least as well for the total time, and significantly better in terms of the number of flushes (Figure 6, t-test, df=998, P<0.01). (Campos et al., 2000) explain the better results in the number of flushes by the specialization effect displayed in the ant-based algorithm. However, they are very cautious about affirming the superiority of their approach over (Morley, 1996). First, because a certain number of ad-hoc assumptions were made since the details of Morley's algorithm, being an actual industrial process, were kept

secret by GM. Secondly, because it would not be hard to add the concept of specialization to the market-based approach, which could end up producing similar results. Nonetheless, the benchmark problem and its results have become a reference for testing agent-based dynamic task allocation algorithms.

Test set 1								
	Market-Based				Ant-Based			
	Time		Flushes		Time		Flushes	
Booths	Avg.	Std. Dev.	Avg.	Std. Dev	Avg.	Std. Dev.	Avg.	Std. Dev
8	5.21	3.02	326.82	10.85	5.20	3.44	315.65	16.19
10	3.01	1.13	298.39	11.17	2.88	0.87	260.96	11.89
12	2.72	1.04	263.52	13.15	2.60	1.13	220.06	12.18
15	2.27	1.31	211.49	13.74	2.14	1.34	162.12	12.92

Figure 6 Comparison of the market-based approach and the ant-based dynamic threshold algorithm for the paint booth problem. Time is the number of steps after the last truck is produced to have all the trucks painted. Flushes is the number of flushes required in one run of the program. Average and standard deviations are computed over 1000 runs. From (Campos et al., 2000)

2.4 R-WASP (Cicirello & Smith, 2001)

Parallel to the work on an ant-based algorithm (ABA) by (Campos et al., 2000), (Cicirello & Smith, 2001) developed a very similar algorithm based on wasp colonies. Wasps, like ants, use a stigmergy based system to solve the task allocation problem. In addition, wasps also interact directly with each other to form a hierarchical social order. Cicirello & Smith call their algorithm R-WASP, which stands for 'routing wasp', the basic decision making agents in their system. They also decided to show a concrete example by adapting their algorithm to Morley's booth painting problem. There are a few differences between R-WASP and Campos et al.'s ABA:

In ABA, there is a stimulus for each color. R-WASP instead decide to attribute a stimulus to each unassigned job (i.e. each truck that comes out), which is equal to the number of time units since the truck came out of the assembly line. Then, they use the stimuli to compute a probability of having a wasp-like booth request the truck. Let S_j be the stimulus associated to truck j and c_j be its color. Let $\theta_{k,cj}$ be the threshold of booth k for color c_j, the probability that booth k requests truck j is:

$$P_{k,j} = \frac{S_j^2}{S_j^2 + \theta_{k,c_j}^2} \qquad (14)$$

In R-WASP $P_{k,j}$ is used as an actual probability so it is possible that no booth would request a truck right as it comes out. Recall that in ABA on the other hand, the $P_{k,j}$ were compared and the truck was assigned to the booth with the highest value.

Thresholds are then updated. In ABA only the threshold for the color of the last truck assigned was updated. In R-WASP, all thresholds are updated each time unit. If booth *k* is currently painting or setting up for a truck of color c_j, the threshold for that color becomes

$$\theta_{k,c_j} \leftarrow \theta_{k,c_j} - \delta_1 \tag{15}$$

While for all other colors $c_o \neq c_j$:

$$\theta_{k,c_o} \leftarrow \theta_{k,c_o} + \delta_2 \tag{16}$$

However if booth k is currently idle and has an empty queue, then instead for all colors *c*:

$$\theta_{k,c} \leftarrow \theta_{k,c} - \delta_3^t \tag{17}$$

where *t* is the number of time unit that the machine has been idle.

Finally, R-WASP differs from ABA in the way ties are broken. To emulate the wasp hierarchical organization, R-WASP defines a force for each booth k:

$$F_k = 1.0 + T_{p(k)} + T_{s(k)} \tag{18}$$

where $T_{p(k)}$ is the sum of the processing times for the trucks in the booth's queue, while $T_{s(k)}$ is the sum of setup times that will be required to go through its queue. When two or more booths compete for the same job, a dominance contest is set up between the *n* contenders and booth *k* will get the job with probability

$$P_k(F_1, \ldots, F_n) = \frac{\sum_{i \neq k}^{n} F_i^2}{(n-1)\sum_{i=1}^{n} F_i^2} \tag{19}$$

This ensure that booths with short queue and low setup time for their queue will have a higher probability to be assigned the truck in case more than one responded to the stimulus.

	Morley	R-Wasps-D	p-value
Average	873.69±20.07	**972.22±2.11**	<0.0001
Median	876	974	–
Low	678	934	–
High	993	991	–

Figure 7 Throughput comparison (in number of jobs over 1000 minutes) between Morley's MBA and R-WASP. 95% confidence intervals and two-tailed p-values from a paired t-test are shown. From *(Cicirello & Smith, 2001)*

With those rules in place, (Cicirello & Smith, 2001) go on to test their implementation against Morley's market-based approach. However they also made different assumptions than Campos et al. used to test ABA. For Cicirello & Smith, one minute is 5 time units. The trucks still arrive at the rate of one per minute for a duration of 1000 minutes. The time to paint a truck stays three minutes but the time to flush and setup for another color was chosen to be 10 minutes instead of three. With a set of hand-chosen parameters ($\theta_{min} = 1$, $\theta_{max} = 10000$, $\delta_1 =$

100, $\delta_2 = 10$, $\delta_3 = 1.05$), R-WASP was shown to be significantly superior to Morley's as well (Figure 7), but since their settings were different (Campos et al. and Cicirello & Smith worked independently from each other), it was not possible to draw a direct comparison with ABA.

2.5 ATA and heterogeneous generalization (Nouyan, Ghizzioli, Birattari, & Dorigo, 2005)

Building on Cicirello & Smith's results, (Nouyan et al., 2005) propose four ameliorations to the R-WASP algorithm for the painting booth problem. They name this improved algorithm Ant Task Allocation, or ATA.

First, they notice that R-WASP updates the thresholds according to the current job that is being processed. This is motivated by the behavior of social insects who tend to specialize in the task they are currently doing and tend to continue doing it rather than switching to another task. However, in the paint booth problem it is very possible that a booth already accepted a truck that will require a change of paint in its queue. The booth thresholds to different colors should thus depend on the last truck in their queue and not their current job.

Second, they modify the force variable used in R-WASP dominance contest to take into account the fact that the new job would require a setup time. Thus the force variable becomes

$$F_k = 1.0 + T_{p(k)} + T_{s(k)} + t_{s(j)} \qquad (20)$$

where $t_{s(j)}$ is 0 in case no setup will be necessary between the last truck of the queue and truck *j*, and the required setup time otherwise.

Third, the dominance contest rule (19) is such that the more machines compete in the contest, the smaller are the differences between the probabilities to win. For that reason, equation (19) is changed to

$$P_k(F_1, \dots, F_n) = \frac{\frac{1}{F_k^2}}{\sum_{i \neq k}^{n} \frac{1}{F_i^2}} \qquad (21)$$

And last, they observe that despite the update threshold rule for idle machines, a machine can stay idle for a long time with a negative impact on the performances. They add an additional update rule for idle machines to remedy that. If an idle booth *k* refuses to bid for truck *j*

$$\theta_{k,c_j} \leftarrow \theta_{k,c_j} - \gamma \qquad (22)$$

where γ is a fixed value parameter.

They then test ATA against four other algorithms: ABA (Campos et al., 2000), R-WASP (Cicirello & Smith, 2001), MBA (Morley, 1996) and a greedy non-adaptive algorithm they call LOCUST

used as a base reference. For their simulation, they have 24 painting booths with a maximum queue of 10 trucks each. A booth takes 5 minutes to paint a truck and 10 minutes to setup for a new color. Each booth also has a 0.02 probability of breaking down, although they make no

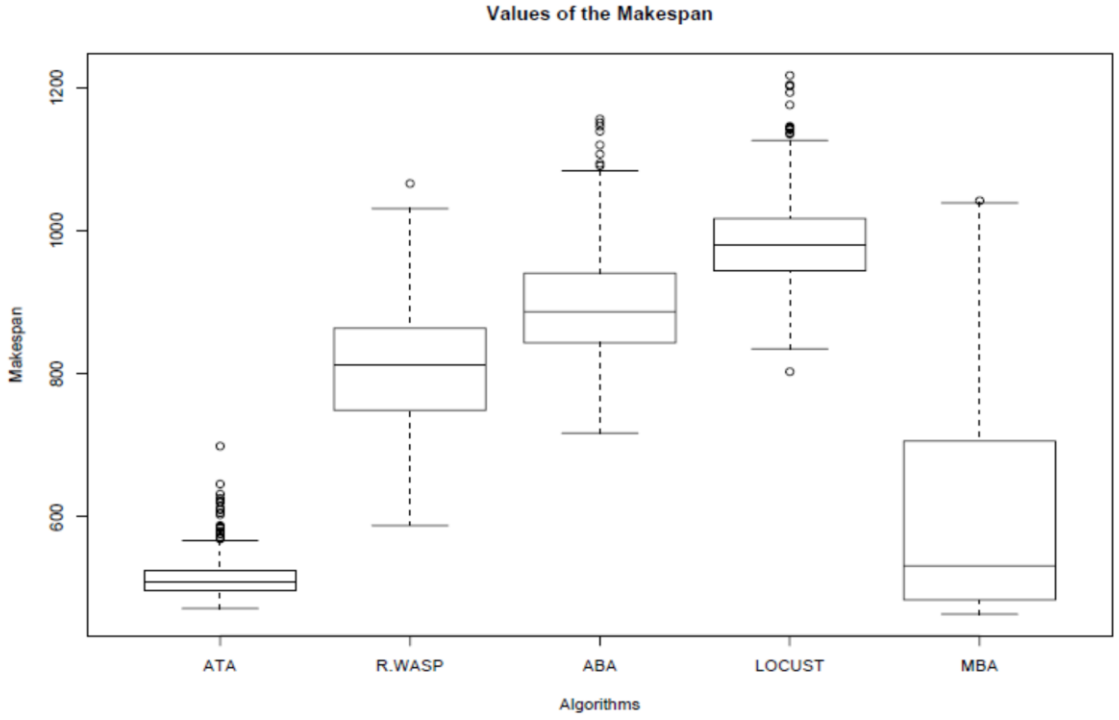

Figure 8 Box plot of the makespan (the total time it takes to paint all the trucks) for each algorithm over 1000 runs (lower is better). ATA is the best performing one and has the lowest variability. From (Nouyan et al., 2005)

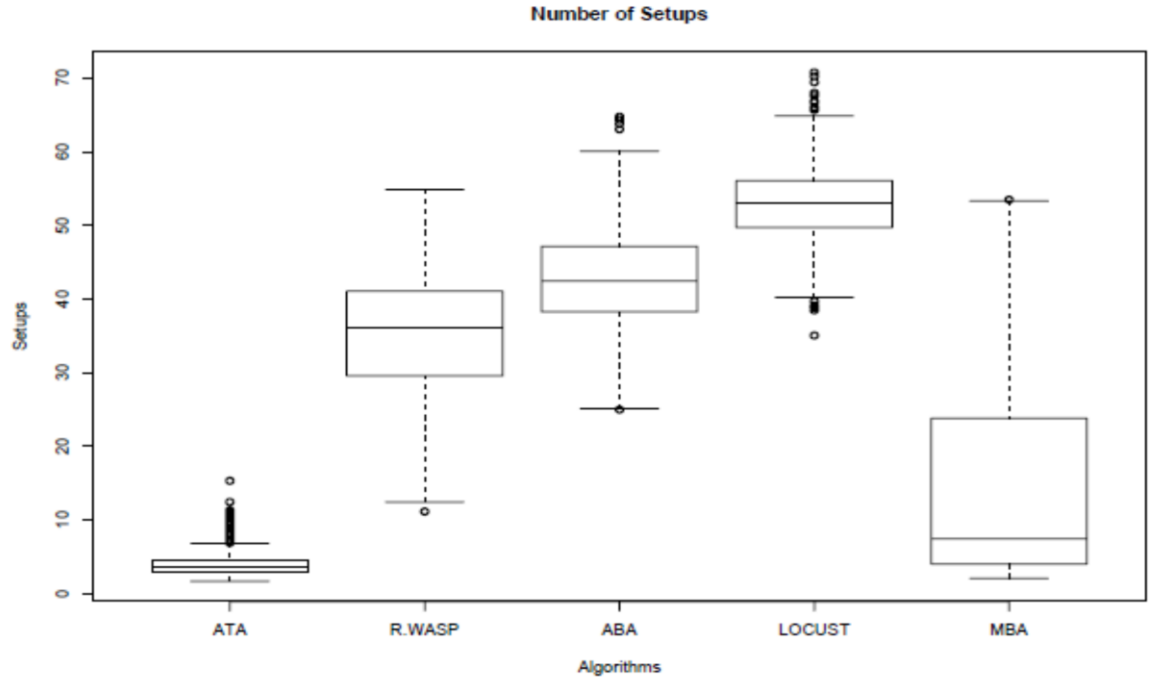

Figure 9 Box plot of the number of setups for all algorithms over 1000 runs. ATA is also the best performing one for this criterion. From (Nouyan et al. 2005)

mention of how long a booth stays inoperative.

The simulation runs for 420 minutes, during which a total of 2016 trucks come out of the assembly line. That number was chosen because it is the maximum number of trucks that could be painted without any setup or idle time by the 24 booths. They also use a non-uniform probability of a random color being requested, with one fourth of the colors being chosen 75% of the time. After tuning each algorithm's parameters with an evolutionary algorithm, the results show ATA significantly outperforming all other algorithms in both the time it takes to paint all trucks (Figure 8) and number of setups (Figure 9).

Alongside those results, Nouyan et al. also extended the dynamic task allocation problem to the heterogeneous case, where some of the agents can differ on how much time they take to complete a task. In a heterogeneous colony (such as the two castes present in Wilson's original observations) some agents are more suited for a given task than other, giving them a lower base threshold. This could be the case in a factory with old and new equipment for example. To take this setting into account, the probability function from (14) becomes

$$P_{k,j} = \frac{S_j^2}{S_j^2 + \theta_{k,c_j}^2 * (t_{proc(k,c_j)} - tmin_{proc(c_j)} + 1)} \qquad (23)$$

where $t_{proc(k,cj)}$ is the processing time of booth k for color c_j and $tmin_{proc(cj)}$ is the minimum processing time among all agents of the system for color c_j. This added term performs like a weight on a color's threshold to indicate the affinity of a booth with a given color. The authors then test that new rule on a modified instance of the problem. This time they limit the number of booths to 12 and lower down the number of trucks accordingly to 840. They separate

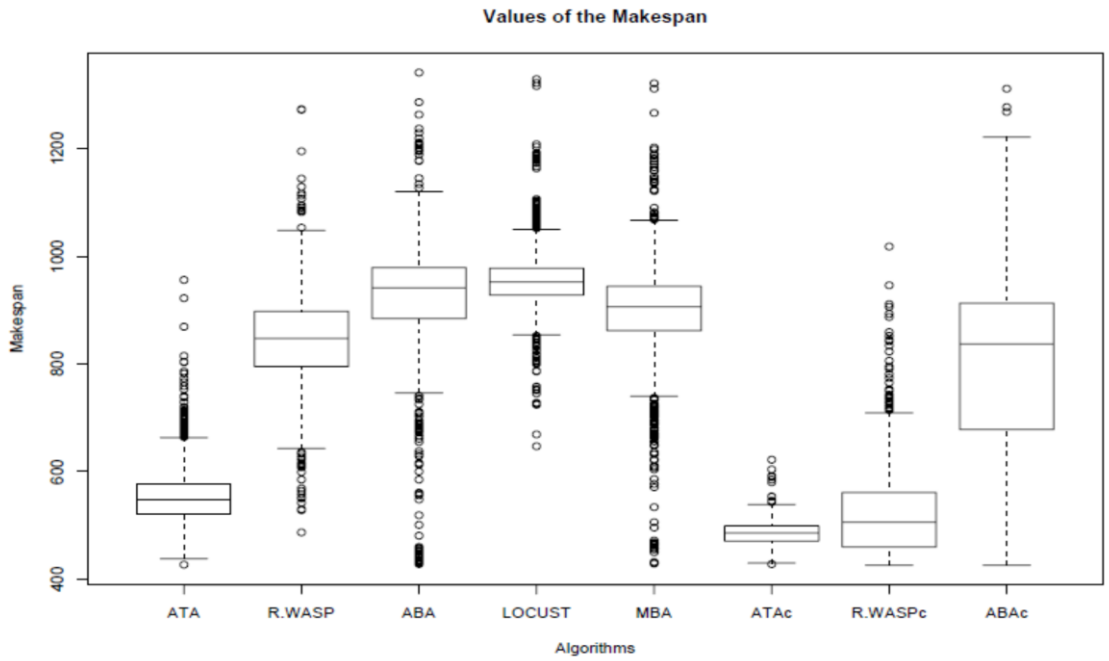

Figure 10 Box plot of the makespan values for the heterogeneous case. ATAc, R-WASPc and ABAc are the modified version of the algorithms using equation (23). From (Nouyan et al., 2005)

the booth in two subsets. The first subset needs three minutes to finish the half of the colors and nine minutes to finish the other half. The second subset has the opposite behavior. All other settings are the same as in the homogeneous case. They run a simulation with those settings using the original five algorithms along with a modified version of the three insect-based algorithms implementing the modification in (23). Once again, ATAc, the modified version of ATA for the heterogeneous case is the best performing (Figure 10).

If we look back more closely at Figure 8 and Figure 9, we see that within the scope of the scenario set forth by (Nouyan et al., 2005), Morley's original MBA is performing very well against the different insect-based approaches. In fact, only the finely tuned ATA can overcome it. We also can't help but notice that each of the approaches proposed in this chapter claims to best Morley's approach in a specific setting, but only ATA achieves better results in this last homogeneous experiment. With such a variance in the result, it is hard to proclaim one algorithm superior to the other, and we will suggest that each algorithm has its advantages in some situations. For the heterogeneous case however, Morley's MBA trails any of the modified insect-based approaches, so possibly this is where those algorithms can shine.

2.6 Hierarchical Model

There is very little work done on hierarchical or otherwise layered approaches to the response threshold model. Indeed one of the goals of a decentralized task allocation model is to remove the need for a higher level central decision making authority. Some species of wasps are known to be organized in a hierarchical manner (Guy Theraulaz, Bonabeau, & Deneubourg, 1995), and that aspect is used to model dominance contests in R-WASP, but beside that the only relevant piece of literature on that subject is (Wolf, Jaco, Holvoet, & Steegman, 2002). Their model combines a nested layered architecture with the response threshold mechanism. Their ant agents have a set of thresholds to stimuli both for a set of high level tasks (*behaviors*) and for the low-level actions (*sub-behaviors*) needed to fulfill the higher level goal. Changing stimuli can mean either reallocation to another sub-level task within the same behavior or a change or behavior entirely. For example, they define three behaviors: scouting, foraging and nursing. Scouting is then divided between the "search food source" and "lay trail to nest" sub-behavior; foraging can be either "follow trail" or "return to nest"; and nursing consists of only one sub-behavior, which is "feeding larvae".

Their agents' response thresholds are built around a *relevance* principle. While executing a sub-behavior, agents receive a positive or negative feedback. A positive feedback indicates the sub-behavior was partially or fully successful. A negative feedback can be generated to indicate the sub-behavior was not appropriate, for example when the execution was interrupted prematurely. For each sub-behavior, the following statistics are kept:

	BEHAVIOR ACTIVE	*BEHAVIOR NOT ACTIVE*
Positive Feedback	*j*	*k*
No Positive Feedback	*l*	*m*
Negative Feedback	*jn*	*kn*
No Negative Feedback	*ln*	*mn*

From that table, correlations between the state of the sub-behavior (active or passive) and the feedback can be computed:

$$corr(P,A) = \frac{j*m - l*k}{\sqrt{(m+l)*(m+k)*(j+k)*(j+l)}} \quad (24)$$

Here, *corr(P,A)* is the correlation between a positive feedback and the active state of the sub-behavior (*corr(N,A)* can be computed similarly). The relevance of the sub-behavior *j* is then

$$relevance_j = (corr_j(P,A) - corr_j(N,A)) \quad (25)$$

Since *relevance* can take values from -2 to +2, a threshold θ_j for sub-behavior *j*, varying from 0 to 100 can be obtained by

$$\theta_j = 100 - (relevance + 2)25 \quad (26)$$

This threshold can then be compared to the stimulus associated to that behavior, which is a weighted sum of the stimuli that the authors deemed relevant for a given behavior. For example, the stimuli associated to the foraging task are “food observed” and “food pheromone observed”, while the stimulus for scouting can be “food observed” and an “explore” internal stimulus. The probability that the behavior or sub-behavior will become active is computed according to the original FTM equation (1).

In their nested approach, a decision is first made at the level of the sub-behavior. If no sub-behavior is appropriate, control is passed over to the upper behavior level. This is the case when

- There are only weak or no stimuli for any of the sub-behavior in the current behavior.
- A huge percentage of the stimuli are irrelevant for the current sub-behavior or behavior
- The highest probability for a sub-behavior within the same behavior is lower than a given minimum.

When the ant finds itself in such a situation, it evaluates stimuli at the behavior level instead of the sub-behavior level, and then goes back to selecting a sub-behavior among those available for the newly selected behavior.

The effectiveness of such a model is not very clear. The results presented in (Wolf et al., 2002) only show how a single ant behaves when choosing between the forage or scout behavior. While the ant seems to behave in a ‘natural’ manner (i.e. similarly to what a real ant would do), there is no explanation of how the additional layer improves on having just a flat architecture. This leaves the question of a hierarchical threshold model very open and we will try to field some answers in chapter 6.

Chapter 3. Artificial Intelligence in Games

Games have been a domain of interest for artificial intelligence research since the origins in the 1950's, first with board games like chess and checkers. However it is only in the early 2000's that video games started to be considered as a serious research area. In (Laird & van Lent, 2001), the authors suggest that research in AI has lost sight of the initial grand goal, which is to create a human-level artificial intelligence. One of the reasons, according to them, is that most current applications can be solved more cheaply and more effectively by developing specialized AI solution rather than having a full-blown human AI and they propose to use video game challenges as a basis for this next-gen AI development. There are many arguments in favor: First, modern video games can offer a simulation environment that is controlled and can be easily interfaced with. Human-like AI will require at least some human-like sensory perception, and compared to dealing with real sensors and motor systems, video games provides synthetic environments where perception and actuation are perfectly under control. Second, working with a computer game rather than a homegrown simulation can save time and avoid the criticism that could result from simulation bias. Finally, since financing is often one of the decisive factors influencing research direction, the computer game market has now become large enough to have the means of financing research. Some studios have already taken steps to increase their collaboration with academia. For example, Ubisoft, the world's third largest game publisher, has signed an agreement with the university of Montreal and together they created the *NSCER-Ubisoft Industrial Chair on Learning Representations for Immersive Video Games*, which focuses on developing deep learning algorithms and has an annual budget of $200,000 (UdeM, 2011).

3.1 AI research areas in Games

There are a great many type of games. Each presents their own challenges in term of AI (Figure 11). For most of the 1980's and 1990's, game AI was limited to scripted behavior and path-finding algorithms like A*. A scripted AI is the equivalent of a rule-based expert AI system. It can perform very well but it is very rigid. In practice, this type of AI often performs poorly because the game mechanics can change during the development and writing the rules for the AI becomes the equivalent of trying to hit a moving target. Despite those limitations, this approach was the mainstream until recently. Indeed it doesn't require a lot of CPU power to run, nor any knowledge of more advanced, academic techniques. This is changing however, both because multi-core CPUs can provide enough resources to run move intensive AI processes without slowing games down, but also because the first generation that grew up playing video games has now joined the labor force, and they are eager to apply their newly acquired Ph.Ds. to game development (Takahashi, 2000).

Today, modern AI in games essentially can be found in three forms. First there is AI as a design aid. In what is commonly called procedural generation, an AI algorithm can generates game environments much faster and at a fraction of the cost of a game designer. To make those worlds believable for the players, this kind of AI has to give a sense that things were placed

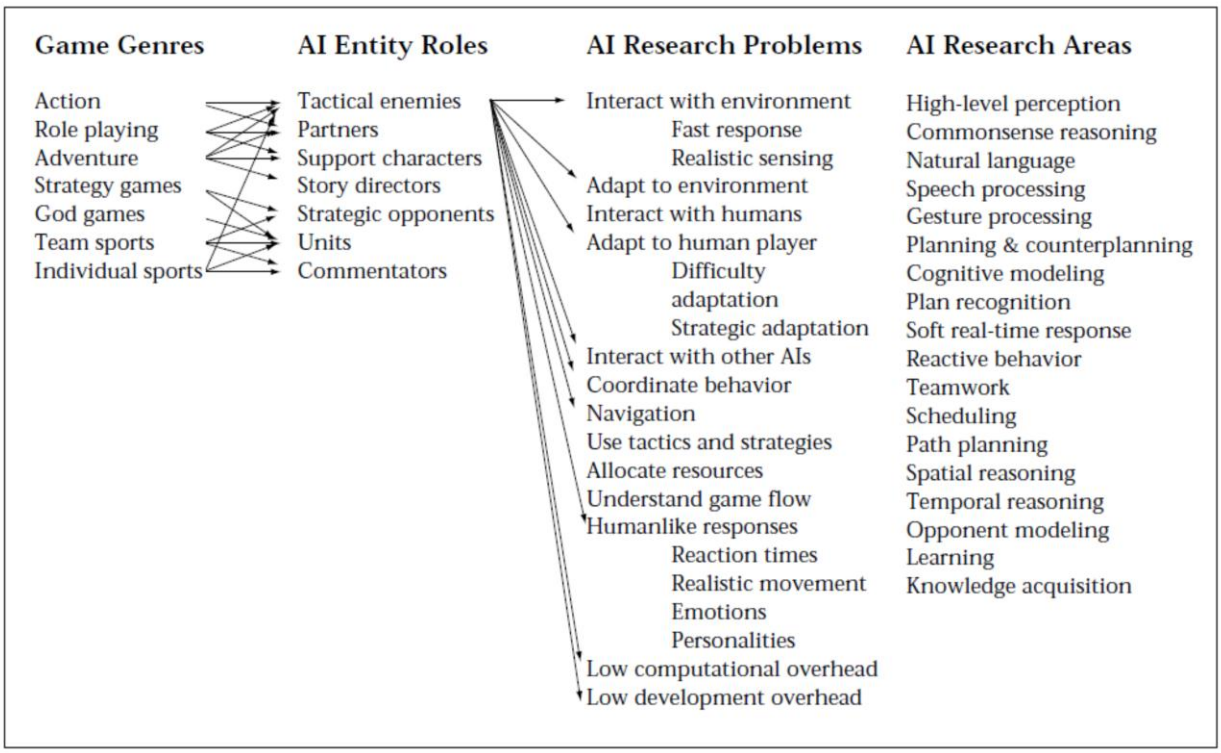

Figure 11 AI Roles in Game Genres with illustrative links to their associated research problems. From *(Laird & van Lent, 2001)*

where they are on purpose and not at random, which requires some level of awareness from the artificial designer (Nitsche, Fitzpatrick, Ashmore, Kelly, & Margenau, 2006).

Second, there is AI as a virtual actor, or non-playing character (NPC). In a lot of computer games, the player(s) control a hero or a set of heroes, and it is their interaction with computer controlled characters (NPC) that drives the story. Those interactions can be as basic as a scripted dialogs given by a random passersby, to the complex strategies that a SWAT team can set in motion during a hostage crisis. Whether those NPCs are simply cannon fodder or full-fledge artificial companions, it is important that they act and react in a believable way. Many techniques have been tried here, the most popular being behavior trees and artificial neural networks (Champandard, 2011).

Third there is AI as an artificial opponent. While it could be argued that it is the same category than the previous one and artificial opponent certainly wouldn't lose to behave more like humans, there is the additional requirement that this opponent can adapt to the human player strategy (or lack of), and provide an adequate level of challenge no matter the skills of the player. And while more and more games are now focusing on the multiplayer experience, playing online is not for everyone. Not taking into account the people who do not have a reliable internet connection since that category is shrinking fast; playing against other humans can be a stressful and sometimes unnecessarily emotional affair.

In the remainder of this thesis, we will show how we can adapt the response threshold model described in Chapter 2 to create such an AI system for a turn-based strategy game.

3.2 AI for a turn-based strategy game (TBS)

In a typical TBS, such as *Colonization*, each player takes turn growing their resources, via diplomatic, economic or military means, until one of the players becomes strong enough to declare victory over the others. The TBS genre is interesting for AI research because it presents a high level of complexity, while not placing a hard bound on how much computation time is available. Indeed, just like in (non-competitive) chess, each player can take several minutes or even hours before finishing their turn. Furthermore, TBS usually present the game world as a grid, where each square (also called *tile*) represents a possible location. This ready-made discretization of the world is often welcomed for state-space searches. (Bergsma & Spronck, 2008) identify six research challenges commonly found in a TBS game:

- **Adversarial planning**: trying to anticipate the adversary's moves to plan around it.
- **Decision making under uncertainty**: Coming up with a plan without knowing the full probability distributions for each actions' consequences
- **Spatial reasoning**: the ability to understand one's geographical surroundings.
- **Resource management**: Optimizing the output of a complex system with limited input resources.
- **Collaboration**: fostering complementary behaviors in multi-agent systems to overcome problems that a single agent cannot solve.
- **Adaptivity**: being robust to changes.

3.2.1 Related work

Many papers delve into one or the other challenges mentioned above. We will now present five of them which each are related to our own research in some aspects. The following table summarizes the challenges tackled and the techniques used.

(Wender & Watson, 2008): Decision making under uncertainty and spatial reasoning. They used a Q-learner to learn the best positioning for a city.
(Amato & Shani, 2010): Adversarial planning, decision making under uncertainty, adaptivity. They compared the results of a simple Q-learner, Dyna-Q and a factored Dyna-Q to learn a high-level adversarial strategy
(Bergsma & Spronck, 2008): Adversarial planning, spatial reasoning and adaptivity. They used an evolutionary algorithm to compute influence maps to control units actions.
(Sánchez-ruiz, Lee-urban, Muñoz-avila, Díaz-agudo, & González-calero, 2007): Decision making under uncertainty, adaptivity. They mixed a hierarchical task network (HTN) planner with a ontological knowledge base to build strategy plans from a case base of similar plans.
(Hinrichs & Forbus, 2007): Resource management, adaptivity. They attempt to maximize the production output of a city by using a HTN planner coupled to a qualitative model.

A couple of Q-learning approaches have been tried for the TBS *Sid Meier's Civilization IV*, a sequel to the original 1991 game. The first approach, in (Wender & Watson, 2008), tries to learn the best sites to build a city. In *Civilization IV*, cities are the backbone of your empire, bringing you gold, science and production capabilities. However, cities can only exploit a few tiles around their founding location, which makes it critical to choose the best available site. The authors replaced the built-in deterministic rule-based logic for site selection by an ϵ-greedy Q-learner, while keeping the rest of the built-in AI. One of the issues of Q-learning in TBS is to choose an appropriate model for the game state. Indeed, a complete state can number hundreds of variables and the Q-learner would not converge towards the optimal policy in any reasonable amount of time. Here, the authors decide to model a state $s \in$ state space S as a triple (x,y,z), where (x,y) are the coordinates of the city on the map, and z is the rank in the founding sequence (1 being the first city built, i being the i^{th} city built). Despite restricting the state space to a fraction of what a human player would consider, playing the game only 50 turns (enough to build about two cities on average) on a small map, they calculate $|S| \approx$ 90000. With those limited settings, their Q-learner starts to give better results than the built in AI after about 2000 episodes (Figure 12). While those results can seem promising, there are several limiting factors. First the game is played only during a short period that might not be representative of a full game, since it can last for several hundred turns. Increasing the number of turns also increases the number of cities built and thus the state space grows exponentially every 30 turns or so. Also the time to run each episode increases with the number of turns, and training the Q-learner quickly becomes unmanageable. To run their Q-learner for episodes of 100 game turns, they had to cut the number of training episodes by half to keep

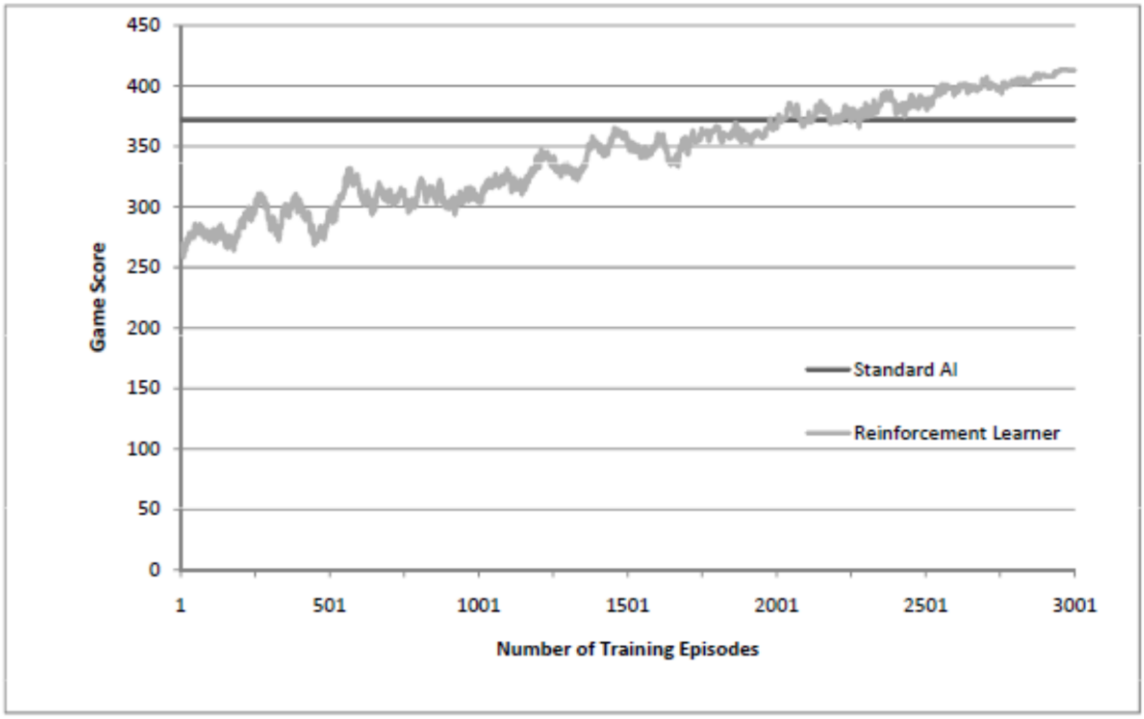

Figure 12 Average Game score for Q-learner and standard AI after 50 turns. From *(Wender & Watson, 2008)*

the running time of the experiment manageable. While the Q-learner showed increasing performance over time, it was not able to score better than the standard AI due to the lack of training time. Finally, and this is probably the biggest downside to this approach, the limited number of information contained in the game state means that the Q-learner has to be retrained on every map. Indeed, it does not understand why a location is better than another (proximity to resources, strategic positioning, etc…). The fact that an (*x,y*) location is desirable on one map is a function of its surroundings and does not generalize for other maps.

Using the same game, (Amato & Shani, 2010) detail their efforts to implement a high-level learner for the complete game. In *Civilization IV*, the AI opponents are given generic traits (i.e. aggressive, expansive, financial, organized, etc…) which influence the way they play the game strategically. Those traits are fixed throughout the game, and the idea is to see whether the AI can learn to adopt the best trait for the current situation to improve its performance. Like Wender & Watson in the approach detailed above, they start by defining the state space. Here the authors include four features: The population difference between the two opponents, the difference in land area controlled, military power difference and finally how much land is left up for grabs. They discretize each feature to restrict them to {0, 1, 2}. This gives 3^4 possible states, a much more manageable search space than Wender & Watson's attempt. They train a Q-learner, Dyna-Q (a model-based Q-learner) and a modified version of Dyna-Q learning over the factored state space where the transitions between features (population difference, land difference, military power difference and remaining land) can be learned independently. The training takes place over 50 and 100 episodes to evaluate which of those

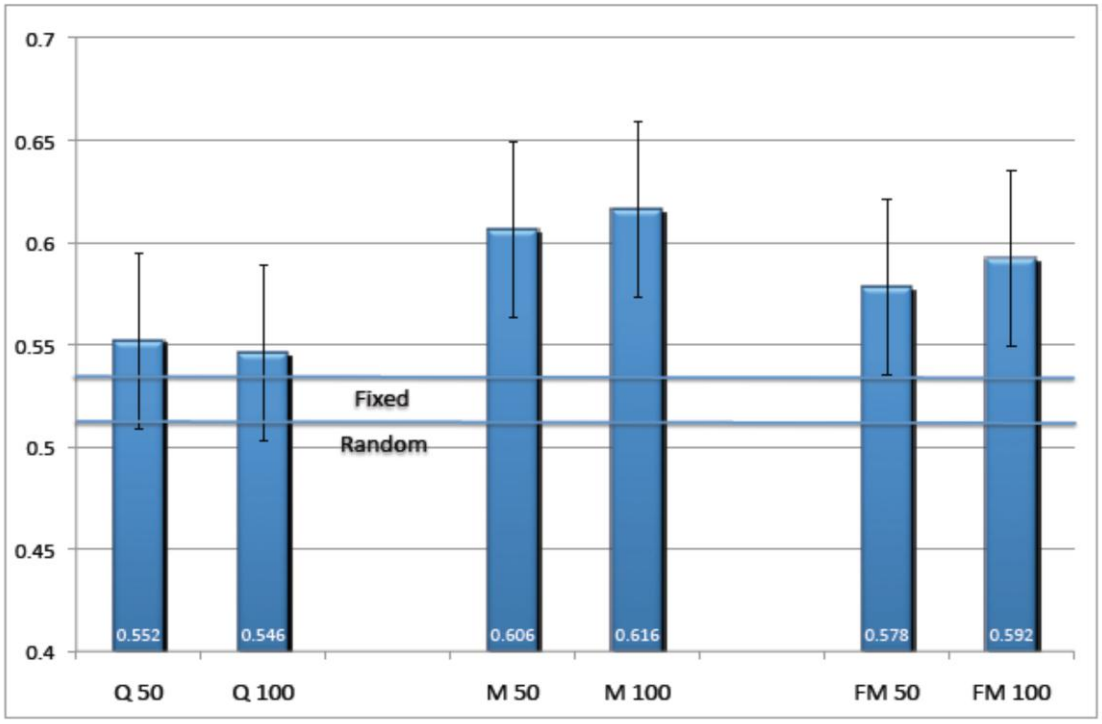

Figure 13 Winning frequency for the Q-learner (Q50, Q100), the model-based learner (M50, M100) and the factored model-based learner (FM50, FM100) over 50 and 100 training episodes, compared to a fixed and a random policy. From *(Amato & Shani, 2010)*

three approaches performs best (Figure 13). All Q-learners demonstrated better performance than the fixed built-in AI, with the model-based Dyna-Q being the best, thanks to the faster learning curve gained from learning the model.

Those two examples show that while reinforcement learning could be a valid venue for AI in TBS, the model has to be kept simple. Wender & Watson only kept the map coordinates while Amato & Shani used composite high-level scores. Once we delve into lower-level management with potentially huge branching factor, like ordering single units around, other approaches should be investigated.

One of those is tried by (Bergsma & Spronck, 2008). Using their own custom TBS game (a simplified clone of NINTENDO's *Advanced Wars*), they propose an architecture they call ADAPTA (Figure 14). ADAPTA divides the AI into tactical and strategic modules. A tactical module contains the basic subtasks for a given policy, such as *move* or *attack*, while a strategic module represents a goal that the AI has given itself (such as *destroy unit X*). Interestingly, they propose an allocation system where tactical modules would bid for game assets such as units or resources, which reminds the market based approach from Chapter 2. Those bids are then weighted by the utility of their tactical module and assets are then allocated to maximize the 'social welfare' of all modules. Unfortunately, they do not provide any detail on how such a mechanism would work and focus instead on developing spatial reasoning for a single tactical module (the extermination module in charge of combat operations). The extermination module essentially has to decide where to move units and what to attack. It accomplishes that by computing influence maps, which assigns a value to each map tile to indicate the desirability for a unit to move towards the tile. This value is computed by an artificial neural network, taking the different parameters as input and producing one influence map for movements and another for attacks. They use an evolutionary algorithm to tune the weights of the ANN, where the fitness function is obtained by pitching an AI using those influence maps against a scripted basic AI for the first generation, then against the previous generation for the subsequent iterations. Their final evolution is able to consistently beat previously generated AI, showing adaptability and spatial reasoning.

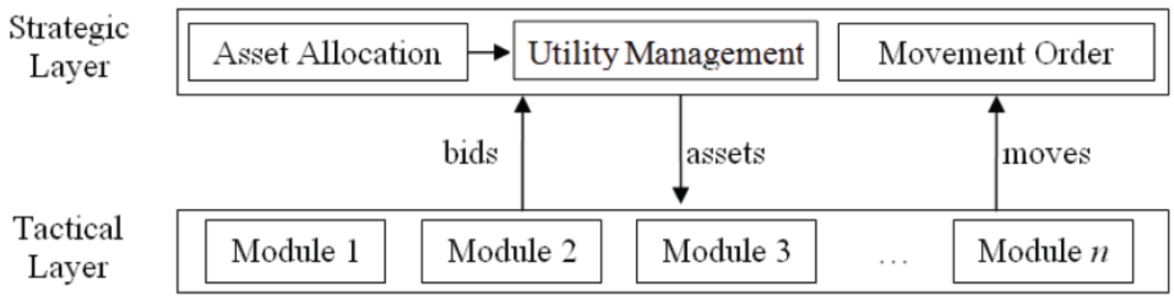

Figure 14 The ADAPTA game AI architecture. From (Bergsma & Spronck, 2008)

Another technique is proposed in (Sánchez-ruiz et al., 2007). They studied how to combine case based planning techniques and ontological knowledge of the game environment to construct an adaptive game AI. Using *Call to Power 2* as a test bed, a game similar to *Colonization*, they engineered a combination of several existing tools into a somewhat complex architecture (Figure 15). Their planner, Repair-SHOP (Warfield, Hogg, Lee-Urban, & Munoz-Avila, 2007), can retrieve and adapt plans that were used successfully in similar situations. Usually, case based retrieval works by computing the foot-print similarity between the current situation and each situation in the database. The foot-print is the number of exact matches between elements present in both situations, and the planner simply tries to adapt the plan with the highest foot-print similarity. To improve the relevance of the plan returned, the authors add a knowledge based under the form of a world ontology. The ontology can be seen as graph linking all game objects together. For example the graph will have a parent node '*Unit_type*' with children nodes for '*ground*', '*naval*', and '*aerial*' units. Partial similarities can be computed by looking at the distance between two objects. The authors hope that this can improve the relevance of the plans retrieved, and make it easier to adapt them. For example, if the AI wants to attack a city with a soldier and a tank, and has a successful plan in the case base to attack a city with a warrior and a knight, the foot-print method would return 0 (no exact matches), while the ontology-based retrieval method would figure out the similarity between the soldier and the warrior, and between the tank and the knight, and add this plan to the list of potential plans. However, while it seems like a very sensible approach, in their paper the results were

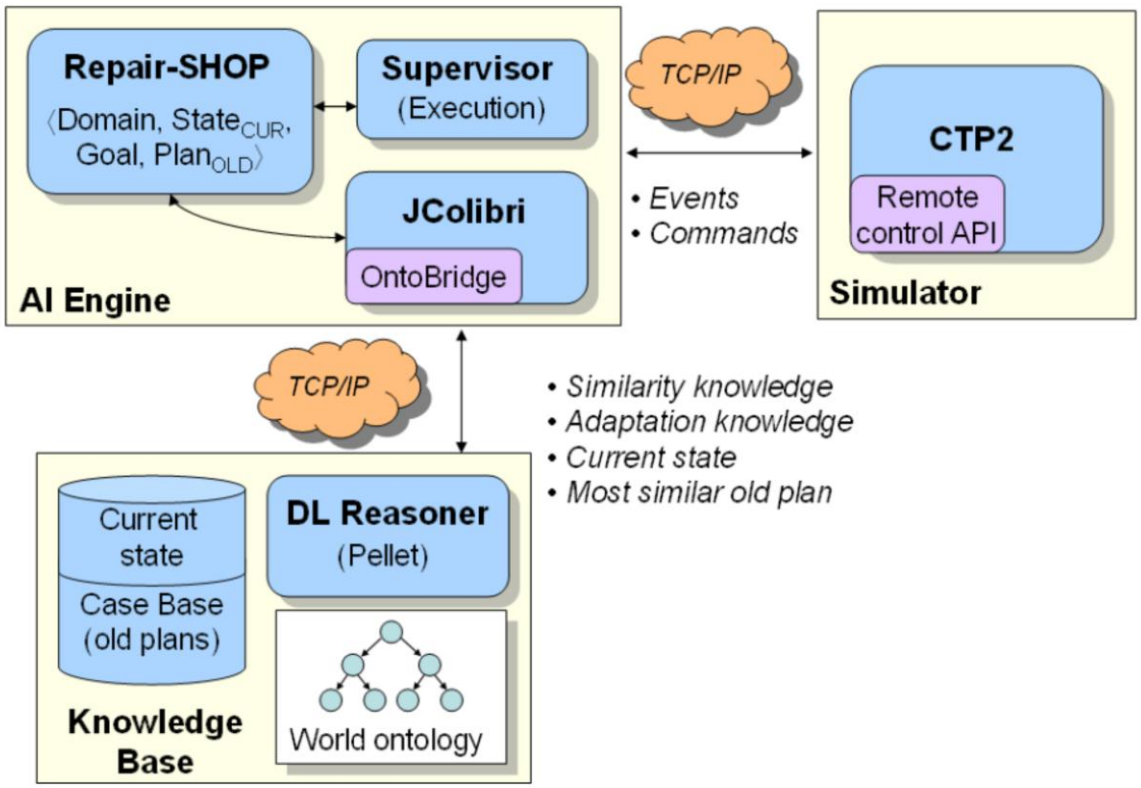

Figure 15 Architecture for Plan Adaptation with Ontology-based retrieval. From *(Sánchez-ruiz et al., 2007)*

limited to showing how this could technically be done, and they were planning to build a case base large enough to start running tests. Without actual experiments, it is hard to say for sure if it would make much of an improvement on the standard retrieval method.

The last example we would like to present also uses case based planning. (Hinrichs & Forbus, 2007) integrate a relatively complex system for planning, executing and learning in strategy games. While their ultimate goal of analogical learning is more ambitious than our own, they focus their experiment on maximizing a city's food output (they are using a *FreeCiv*, an open-source clone of *Sid Meier's Civilization*) by learning how to allocate workers within the city, and this is somewhat similar to what we're trying to achieve. Like (Sánchez-ruiz et al., 2007), they incorporate a hierarchical task network planner using the SHOP algorithm (Nau, Cao, Lotem, & Muftoz-avila, 1999). Their planner is capable of computing similarities between situations and can learn by quantitatively evaluating the success of its plans, rather than simply categorizing them as 'success' or 'failure'. Furthermore, to help the planner explore new solution, it can query a qualitative model (Figure 16). This model can suggest primitive actions that influence the goal quantities in the right way. Finally, to overcome local maxima in the learning process, the planner can generate new learning goals when it fails to improve on a plan. For example, if it fails to improve its goal of having the largest food surplus possible, it posts a new learning goal of figuring the maximum food that can be produced on an individual tile. Once it is done, it can evaluate how close from the theoretical maximum his current plan is, and according to a predefined tolerance for risk, can either explore new venues of be satisfied with the current plan.
For their experiments, they tested not only the learning power of their planner, but also the effects of learning transfer. They first train their planner for 12 sequences of 10 games for the city of Philadelphia. Each time they measure the average food production after 50 turns. Then

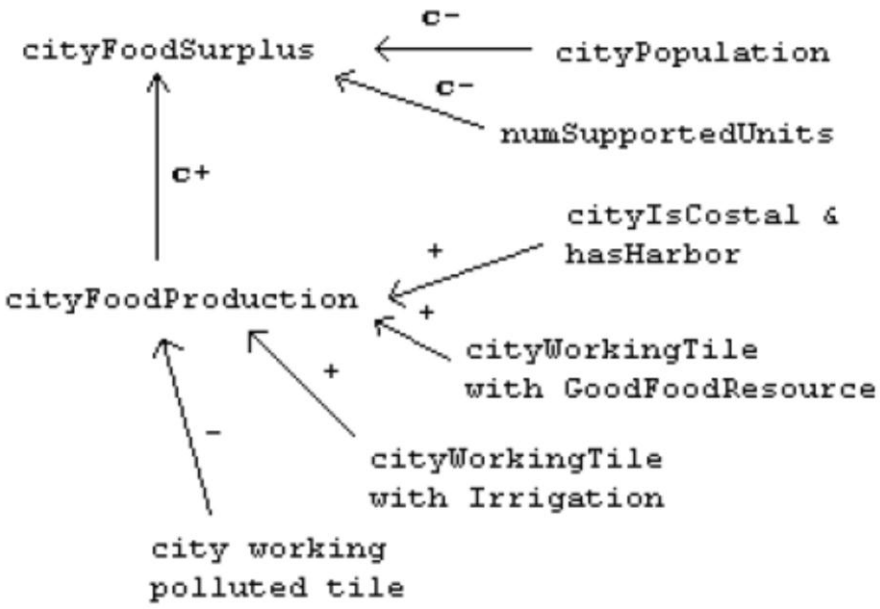

Figure 16 A portion of the city qualitative model with positive and negative factors affecting the growth of the city. From (Hinrichs & Forbus, 2007)

they apply their planner to another city, New York, which has a very different set of tiles to exploit; first with an empty learning slate, then with the case base learned from training on Philadelphia. They observed that not only their planner can learn how to allocate units correctly, it also benefits from knowledge transfer, as the case base learned from Philadelphia gives a 36% initial improvement increase to the learning process for New York (Figure 17).
Despite those promising results, we can't help but notice that the scope of the experiment is very limited. First, food production is only one of many aspects of city management, all other game parameters were fixed. It would have been interesting to see if their planner can also learn the optimum balance between several goals (i.e. maximizing production and gold as well as food). Second, the experiment runs only for 50 turns while a full game usually lasts between 400 to 600 turns. As Figure 17 shows, in a game like *Civilization*, there isn't room for a lot of variance in such a short episode. The planner's sub-optimal choices may only become apparent over longer runs.

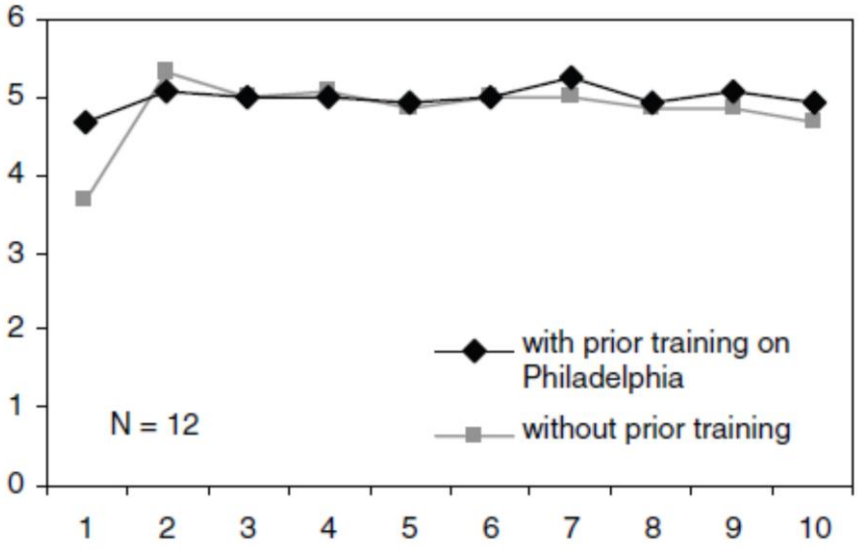

Figure 17 Effects of learning transfer. Average food production in New York after 50 turns (12 sequences of 10 games). From (Hinrichs & Forbus, 2007)

From those five examples, we see that the challenges of strategy games are by no means trivial, and that like many other domain of applications for AI, it is still in infancy with no technique proving substantially superior to another. Our work will thus add to the research by showing yet another promising path of investigation in this domain.

Chapter 4. FreeCol

FreeCol is an open-source implementation of *Sid Meier's Colonization (MicroProse, 1994)*, which is itself a variation of the archetype for turn-based strategy game: *Sid Meier's Civilization (MicroProse, 1991)*. In this chapter, we will first introduce the main game concepts and sketch what would be a typical game of *Colonization*. Then we will delve deeper into some of the game mechanics that will influence our experiments.

4.1 Game concepts

A game of FreeCol takes place on a isometric 2-D *map* representing the New World as it was when discovered by Christopher Columbus in 1492. Although it is possible to play with an accurate map of the Americas, to increase the replay factor a standard game will generate a random map, complete with oceans and forests, plains and mountains, rivers, deserts and local Native American tribes. This map is discretized according to a grid, and each square is called a *tile* (Figure 18). Each tile is uniquely identified by a (*x,y*) set of coordinates, which makes it easy to handle for a computer AI. With the standard settings, a player can only see the tiles that have been explored by his units, the rest being covered in the '*fog of war*', which simulate the imperfect geographical knowledge that early explorers experienced.

Figure 18 Partial FreeCol Map with tile grid displayed. The Dutch player colonies and their area of influence are shown in orange, while an unknown Native American settlement is shown in brown.

Figure 19 The discovery of America in FreeCol. Most tiles are covered with a non-descript dark blue 'fog of war' before they are explored

Figure 20 Foundation of the first Spanish settlement: Isabella. The number on the settlement indicates how many units are working in there.

A player starts the game with a *vessel* transporting two *colonists* approaching the shores of the new continent. Only a few tiles around the boat are visible, the rest is covered by the fog of war. After navigating west for a turn or two he gets his first sight at the new land (Figure 19). Now comes the time to establish the first colony. The two colonists disembark from the

boat and start exploring their surroundings, before settling in a nice spot by the ocean (Figure 20).

Then comes the part that will be the focus of our experiment: colony management. For our New World nation to grow, the player will need to try to optimize the *goods* output of each of his colonies. Colonies automatically produce some basic goods from the tile it is built on, but it can also exploit the eight tiles around it by assigning a colonist to work on them. Tiles will produce different basic goods (food, lumber, ore) depending on their types (plain, forest, marsh, hills,…) and bonus resources that are sometimes located on them (gold, corn, tobacco, furs,…). Furthermore, a colony contains a certain number of *buildings* in which colonists can also work to produce either refined goods (cigars from tobacco, clothes from cotton, hammers from lumber, …), or special goods: *bells* that measure the liberty of the colony from their European monarch, and *crosses* that measure their attractiveness to new immigrants. All of this is done through the *colony screen* (Figure 21). From top to bottom, left to right, we can see the name of the colony, and the number of resources that are being produced. Below we have the minima with the tiles that the colony can exploit and the buildings present in the town and where colonists can work. Below the minima we have the size of the colony, which is simply the number of colonists working, and their split between rebels and royalists, which we will explain below. Then, one can see the advancement of the current *production item*, which can be either a building or a *military unit*. Under the production and buildings area, there are three rectangles showing which non-working units are where (port, on a vessel at port, or idling right outside the colony). Finally, the last line shows the state of the *warehouse*, where the different goods are stored.

Figure 21 Annotated Colony Screen with 1 Colony tile with a colonist producing food. 2 Colonist producing lumber. 3 Colony production output. 4 Town hall with a colonist producing bells. 5 Carpenter's house with a colonist producing lumber. 6 Colony size. 7 Production modifier. 8 Current item being built with progress bars for hammers and tools. 9 Colonist outside the colony (does not count toward colony size). 10 storage status for the different material goods.

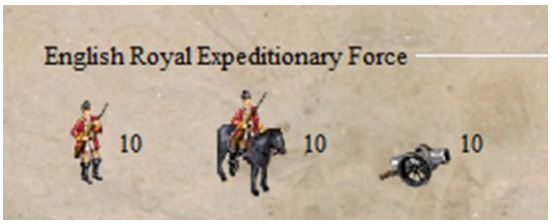

Figure 22 The Royal Expeditionary Force, which the player has to defeat to win the game.

Over the course of a game, the player alternates between moving around some of his units on the game map, and assigning the rest to work at different jobs within his colonies. Victory (or bitter defeat) comes when the player managed to foster enough liberty sentiment among the colonists to *declare independence* from his *monarch*. At which point said monarch sends the *royal expeditionary force*, a fearsome ensemble of military units, to crush the rebellion and reclaim the colonies for the crown. Victory is awarded to the first player that defeats his monarch's expeditionary force.

It is worth noting that some aspects of the game are only accessory to claiming victory. For example, the player can trade and sell surplus goods produced in his colonies, either to the natives or the monarch. Gold obtained from those transactions can buy military units or helps new immigrants join your fledging nation. However, military units can also be built at no monetary costs inside your colonies, and new colonists will automatically appear both as a result of food surplus in a colony and from natural immigration. Likewise, no real military strategy is required besides defending colonies. While the player can certainly engage in complex tactical warfare operations with other nations and native tribes, after declaring independence the royal expeditionary force will automatically appear next to the player's colonies and start trying to reclaim them.

Thus, to ensure victory, our AI player will be satisfied with mastering only the following subset of the game concepts: managing colonies for food, liberty, production and military defense. We will now details each of the concepts *outlined* in the previous section.

4.1.1 Units: Free Colonists, vessels and military units

The basic workhorse of the game is the free colonist. This unit is very flexible and can work most of the job positions within a colony. It can also be equipped with tools to build roads, clear forests to transform them into plain, or plow a plain, prairie or grassland to increase its food output. Last, it can also be equipped with muskets to become a soldier.

Moving a unit to a different job within a colony is immediate. For example, a worker can be moved from farming a tile to working in the carpenter's house and then back to working a tile in the same turn without penalty.

When working a tile for a colony, the free colonist will produce some goods and gain experience. Experience gives him a chance to become an expert at its current job (Figure 23). Each turn, a free colonist gains a number of experience points equal to the production output of the tile it is working on. For example, a colonist farming a plain will gain 5 experience points

Figure 23 Areas of expertise available to free colonists

per turn, while a colonist working as a lumberjack in a mixed forest with extra lumber will gain 10 experience points per turns (Table 1, see 4.1.2). At the beginning of each turn, a check is made for each colonist to see if they have become expert in their profession. The probability P_{expert} of such an event is defined by

$$P_{expert} = \frac{\min(Experience_{unit}, 200)}{5000} \qquad (27)$$

In other words, a unit has at most 4% chance to become an expert each turn, if it has maxed out its experience. Expert units will produce more basic goods when working their profession. An expert farmer will add 3 units to its food production, while an expert lumberjack or an expert miner will double their respective production on their tiles. An expert that is not working a job in his area of expertise does not receive any bonuses. Also, while units can decide to change their area of expertise, they cannot be expert in two domains at the same time. Finally, colonists cannot become experts by working in buildings. The option to become an expert at those tasks is the result of a somewhat complex process involving training units in a school or university and paying for the appropriate trainer. Since we do not concern ourselves with the financial aspect of this game, we restrict the expert classes to expert farmer, expert lumberjack and expert ore miner.

Besides producing goods, units can also be used to transport goods from one place to another, or to provide military protection. At the beginning of the game, the player starts with a vessel transporting two land units. There are different types of vessels, each with their own capacity, speed and military power. Mostly, during the game they handle shipping of goods and new immigrants from and to Europe. Since our AI will not focus on trade, we will not expand more on this topic.

Military protection is ensured either by armed colonists, or by artilleries, which can be built in colonies. Contrary to armed colonists who can lay down their muskets and go back to working colony tiles, artilleries can only attack or defend. In exchange, they are much more powerful than armed colonist. While our AI will try to protect itself as best as it can in expectation of

the onslaught of the royal expeditionary force at the end of the game, it has no specific knowledge of the combat system except “more is better”. In that extent, it will be content with trying to create as many artilleries as it can, without any tactical after thoughts.
The royal expeditionary force (Figure 22) can vary during the game but starts with 31 infantry units, 15 cavalry units, and 14 artilleries, in the latest version of FreeCol (0.10.5). However when our AI was developed with an earlier build of the game in which the expeditionary force was a much more manageable 10 infantries, 10 cavalries and 10 artilleries. We will be content to have our AI defeating the weaker expeditionary force, and leave victory against the larger opponent as a potential goal for future work.

4.1.2 Tiles and Goods production

Tiles are basic location elements. Each tile has a terrain type, which defines the basic resources output, the possible bonus resources, the cost it takes for a unit to cross it, and the defense bonus it grants unit defending themselves while standing on a tile of that terrain type (Figure 24). A colonist working on a tile or in a building produces a certain number of resources every turn. The resources, or goods, can be split in three categories: basic goods (*FOOD, LUMBER and ORE)* are harvested on tiles and depend of the terrain type. Refined goods (*HAMMER, TOOLS*) are created from basic goods in colony buildings and are used to build other buildings or units. Immaterial goods (*BELLS, CROSSES*) are metaphoric representations of a game concept. They are produced in special buildings and do not require any basic goods to be created.

The basic resource output that we will concern ourselves with, for the most common terrain types are:

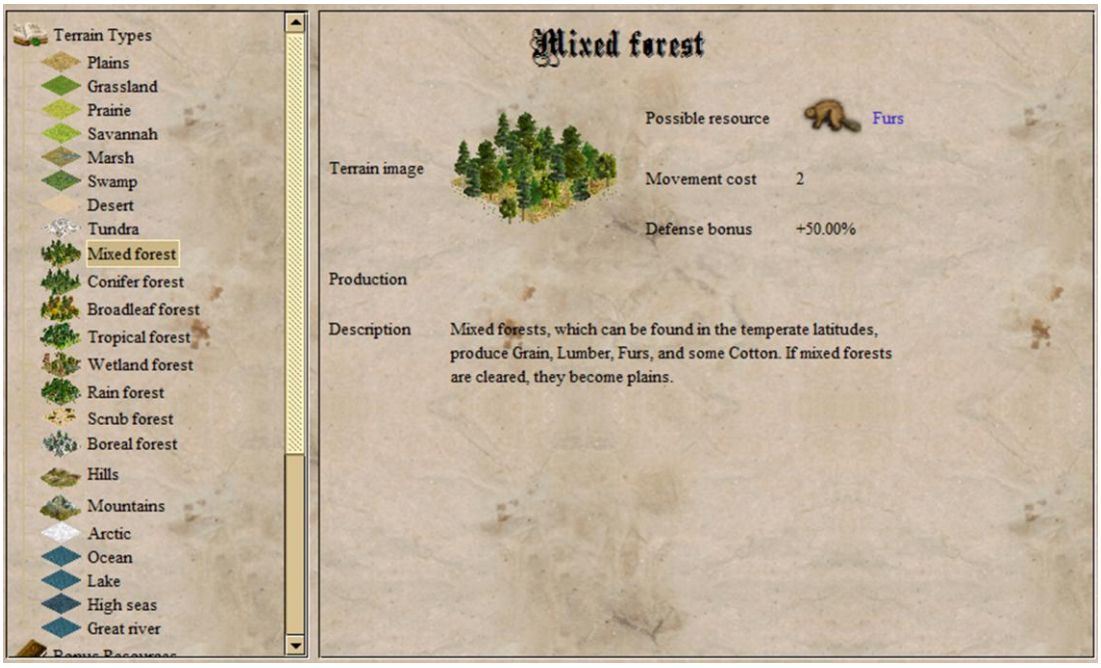

Figure 24 The different types of terrain for tiles in FreeCol

Table 1 Goods production for each terrain type.

Terrain Type	Food	Lumber	Ore	Bonus Resources
Plain	5	0	1	Grain: +2 Food
Grassland	3	0	0	Tobacco
Prairie	3	0	0	Cotton
Marsh	3	0	2	Minerals: + 3 Ore
Desert	2	0	2	Oasis: +2 Food
Mixed Forest	3	6	0	Furs
Conifer Forest	2	6	0	Extra Lumber: +4 Lumber
Tropical Forest	3	4	0	Extra Lumber: +4 Lumber

The table above gives the number of resources that the tile gives each turn when worked by a non-expert unit. The unit has to choose if it wants to work a tile as a farmer (food), lumberjack (lumber) or miner (ore). If there is a bonus resource on that tile, the resource output will be greater. Tobacco, Cotton and Furs are only used to either trade or sell directly or to manufacture cigars, clothes and coats which will then sell for a higher price. Since we don't concern ourselves with trade, those resources are ignored by the AI.
A first challenge of our algorithm will be to allocate units where it is most useful for their kind of work. Prairies and mixed forests give the same amount of food when farmed; however prairies do not give any lumber when worked by a lumberjack, so forest should not be farmed unless there is nothing else available.

4.1.3 Buildings

Beside colony tiles, colonists can also work inside buildings (Figure 25). Each colony starts with a standard set of buildings already constructed. Those are the town hall, the chapel, the depot and the pasture, plus one house for each of the professions that can refine raw goods: the carpenter, the blacksmith, the tobacconist, the weaver, the distiller and the fur trader. Contrary to tiles which can be worked by only one colonist at a time, buildings can usually accommodate up to three colonists (except the depot, chapel and pasture which cannot be worked by any units). Putting a colonist to work in a building will either produce a special type of goods (bells or crosses), or transform a certain number of basic resources (input goods) into refined resource (output goods). For example, the carpenter's house transforms lumber into hammers. Putting a colonist to work at transforming basic resources into refined ones does not do anything if the basic resource is not present in the colony.
On top of the starting buildings, each colony can decide to build additional buildings or to upgrade the ones already in place, which will boost their production. Not all buildings are useful for our goal, so Table 2 resumes which buildings the AI will be concerned with.

Table 2 Buildings available to our AI player

Building	**Input Goods**	**Output Goods**	**Starting building**	**Upgradable**
Town Hall	None	Bells	Yes	No
Carpenter's house	Lumber	Hammers	Yes	Yes
Blacksmith's house	Ore	Tools	Yes	Yes
Armory	Tools	Muskets	No	Yes

Depot	None	None	Yes	Yes
Stockade	None	None	No	Yes
Printing Press	None	None	No	Yes

The depot, stockade and Printing press do not produce anything directly so they cannot accommodate any colonist.

The depot stores the goods produced in the colony. It has a limit of 100 for each kind of material goods except for food where the limit is 200. Non-material goods are bells, hammers and crosses and are not stored in the depot, so they have no limit; we will come back on them later. If a material goods, say lumber, is harvested to the point that it goes over the depot limit, the surplus goods will be wasted. The depot can be upgraded to a warehouse with an increased 200 limit on material goods (the maximum food stays 200).

The stockade increases the protection of the colony, giving a 100% defense bonus to all units inside. It can also be upgraded to a fort when the colony reaches at least a size of 3, then to a fortress, which increases the defense bonus to 150% and 200% respectively.

The printing press cannot be worked by any colonists, but increases the bell output for colonists working in the town hall by 50%. When the colony reaches a size of 4, it can be upgraded to a Newspaper and give a bell production bonus of 100%.

The armory not only allows colonists to transform tools into musket, it is also a prerequisite for the construction of artilleries. This makes it the only building that is mandatory to build if the AI wants to win (the other mandatory buildings are already present when a new colony is founded)

Figure 25 Colonists assigned to work in colony buildings and their goods output

The carpenter's house converts lumber into hammers, at the rate of 3 lumbers for 3 hammers per colonist working in the building. Hammers are used to build military units and other buildings so it is important to maximize their production. The house can be upgraded to a lumber mill when the population in the colony reaches 3. A lumber mill converts lumber into hammer at the rate of 6 lumbers for 6 hammers per colonist working in the building. A maximum of three colonists can work in the building at the same time.

The blacksmith's house is similar to the carpenter's house. It converts 3 ore into 3 tools per colonist working in the building. Tools are a secondary requirement to build artilleries and some building upgrades also require them. The house can be upgraded to a blacksmith's shop to boost production to 6 ore for 6 tools per colonist.

Finally, the town hall is produces liberty bells. It does not need any input goods to produce the bells and a colonist produces 3 bells each turn working in the town hall. Additionally, the town hall produces one extra bell each turn even without the presence of a colonist. The bell production increases when the printing press of the newspaper is built in the colony.

4.1.4 Colony growth

A colony size is determined by the number of colonists working on colony tiles and in colony buildings. Each working colonists requires two units of food per turn. If not enough food is produced to sustain all colonists, a famine occurs and a colonist, chosen at random, is lost. This obviously should be avoided at all costs. On the other hand, each unit of food that is not consumed by the colonist is added to the storage, and once the food storage reaches 200, a new colonist appears and can be assigned to work in the colony. The 200 units of food in the storage are consumed in the process. This is the primary mean to increase the number of units, the other being colonists emigrating from Europe. Since the immigration process has great random variations in the type of units that will be offered, we decided to not incorporate it in our strategy.

4.1.5 Liberty Bells

One of the central concepts in FreeCol is liberty. Liberty bells, named after the famous Philadelphia bell that was supposedly rang to announce the congress's vote for independence on July 4th 1776, are what measures liberty in FreeCol. Each colony town hall produces one bell by default. The player can also assign colonists to work in the town hall to increase bells production. For each colony, a *Sons of Liberty* (*SoL*) percentage is computed based on the total number of bells $Bells_{total}$ in the colony and on the colony size $Size_{colony}$:

$$SoL = \frac{Bells_{total}}{Size_{colony} * 200} * 100 \qquad (28)$$

The *SoL* percentage is used for two things. First a player cannot declare independence if all his colonies do not have a *SoL* percentage above or equal 50%. Thus it is a mandatory condition for the AI to be able to increase the colony's *SoL* percentage to at least that level by the end of

the game. Second it is used to compute how many rebels and how many royalists are present in the colony (Figure 26)

$$Rebels = Size_{colony} * SoL \quad (29)$$

$$Royalists = Size_{colony} - Rebels \quad (30)$$

The ratio of Rebels vs. Royalists is used to compute a production modifier Mod_{prod} for the colony. Having 6 royalists or more over the number of rebels causes a penalty to the production of all workers.

$$Mod_{prod} = \begin{cases} -2 & if\ (Royalists\text{-}Rebels > 10) \\ -1 & if\ (10 \geq Royalists\text{-}Rebels > 6) \\ 0 & if(6 \geq Royalists\text{-}Rebels > 0) \\ +1 & if\ (0 \geq Royalists\text{-}Rebels)\ and\ (Royalists > 0) \\ +2 & if\ (Royalists = 0) \end{cases} \quad (31)$$

This production modifier applies to each working colonist and is essential to having an efficient colony. If adding a colonist decreases the production modifier, often time the added production from the new colonist is not enough to compensate for the loss of production for all units.

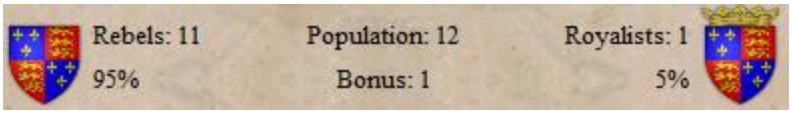

Figure 26 Rebels and Royalists in a colony of size 12, with the associated production bonus (Bonus: 1)

Additionally, bells are consumed each turn according to the following equation

$$Bells_{consumed} = \max((Size_{colony} - 2), 0) \quad (32)$$

In other word, each colonist over two will consume one liberty bell each turn. Optimizing the bells production in function of the size of the colony is critical to winning the game.

4.1.6 Colony Production

Colony production is defined in term of hammers. Hammers are immaterial goods, just like bells, so they do not need to be stored in the depot. A colony can use hammers to build new buildings or upgrade the ones already present, or create new military units (artillery). New and upgraded buildings gives different bonuses (see 4.1.3). Some constructions also require tools in addition to hammers. Contrary to hammers, tools have to be stored in the depot, so they are limited to maximum 100 (or 200 if the colony has upgraded the depot to a warehouse) at any given time. Extra tools generated by the colony are wasted, so an efficient colony produces them in burst, whenever a construction project requires it. Table 3 shows for each item their cost in hammer, tool, and if the prerequisites that need to be met before it can be placed in the production queue.

Table 3 List of buildable items, with their production costs

Item	Hammers	Tools	Prerequisite/Notes
Artillery	192	40	Armory is required to build artilleries
Armory	52	0	None
Lumber Mill	52	0	Colony size ≥ 3. Replaces Carpenter's house
Blacksmith shop	64	20	Replaces Blacksmith's house
Stockade	64	0	None
Fort	120	100	Colony size ≥ 3. Replaces Stockade
Fortress	320	200	Colony size ≥ 3. Replaces fort. Also requires warehouse since it needs to accrue 200 tools.
Warehouse	80	0	None
Printing Press	52	20	None
Newspaper	120	50	Colony size ≥ 4. Replaces Printing Press.

4.2 Summary of the AI tasks

Now that all important game concepts have been explained, we can summarize the functionalities we want our AI to have, in order of importance:

1. The colony needs to reach the year 1776 with at least 50 % *Son of Liberty* to be able to declare independence.
2. The colony needs to have the best defense possible to resist the attack of the royal expeditionary force.
3. The colony needs to allocate workers to
    a. Foster growth while minimizing famine
    b. Keep the production modifier has high as possible
    c. Insure the correct supply of basic goods to be transformed into refined goods
    d. Insure the highest production of hammers over the course of the game
4. The colony needs to assign items to the build queue in order to sustain the above mentioned goals.

Chapter 5. Artificial Player implementation

The implementation connects three different mechanisms. The foundation is the response threshold model that was presented in Chapter 2. This allows the AI to dynamically allocate workers to tiles or building. The different weights of the model are tuned with an evolutionary algorithm similar to (Campos et al., 2000). The build queue sequence is also created with the same evolutionary process. Finally, we try to give some sense of planning to the colony, first by trying a multi-layered response threshold approach, then by augmenting the original algorithm with some simple rule-based planning information.

5.1 Dynamic Task Allocation

Our main goal is to allocate colonists to goods production such that goals 1 to 4 described in 4.2 are met in the best possible way.

We consider that the colony emits stimuli to indicate its need for the different goods. Goods that are considered are {*FOOD, BELLS, LUMBER, HAMMERS, ORE, TOOLS, NONE}.* The *NONE* stimulus is a way for the colony to indicate that it cannot support more colonists without incurring a penalty to the production due to a low liberty value (see equation (31)).

5.1.1 Computing the response probabilities

For each unit *u*, for each goods type *g*, we define a threshold $\theta_{ug} \in [\theta_{min}, \theta_{max}]$. We also define a stimulus S_g for each goods type *g.* At each turn, the probability *P(u,g)* that unit *u* is assigned to produce goods *g* is

$$P(u,g) = \frac{\beta_g S_g^2}{\beta_g S_g^2 + \theta_{ug}^2} \qquad (33)$$

Where β_g are scaling factors. This is very similar to the original equation (1) but in addition, our problem requires that we also balance each stimulus relative to one another. In the literature examples that we presented, there was no case where agents are exposed to competing stimuli simultaneously. Indeed in the paint booth problem, while there is a different stimulus for each color, only one truck comes out of the chain at each time; and if one booth responds to more than one stimulus over several time units, the trucks are lined up in the booth queue. Here our agents have to choose to answer to one of several competing stimuli each turn. To simulate that the colony has a higher need for food than for lumber for example, the food stimulus should be larger than the lumber. How much larger is a matter of experimentation, and we will use the evolutionary algorithm to determine what is the best ratio between all stimuli which is the role of the β_g parameters.

Conversely, each unit has a probability *p* of stopping their actions, following equation (2) which was introduced in 2.1

$$Pi\ (X = 1 \rightarrow X = 0) = p \qquad (2)$$

5.1.2 Computing the stimuli

The stimuli for each goods are user defined based on expert knowledge. They were chosen to be as simple as possible so that the behavior of the algorithm relies on emerging behaviors rather than intricate fine tuning based on the game mechanics. For example, the food stimulus takes only the food production into account, and not the fact that creating a new colonist might decrease the production modifier.

We fix the following rules to compute each stimulus:

FOOD The food stimulus is a function of the food consumed and the food produced every turn. Recall that a colony consumes two units of food per working colonist, while colonists idling outside the colony do not consume anything. The food produced is the sum of the production for each farmer, and is dependent on the tile farmed, the expertise of the colonist and the production modifier. Let $Surplus_{FOOD}$ be the difference between the food produced and the food consumed. We want to minimize the number of famine episode and promote a reasonable growth, but not excessive. The food stimulus S_{FOOD} takes the following values.

$$S_{FOOD=}\begin{cases} 100 & if\,(Surplus_{FOOD} < 0) \\ S_{FOOD} + 1 & if\,(0 \leq Surplus_{FOOD} < 6) \\ 1 & if\,(Surplus_{FOOD} \geq 6) \end{cases} \quad (34)$$

BELLS The bell stimulus is a function of the bells consumed and the bells produced every turn. Recall that each working colonist over the initial two consumes one bell, while the bell production if function of how many colonists working in the town hall. Let S_{BELLS} be the bell stimulus and $Surplus_{BELLS}$ be the bell surplus. We want to have a steady bell production to be able to benefit from a positive production modifier while keeping the colony growing. Also, we want to make sure the *Son of Liberty (SoL)* percentage, which is the amount of freedom experienced in the colony (see 4.1.5), will be high enough for us to be allowed to declare our independence. Since the town hall can only accommodate three colonists at the most, we define S_{BELLS} to be 0 if the town hall is full. In other cases, we have

$$S_{BELLS} = \max(0, (15 - Surplus_{BELLS})) \quad (35)$$

We aim to have a 15 bell surplus. Since that number is actually not reachable without production modifier, this will steer the colony towards allocating workers in that direction.

LUMBER The lumber stimulus is computed simply by looking at $Storage_{LUMBER}$, the current amount of lumber stored, compared with the amount that we'd need to produce the item currently in the build queue. Built items require a number of hammers to be built, and hammers can be obtained from lumber at a ratio of one-to-one. So if an item requires 52 hammers, we will need to harvest 52 units of lumber first. Let S_{LUMBER} be the stimulus for lumber, and $Build_{hammers}$ be the

hammer requirement for the current item.

$$S_{LUMBER} = \begin{cases} 0 & if(Storage_{LUMBER} \geq Build_{HAMMERS}) \\ 10 & if(Storage_{LUMBER} < Build_{HAMMERS}) \end{cases} \quad (36)$$

HAMMERS The hammers stimulus is computed similarly to the lumber stimulus, by looking at the requirements of the production queue. However, to prevent having colonists trying to produce hammers without having the lumber requirement on hand, we add an exception for that case. Let $S_{HAMMERS}$ be the hammers stimulus and $Storage_{HAMMERS}$ be the amount of hammers accrued so far in the colony; we define it to be 0 if the carpenter's house is full. In other cases we set it to be:

$$S_{HAMMERS} = \begin{cases} 0 & if \begin{pmatrix} Storage_{HAMMERS} \geq Build_{HAMMERS} \\ or\ Storage_{LUMBER} \leq 15 \end{pmatrix} \\ 10 & if\ not \end{cases} \quad (37)$$

ORE The ore stimulus follows the same pattern as the lumber stimulus, but taking into account the tools requirement instead of the hammer requirement. Let $Storage_{ORE}$ be the amount of ore units stored in the colony, let $Build_{TOOLS}$ be the requirement in tools for the current build item, the ore stimulus S_{ORE} is defined as

$$S_{ORE} = \begin{cases} 0 & if(Storage_{ORE} \geq Build_{TOOLS}) \\ 10 & if(Storage_{ORE} < Build_{TOOLS}) \end{cases} \quad (38)$$

TOOLS The tools stimulus is computed in a similar fashion to the hammer stimulus. Let S_{TOOLS} be the tool stimulus and $Storage_{TOOLS}$ be the number of tools currently stored in the colony. We define S_{TOOLS} to be 0 if the blacksmith's house is already working at maximum capacity. In other cases

$$S_{TOOLS} = \begin{cases} 0 & if \begin{pmatrix} Storage_{TOOLS} \geq Build_{TOOLS} \\ or\ Storage_{ORE} \leq 15 \end{pmatrix} \\ 10 & if\ not \end{cases} \quad (39)$$

NONE As mentioned in 4.1.5, a colony suffers a penalty for having a large number of working colonists before enough bells are accrued. Equations (28) to (31) describe how the production modifier is computed. Let's demonstrate the need for the 'none' stimulus with a simple example: A colony has 6 colonists working as farmers, producing 3 food units each for a total of 18. Since no one is working in the town hall, the number of bells in the colony is 0, and so is the *SoL* (equation (28)). This means there is 6 royalists in the colony and 0 rebels, according to equations (29) and (30), and from equation (31) we can see that the production modifier is 0. Adding a colonist in the town hall will bring the colony size to 7, and the number of royalists to 7 as well. Now equation (31) shows a production modifier of -1. Not only does the food production drop to (3-1)*6=12, which is

not enough to sustain 7 colonists, but also the colonist in the town will not have any effect, since his bell production would be (3-1) = 2, which is not enough to offset the 5 bells consumed by a colony of size 7. To avoid situations like this, we add a stimulus for doing nothing, until the number of royalists drops enough to not be subjected to a production penalty. Let S_{NONE} be that stimulus, we compute it the following way:

$$S_{NONE} = \begin{cases} S_{NONE} + 5 & if \begin{pmatrix} SoL < 50 \\ and\ Royalists - Rebels > 6 \end{pmatrix} \\ 0 & if\ not \end{cases} \quad (40)$$

5.1.3 Computing the thresholds

Before going further, we want to stress the difference between a specialist and an expert. A specialist is a unit that has a low threshold for a job, and thus is more likely to keep doing this job turn after turn. This is the mechanism from the threshold response model. An expert is a FreeCol game concept that grants bonuses to the units when they are working in their area of expertise. While we will make it so that every expert is a specialist and has a low threshold for their field of expertise, not every specialist becomes an expert.

Now, in evaluating the thresholds of the colonists for the different tasks, we want to achieve two distinct effects: First, expert workers should answer with great probabilities to even small stimuli in their area of expertise. Second, we want to encourage workers to specialize and stick to a given job for extended periods of time. This gives a chance to units working tiles to become experts, and this prevents units working in buildings from switching to tile job at the first incentive.

We will correlate the specialization to the experience of a unit, and thus, even though normally in FreeCol units working in buildings do not accrue experience, we allow each unit to do so. However we keep the restriction that they cannot become expert in jobs worked in buildings, so this extra experience has no effect on the game mechanics. The experience level $Exp \in [0, 200]$ of each unit is used to compute the thresholds. Exp is such that for job $j \in J$, the set of all possible jobs, if $Exp_j \neq 0$, then for all jobs $k \in J, k \neq j$, $Exp_k = 0$. In other words, a unit can only accrue experience for one job at a time, and switching jobs resets their experience to 0.

We define θ_{ug}, the threshold unit u has for producing goods g as follow:

$$\theta_{ug} = \begin{cases} 0.1 & if (u\ is\ expert\ for\ g) \\ \min(10, \left\lfloor \frac{200 - Exp_g}{20} \right\rfloor + 1) & if\ not \end{cases} \quad (41)$$

Thus, a unit will have a threshold for goods g in [1, 10] if it is not an expert, and 0.1 if it is. Additionally, since we want idle units to be sensitive to low stimuli, we lower their thresholds to 1 for all stimuli except the eventual one in which they are expert which stays at 0.1.

5.1.4 The allocation loop

Now that we defined all the quantities involved in equation (33), we can detail the allocation algorithm (Algorithm 1). First, we test all units to see if they will continue their task or stop,

Algorithm 1: organizeColony

```
begin
    For each unit u
        Stop test      (equation (2))
        Move idle units to the outside colony location
    end (For each)
    For each stimulus S
        Update S      (equations (34) to (40))
    end (For each)
    For each unit u in outside colony
        For each stimulus S
            Response test      (equation (33))
        end (For each)
        if more than one successful Response test
            new_job = the one with highest stimulus/threshold ratio
        else
            new_job = successful Response test, or NONE if unit did not re-
            spond to any stimuli
        end (if)
        Find best location for new_job
    end (For each)
    Update SoL          (equation (28))
    Update production modifier     (equation (31))
end
```

according to equation (2). If a unit stops working, we move it outside the colony. We then update each stimulus. For food and bells, which depend of the colony size, we take the total number of units, including the idle ones outside the colony so we can try to maximize the number of units working.

Then for each unit idle outside, we test if they respond to one or more stimuli. Units who respond to more than one stimulus are assigned to the one which has the highest stimulus to threshold ratio. Thus, a specialized unit has the most chance of answering to its area of expertise, even if the stimulus is not the highest. Once the unit's response has been decided, we assign it to the best location for its new job. This is a deterministic process as all goods outputs for all locations are known in advance, and our algorithm just picks the best location available for the job.

This organizeColony algorithm runs at the beginning of each turn, followed by the computation of goods production, goods consumption, population updates and finally construction completion.

5.2 Evolutionary Algorithm

Another important part of our AI is the evolutionary algorithm we use to find the most performing parameter values. Among all the different machine learning approaches we could have tried to learn the weights of the different parameters, this seemed the most fitting as it is through the process of evolution that social insects came to be as they are, with their finely tuned emergent behaviors.

5.2.1 Solution encoding

We construct a genome of two separate chromosomes: one to evaluate the seven θ_g, the other to find the optimal build queue order. Although we could have included the parameter p from equation (2) and the θ_{min} and θ_{max} delimiting the search interval for θ_g in the first chromosome, we chose handpicked values for those to speed up the learning process.

For the first chromosome, we take inspiration from (Campos et al., 2000) (see 2.3); but instead of transforming each parameter into a bit sequence, we keep their decimal form. We end up with a series of seven decimal values representing the scale factors for the FOOD, BELLS, LUMBER, HAMMERS, ORE, TOOLS and NONE stimuli.

For the second chromosome, we construct it from each constructible building's sequence number in the build queue. Those buildings are the armory, the lumber mill, the blacksmith's shop, the stockade, the fort, the warehouse, the printing press and the newspaper. We choose to not consider the fortress so we can keep an even number for the crossover. Once we know the position in the build queue of those buildings, we can then construct the actual build queue by filling every empty spot until spot number 50 in the queue up with an artillery.

5.2.2 Genetic operators

The crossover operation is a one point crossover for chromosome 1 and a uniform crossover for chromosome 2. Figure 27 shows how the crossover operation is applied to chromosome 1.

θ_g	FOOD	BELLS	LUMBER	HAMMERS	ORE	TOOLS	NONE
Parent A	0.199	0.876	0.372	1.665	0.593	1.321	1.035
Parent B	1.420	1.002	0.029	1.932	0.305	0.978	0.038

Child AB	1.420	1.002	0.372	1.665	0.593	1.321	1.035
Child BA	0.199	0.876	0.029	1.932	0.305	0.978	0.038

Figure 27 One point crossover for chromosome 1, with a crossover point located between BELLS and LUMBER

The second chromosome is trickier to cross. We take example on (Liu, Abdelrahman, & Ramaswamy, 2005) and their genetic algorithm for the single machine total weighted tardiness scheduling problem. We randomly choose half the buildings from parent A to keep their sequence in the child's queue. We then take the buildings from parent 2 in order and attempt to fit them in the queue. If there is a conflict, we push the building back until we find an empty spot (which would otherwise be occupied by an artillery). If no empty spot could be found before the end of the queue is reached (for example, two buildings vie for spot 50), we wrap around until we find an empty spot. Figure 28 shows how the crossover operation is applied to chromosome 2.

We can see immediately that a large range of chromosomes will give very poor results since some of the build items have prerequisites. For example, if there is no building in position 0, the start of the build queue will be occupied by an artillery. However, artilleries cannot be built without an armory so construction will stall and nothing will be built in this colony, re-

Building	Armory	L.Mill	Bl. shop	Stock.	Fort	Ware.	Pr.Press	News.
ParentA	0	1	6	32	12	7	24	45
ParentB	9	2	12	4	7	49	17	33

ChildAB	0	1	**13**	4	12	49	17	45
ChildBA	9	2	6	32	7	**8**	24	33

Figure 28 Uniform random crossover for chromosome 2. Armory, Lumbermill, Fort and Printing press keep their sequence, the other buildings are swapped. The blacksmith shop in childAB and the warehouse in child BA are then corrected to remove sequence conflicts.

sulting a defense score of 0. Likewise, colonies that tries to build a newspaper before a printing press, or a fort before a stockade will fail. Finally, we can note that buildings with a position beyond 30 in the build queue have very little chance of ever being built, because the game usually ends with around 20 or 25 items built in total. Buildings that rank consistently low in the output of the evolutionary algorithm can be considered of limited value for the goal we try to achieve.

The second genetic operator that we apply is the mutation. At each turn, each gene in each chromosome (i.e. the parameters in chromosome 1 and the positions in the build queue for chromosome 2) has a 1% chance of mutating. We keep this parameter low since the genome has 15 genes and we do not want to have all genomes experience one mutation or another.

5.2.3 Fitness

We evaluate the fitness based on two criteria: the SoL percentage and the defense value. The SoL percentage has to be at least 50% to be able to declare victory, so we can already disregard all colonies that do not achieve that result. The defense value is a function of the number of artilleries A and the bonus $Modif_{defense}$ from defensive buildings (the stockade and the fort). We end up with the following formula:

$$Fitness = [(SoL \geq 50)?\,1:0] * A * Modif_{defense} \qquad (42)$$

Since the task allocation process we want to evaluate is stochastic, we run each set of parameters 50 times and we take the average fitness to be the fitness of a given parameter set.

5.2.4 Initial population

We start with 200 individuals. Since the state space is quite large, we direct the evolution by setting the armory as the first building in the queue. This allows all the first generation individuals to build at least some artillery and avoid a fitness score of 0 for the large majority of the generation.

5.2.5 Selection

We run our algorithm for 50 generations. We use a mix of linear ranking selection and tournament selection to choose 100 individuals who will reproduce but also move on to the next generation. The first 50 chosen are the 50 best genomes according to their fitness. The next 50 are the winners of 50 random tournaments between the 150 remaining individuals. From

that pool of 100 parents, we create 50 random pairs for crossover. Each individual is part of one and only one pair. We generate two children from each crossover (child AB and child BA from parent A and parent B). This creates 100 children, which are added to the parent pool to become the next generation. In addition, we also save the best two individuals of each generations to preserve them from unwanted mutation, and we re-inject them in the next generation's pool.

5.2.6 Results

We run the evolutionary algorithm with the following handpicked values for the three parameters that we did not include in chromosome 1:

$$p = 0.2$$
$$\theta_{min} = 0.0$$
$$\theta_{max} = 2.0$$

Since there is a lot of randomness both in the game mechanics and in our allocation process, we run each individual 100 times and average the results. The best individual after 50 generations ends up with the following chromosomes

Table 4 Optimized parameters for building the best defense

Chromosome 1	
θ_{FOOD}	0.319
θ_{BELLS}	0.766
θ_{LUMBER}	0.171
$\theta_{HAMMERS}$	1.479
θ_{ORE}	0.429
θ_{TOOLS}	1.374
θ_{NONE}	1.499
Chromosome 2	
Armory	1
Lumber mill	2
Blacksmith's house	22
Stockade	6
Fort	8
Warehouse	28
Printing Press	3
Newspaper	32

Intuitively, we see that the values in chromosome 2 are not completely optimal. Indeed swapping the positions between the armory and the lumber mill would end up building the armory slightly faster, but the difference is minimal and we're happy with keeping this artefact of our initial population seeding. We note that the blacksmith's house, the newspaper and the warehouse end up having little chance of being built, since they are far back in the queue and the player might not have the time to get to building them.

5.3 Hierarchical response threshold layers

If we look at an expert player tactic for this game, we realize that he starts by focusing on just FOOD and BELLS in the early game before diversifying to other tasks when he gets enough workers to fulfill the basic needs. This is something that our player would not be able to do, as the stimuli and thresholds do not take the size of the colony into account. We will try to give our player a similar sense of planning, and see if it performs better with that added information.

We compare two ways to achieve that effect: Our first approach is to add an additional threshold layer that can be used to steer the colony towards a set of more growth friendly stimuli or towards a set of more build friendly stimuli. We can of course imagine adding other stimuli in this additional layer, such as a stimuli to focus on trade goods, but this is beyond the scope of our tests.

The second approach will be to incorporate the same information directly in the first response threshold layer, by making slightly more complex rules for computing the stimuli. Since from the start we have tried to avoid this, we only test this approach to have a point of comparison with the multi-layered approach.

5.3.1 Colony response thresholds and associated stimuli.

Throughout chapter 5, the colony existed only as a product of the collective behavior of the units. We now make the colony a being of its own, and subject it to a self-generated stimulus: the desire to grow. We formalize it by defining $S_{c,\ GROWTH}$, the self-generated stimulus that incites colony *C* to keep growing, and $\Theta_{c,\ GROWTH}$, the associated threshold. It should be noted that this stimulus does not have to be self-generated, and in a complete game with multiple colonies, we can imagine that this stimulus is a function of the other colonies' size or some other entity. But since we limit our scenario to just one colony, it's left to produce its own stimulus. Let $S_{C,\ GROWTH}$, the growth stimulus be

$$S_{C,GROWTH} = \Gamma \tag{43}$$

while the associated threshold is defined as a linear function $Size_C$, the size of colony c

$$\Theta_{C,GROWTH} = \delta * Size_c \tag{44}$$

We use $S_{C,GROWTH}$ and $\Theta_{C,GROWTH}$ in the same probability function as before

$$\mathrm{P}_C = \frac{S_{C,GROWTH}^2}{S_{C,GROWTH}^2 + \Theta_{C,GROWTH}^2} \tag{45}$$

P_C is defined as the probability that the colony will choose to focus on growing, as opposed to focus on building. With this equation, if the colony is small it has a low growth response threshold and should be inclined to use the growth friendly stimuli, while larger colonies will have a high thresholds and will rarely answer to the growth stimuli. So to make this work, we define two sets of scale factors for the goods stimulus detailed in the previous chapter: A set

of scale factors to foster growth, and another set of scale factors to optimize production. We take for the latter the results from our evolutionary algorithm (Table 4). For the former, we modify the fitness function from (42) to reward growth:

$$Fitness_{GROWTH} = [(SoL \geq 50)?\,1:0] * Size_C \quad (46)$$

We also modify the chromosome 1 to be shortened to FOOD, BELLS and NONE, while we zero out the other stimuli scale factors, and we run the evolutionary algorithm again. Here, chromosome 2 has no impact since nothing will be produced. We obtain the following values:

Table 5 Optimized scale factors for growth

Chromosome 1	
θ_{FOOD}	0.577
θ_{BELLS}	1.306
θ_{NONE}	1.893
$All\ other\ \theta_g$	0.0

We call the scale factors from Table 4 F_{PROD} and the ones from Table 5 F_{GROWTH} and we add the following algorithm before algorithm 1: organizeColony()

Algorithm 2: planColony

begin

Update growth plan thresholds (equation (44))
Response test (equation (45))
if Response test is positive, $\beta_g \leftarrow F_{GROWTH}$
else $\beta_g \leftarrow F_{PROD}$

end

Now, the colony can switch between growth oriented scale factors and production oriented scale factors. Since the thresholds for growth is linear with the size of the colony, we expect growth to be prevalent when the size is small and production to become more and more common as the colony grows in size. We can now run our evolutionary algorithm one last time to evaluate appropriate values for Γ and δ. We place them in a third chromosome (although the first two chromosomes are fixed at this point) and we use a simple one point crossover to find correct values. We obtain the fittest individual using the following values:

Table 6 Optimized parameters for the upper layer response threshold

Chromosome 3	
Γ	9.649
δ	2.684

Plugging in those parameter in equations (43) and (44), the fittest individual from the GA was about 10% better than the fittest individual of the single layer approach. We give more detailed results in Chapter 6.

5.3.2 Rule based planning

To compare the performance of the two-layer approach, we build a simple rule directly into the first layer stimuli. We modify, the LUMBER, HAMMERS, ORE and TOOLS thresholds by adding a condition that the colony size has to be at least 6 (an arbitrary number based on our expert knowledge of the game) to have a non-zero stimulus for those 4 goods. For example, the LUMBER stimulus from equation (36) becomes:

$$S_{LUMBER} = \begin{cases} 0 & if \begin{pmatrix} Storage_{LUMBER} \geq Build_{HAMMERS} \\ or\ Size_C < 6 \end{pmatrix} \\ 10 & if\,(Storage_{LUMBER} < Build_{HAMMERS}) \end{cases} \quad (47)$$

This means the colony will split the workers only according to the FOOD, BELLS, and NONE stimulus until there are at least 6 workers in the colony.
We then run again the evolutionary algorithm to evaluate chromosome 1 and 2 with the modified stimuli. We come up with the following values:

Table 7 Optimized parameters for modified single layer threshold

Chromosome 1

θ_{FOOD}	0.359
θ_{BELLS}	0.869
θ_{LUMBER}	0.899
$\theta_{HAMMERS}$	1.888
θ_{ORE}	1.379
θ_{TOOLS}	1.603
θ_{NONE}	1.635

Chromosome 2

Armory	1
Lumber mill	2
Blacksmith's house	4
Stockade	3
Fort	6
Warehouse	27
Printing Press	2
Newspaper	22

This method also brings significant improvements in results, with the best individual being on par with the one from the multi-layer approach. We also notice slight changes in both chromosomes compared to our first approach. This can mean either that our new setup calls for a different equilibrium of values, or (most likely) that several set of values can produces similar results.

In conclusion, we see that our basic response threshold system can easily be tuned out with either of the two approaches proposed in this section. We also see however that the tuning of the algorithm becomes increasingly complex as we add more and more parameters and some ad hoc choices were made that are not guaranteed to be optimal.

For example, for the multi-layer approach, we decided to pre-train our first layer for two different goals: food, and production. But who is to say that the set of production scale factors we chose isn't flawed? Indeed they were obtained from training over the complete episode, including the first phase when the colony should optimally focus on growth. So it would be interesting to optimize it over an episode that begins not with a basic colony, but rather with a colony that has already accumulated a significant size. This would bring forth another set of questions on what the colony size, the amount of bells already accumulated, and other parameters should be before start this new tuning episode…so we leave those considerations for potential further research.

Chapter 6. Result Analysis

We now present a quantitative study of the performances of our algorithm, and compare it with the two improved approaches presented in 5.3. First, we define a basic test scenario, and we then present some results gained from an expert playing this simple scenario, which we'll use as a base of comparison to gauge the quality of our AI player.

6.1 Test scenario

For our basic scenario, we start in year 1492 with a single colony, Brussels, and two colonists (Figure 29). This is more or less how any game of FreeCol would start, except we skipped the first few turns where we look for a suitable spot for our first colony. We have stripped the

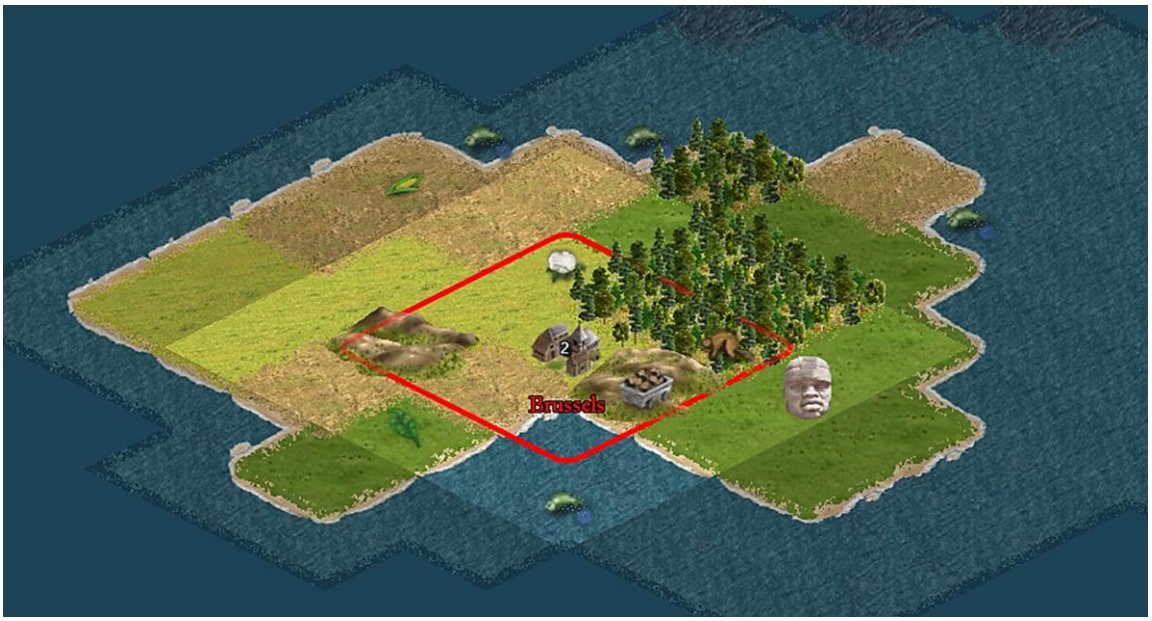

Figure 29 Map of the basic scenario at turn 1.

map of all opponents, so we do not have to worry about fighting for control of the territory and building up defenses early.

The colony is surrounded by a good mix of exploitable terrains (within the red borders in Figure 29), typical of any game, and ensuring a reasonable supply of basic resource: One plain and two prairies which can be exploited for food, two forests for lumber, and two hills for ore. The plain is the obvious first choice for food as it produces 5 FOOD compared to the prairies which produce only 3. Likewise one of the two hill tiles will be the preferred choice for ore mining as it has extra resource, giving it a production advantage of 6 to 4 for the other hill tile. The two forests on the other hand have equal lumber production value. The last tile is an ocean tile, which cannot be exploited.

We play 460 turns and then attempt to declare our independence. From 1492 to 1600, each turn is equal to a year. From 1600 onward, two turns take place every year, one for the Spring season and one for the Fall season, so 460 turns will conveniently take us up to 1776, the year of the American declaration of independence. During those 460 turns, we ignore all decisions

unrelated to our target problem, such as tax increases, immigration opportunities or selection of founding fathers for our colonial congress.

6.2 Expert player

Although this scenario is already complex as it is for an AI player to handle, it is relatively simple for a human with a good familiarity of the game. Figure 30 shows the evolution of

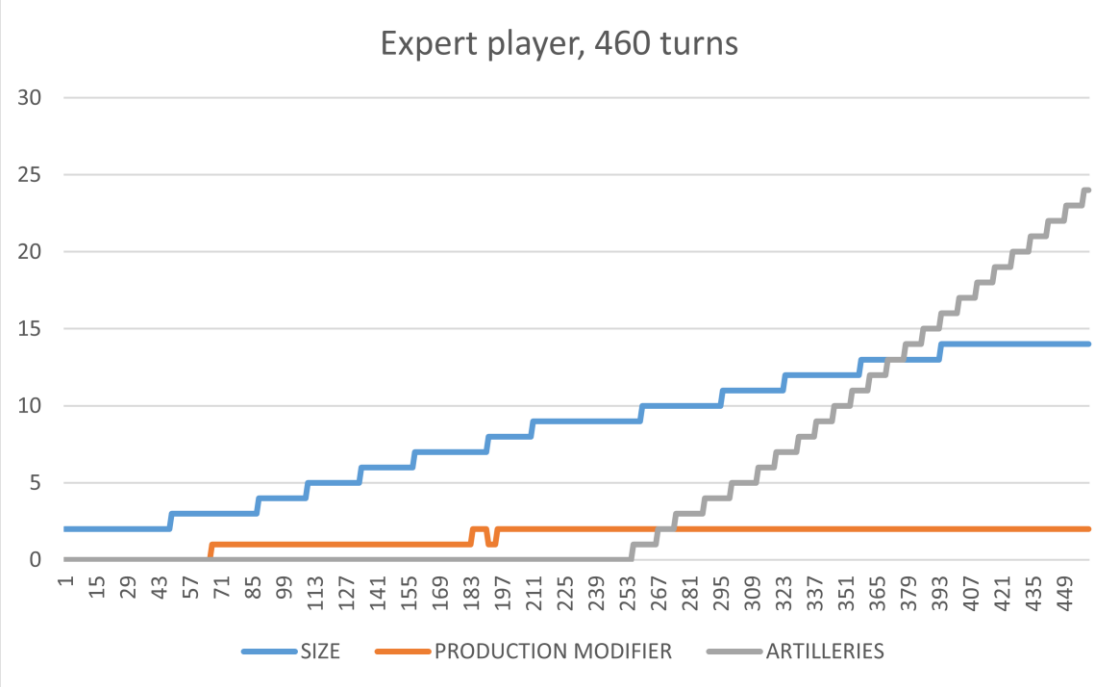

Figure 30 Evolution of the colony size, the production modifier and the number of artilleries during a game by an expert player

three benchmark game parameters: the size of the colony, the production modifier and the number of artillery. We see that the population increases at a relatively slow pace, designed to keep the production modifier at the maximum level of +2. This modifier drops only once, around turn 197, when the colony grows from size 7 to size 8. The production of artilleries, although starting quite late in the game, is very steady, at the rhythm of one every 8 to 12 turns. The expert player finished the 460 turns with a colony size of 14 and 24 artilleries. If we had to attribute it a score similar to the one we use for the fitness function of our evolutionary algorithm, this player would achieve a total score of 48 (24 artilleries * 2 defense bonus modifier).

If we look in detail at the task allocation during the expert game, it is pretty straightforward (Figure 31). The first workers are split between food and bell production, until the town hall is full. Then at size 6, the player starts to allocate the newly created workers between lumber and hammers first, then between ore and tools, since build items require less tools than hammers. A few phenomena indicative of intelligent planning should be highlighted: First, the workers do not spread across all jobs at the beginning of the game, but focus on food and bells to bootstrap the colony growth. Second, when the 6th worker appears around turn 130, it

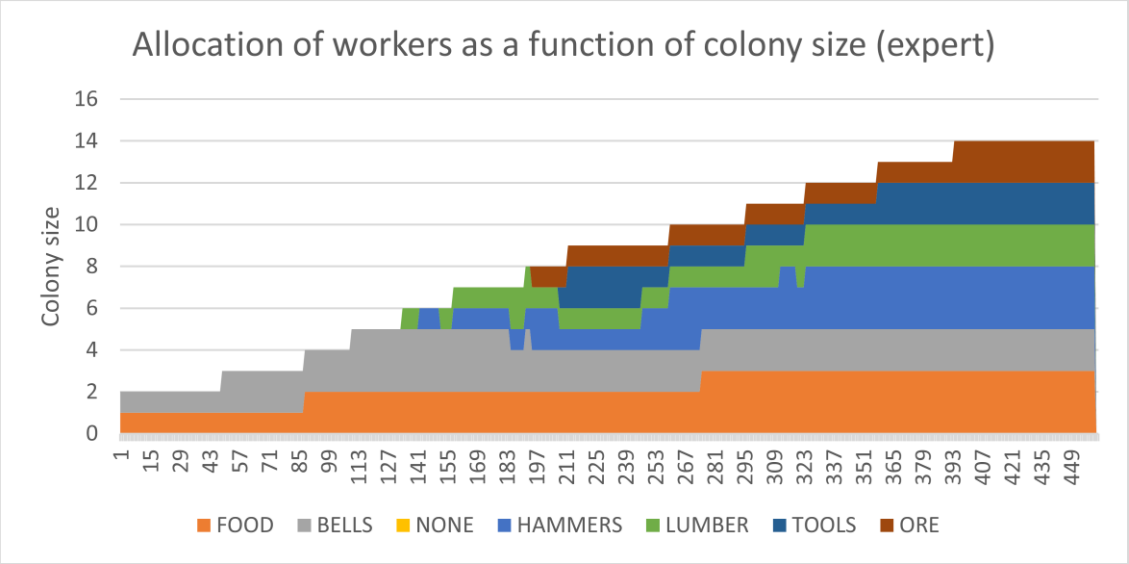

Figure 31 Allocation of the workers by an expert player, as a function of the colony size

oscillates between producing lumber, then transforming this lumber into hammers. This is the only way to balance producing enough lumber to transform into hammer while staying within the boundaries of how much lumber can be stored at any given time (100 units maximum). Third, workers are reallocated once their job is no longer needed, such as the third worker in the town hall that is transferred to producing hammers around turn 185 because the production modifier is already maxed out at that time. We will try to see if our AI player demonstrates the same behaviors.

6.3 AI player: single layer response threshold

Before we can look at a typical game, we need to find out how reliable our player is, since we're dealing with a lot of stochastic processes. Using the optimal settings obtained in 5.2.6, we make our AI player play the same game 100 times and we record several indicators in relation with the goals defined in 4.2: We look at the score, the colony size, the number of military units, the *SoL* percentage and the number of famine episodes.

Table 8 Statistics for different game indicators, over 100 runs by the AI player. 95% interval of confidence displayed with the mean

	Score	Size	Military	*SoL* %	Famine
MEAN	31.34 ± 1.28	15.12 ± 0.85	15.99 ± 0.62	91.57 ± 2.39	0.18 ± 0.09
MEDIAN	34	14	17	100	0
LOW	4	8	3	50	0
HIGH	42	35	21	100	2

Those numbers are encouraging. The score is not really relevant for us, as it is a composite of several parameters. The colony size is more interesting, as it is just slightly higher than what our expert player ended up with so there does not seem to be a disproportional focus on growth. The military score however is the best indication of the quality of the AI player, since having the highest defense possible was one of our goals. We see that on average, our AI player builds 16 military units, which is 66% of the expert's. And while the maximum goes up

to a respectable 21 units, or 88%, we see that the lowest score, at only 3 units, is critically bad. The *SoL* percentage on the other hand, which was the other primary goal of the AI, passes the test every time. The last indicator, the number famine episodes, is reasonably low. On average, only one game out of five experiences a famine episode. While a human player would probably never experience one of those if he's careful because it can easily be fixed, we cannot expect a stochastic process to never make this mistake unless we include some categorical deterministic rules to prevent those.

Let's now look at a typical game in details to see which areas are performing as anticipated and which are lacking. In this game, the AI finished with a colony size of 11, 100 *SoL* and 15 artilleries. Figure 32 gives us a first clue on the lower performances of the AI player compared to the expert player. We see in (A) that the expert's colony grows slightly faster, which in the end will mean more workers available to build artilleries. (B) shows that the evolution of the production modifier is similar for both players, so there's no apparent defect here. Finally, (C) shows that the production of artilleries starts much earlier but at a slower pace. The reason for the earlier start is that the expert player delays building artilleries until the 8 buildings available are built, while the build queue that was optimized for the AI player by the evolutionary algorithm has already an artillery in 3rd position, right after the armory and the lumber mill are built. It then keeps alternating between building a couple artilleries and the remaining buildings. Since the remaining buildings (stockade, fort and printing press) have no direct incidence on the building of the artilleries, the build queue order cannot be blamed for the lower

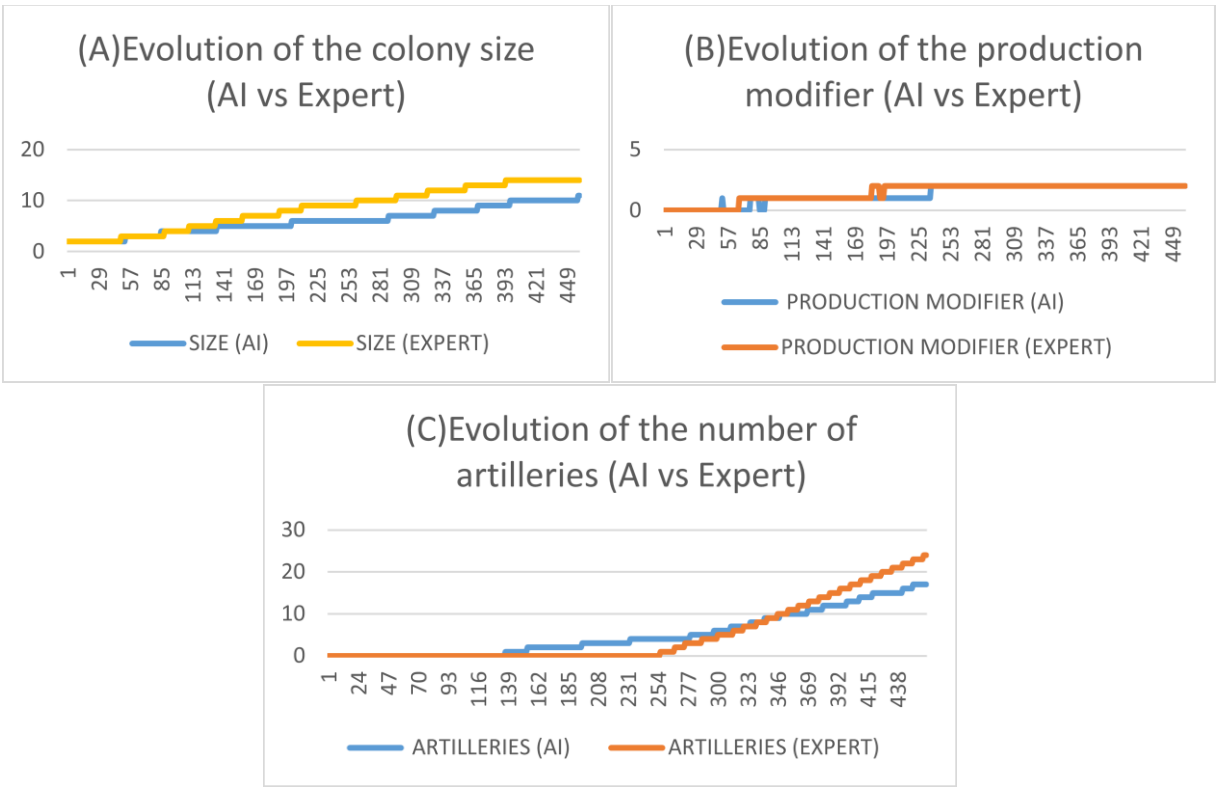

Figure 32 Comparison of the evolution of (A) colony size, (B) production modifier, (C) number of artilleries produced between our AI player and the Expert

total number of military units. Thus, we explain this difference in results with the expert player by a shortage of able workers that could work in the lumber mill or in the blacksmith house to produce the hammers and tools required for the build items.

6.3.1 Unit allocations, thresholds, stimuli

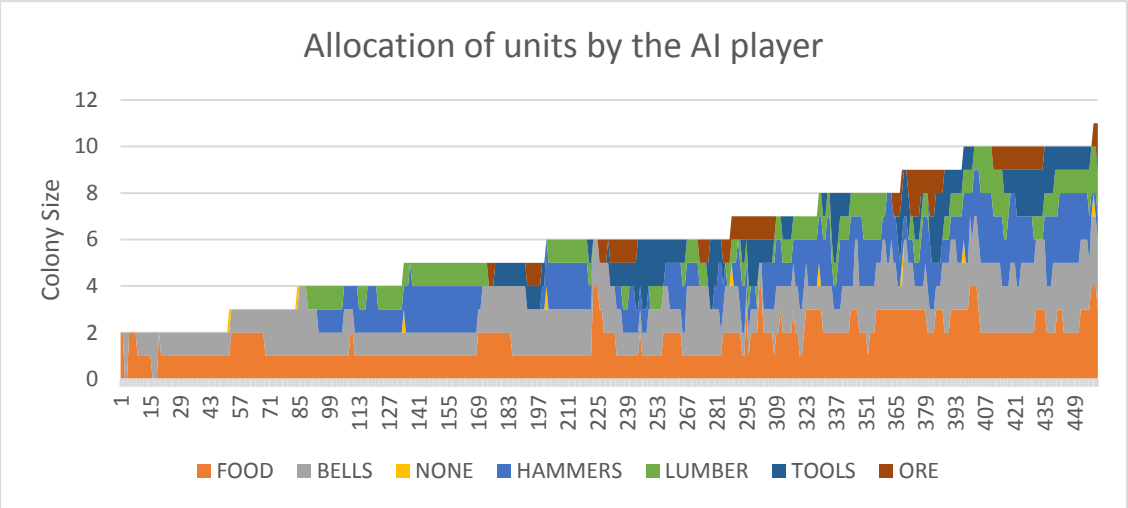

Figure 33 Allocation of units by the AI player, as a function of the colony size

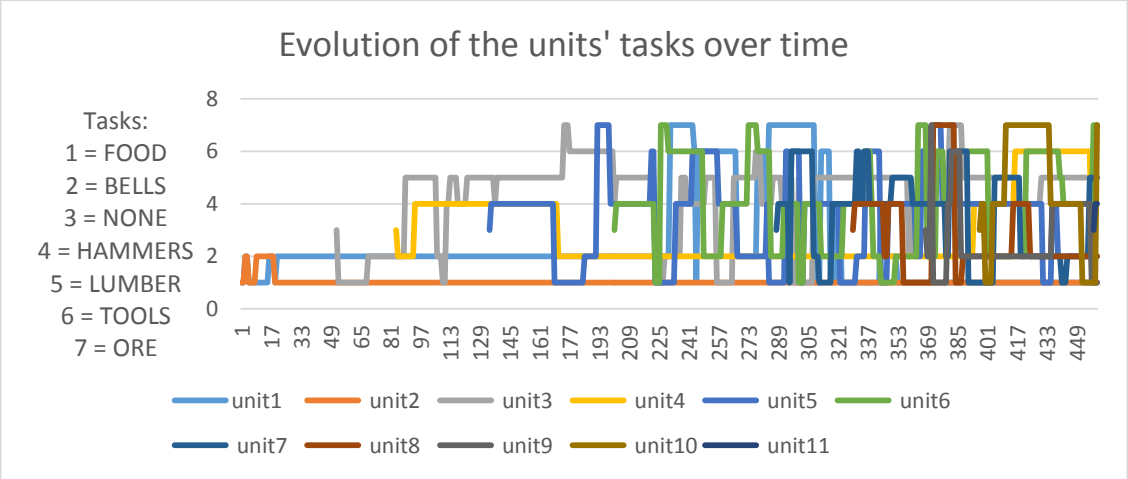

Figure 34 Evolution of a unit's job. The Y axis is the numeral representation of the jobs

Figure 33 and Figure 34 show how each unit is allocated by the AI. In Figure 33, we see that compared to the expert's allocation (Figure 31), our AI player gives less stability to the units. Figure 34 shows that the first units in the early game do not wildly swing from one job to another, while the units created later in the game do not have the time to settle down in a given job, possibly missing the chance to become an expert at their job.

Figure 35 shows the evolution of the raw stimuli (before applying the scale factor) during the game. We clearly see the spikes for the FOOD stimulus every time the colony has a food deficit, and we also see the slow increase that happens when the colony's food production is less than 6 above the food consumption. This is the expected behavior to prevent famine episode,

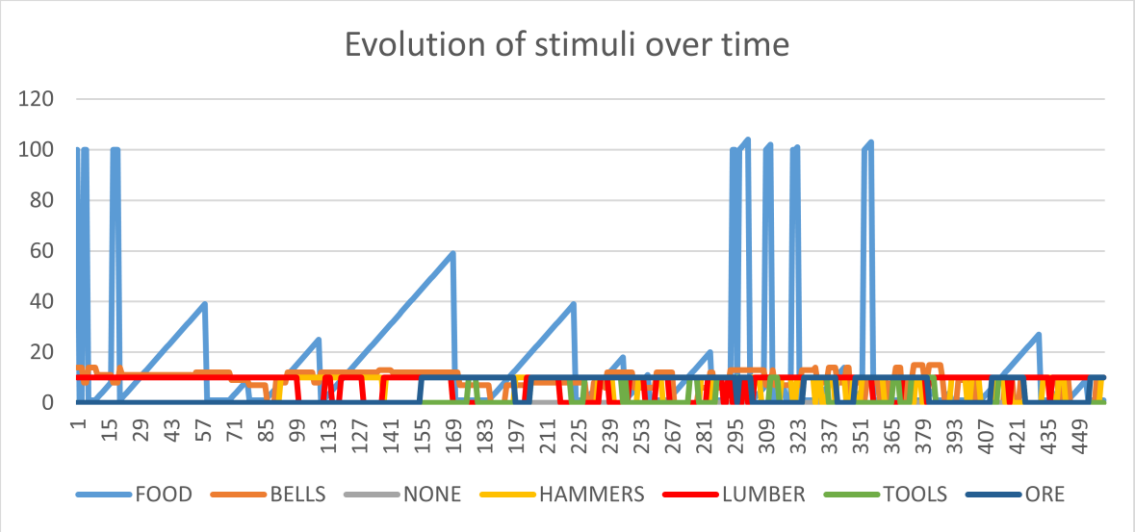

Figure 35 Evolution of raw stimuli during a game played by the AI player.

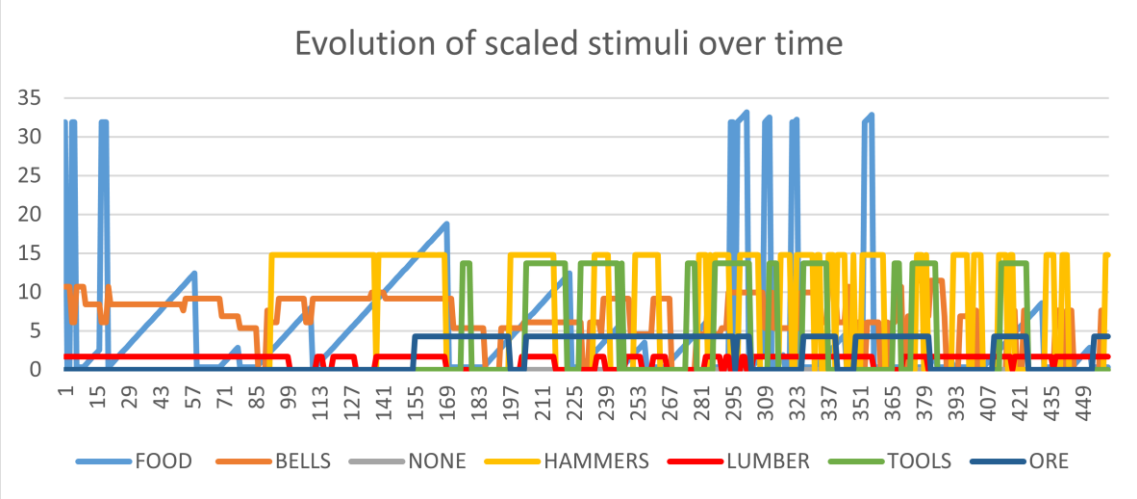

Figure 36 Evolution of stimuli multiplied by their scale factors over time

and Table 8 shows that this system works well since only 1 game out of 5 experiences a famine.

However, beside the FOOD stimulus, Figure 35 is hard to comprehend since most of the other stimuli are either 10 or 0, except the BELLS stimuli which can grow slowly over time when not enough bells are produced. Figure 36 shows the stimuli multiplied by their scale factors and is more revealing. Here we still see the spikes in the FOOD stimulus, but we also see that at the beginning of the game, the BELLS stimulus is quite high, while, around turn 85 the HAMMERS stimulus takes precedence, followed by the TOOLS stimulus about 100 turns later. Indeed we see in Figure 33 that the first unit is assigned to producing HAMMERS right after turn 85, when the HAMMERS stimulus first spike. If we compare it with our expert player unit assignments, it's about 75 turns earlier than what could be considered 'optimal' (assuming the expert player played an optimal game).

Finally, we look at the evolution of individual unit's thresholds and their task allocation, paying special attention to see which units became experts and if they were allocated to their area of

expertise. Only three units achieved expert status in this game. Unit1 and Unit2 became expert in FOOD production while unit3 became expert in LUMBER. This is what we guessed when we saw the oscillations between jobs for the units created later in the game. Only the first few units have a stable enough environment to get a chance of becoming experts. Figure 37 shows the evolution of thresholds for each expert unit and the units behavior over time. Unit 1 first becomes 'specialized' in BELLS and stays in that job for about half the game. It then

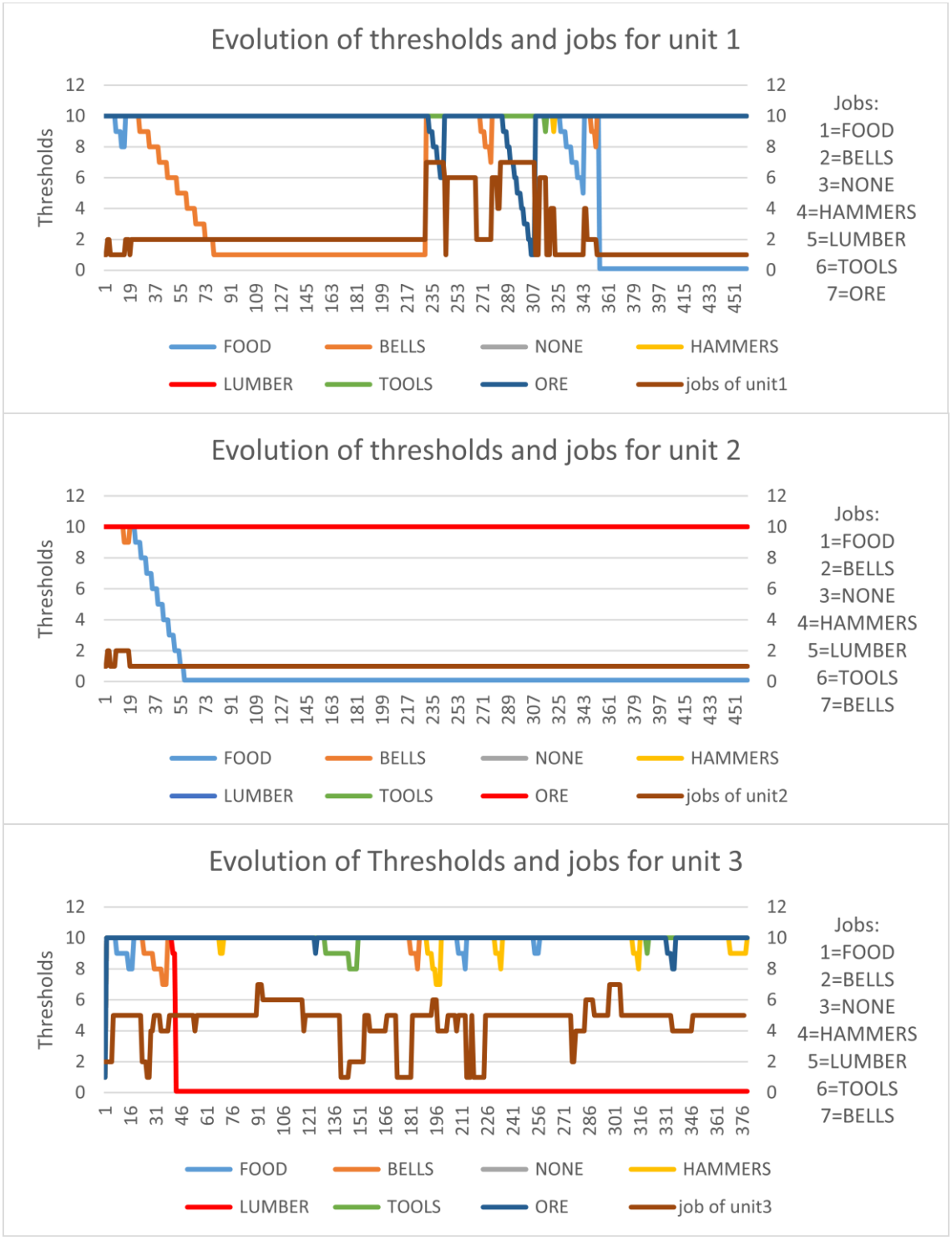

Figure 37 Thresholds and jobs for the three expert units. The brown line shows the jobs assigned to the unit over time

experiences a period of instability where it switches chaotically between jobs, to eventually settle down as a farmer and become an expert at it. The instability period is the result of having too many stimulus active and not enough units.
On the other hand, unit 2 has a much smoother experience, becoming an expert farmer early in the game and staying in that position until the end. Finally, unit 3 becomes specialized in LUMBER around turn 40 and spends most of its time doing just that, however, the LUMBER stimulus being very small after scaling (Figure 36), it gets kicked to another job from time to time. Still, the fact that it always comes back to its specialty shows the robustness of the system to properly allocate expert workers.

On the other hand, units that work in buildings cannot achieve the 0.1 'expert' threshold, because they cannot become expert in those jobs. Recall that we still decided to decrease their threshold to give them more stability, but we set a lower limit of 1 instead of 0.1. Figure 38 shows the evolution of thresholds for one such non-expert unit. Here we see two problems

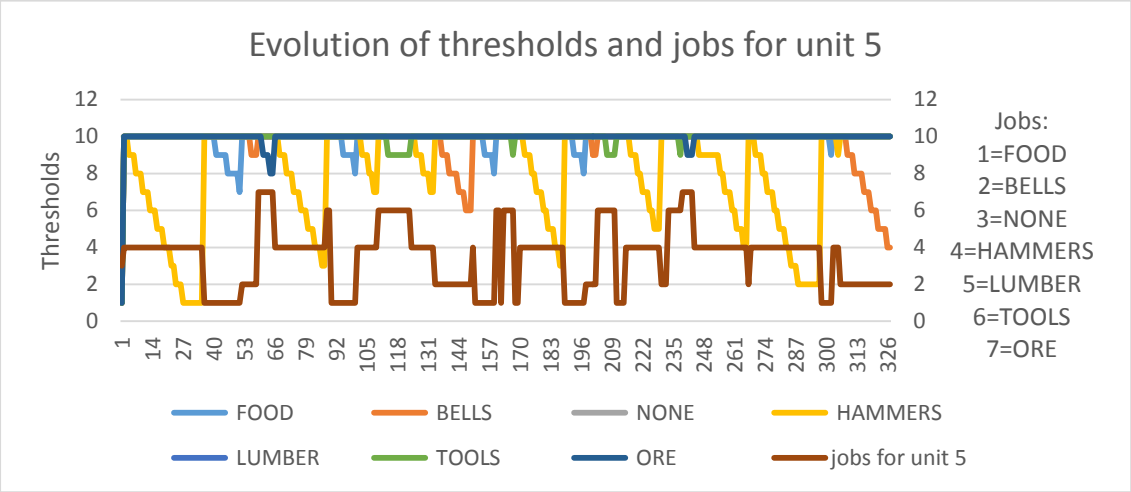

Figure 38 Evolution of the thresholds and jobs for a non-expert unit

with our approach: First, the thresholds do not decrease fast enough to provide stability in the later part of the game as several times the hammer threshold starts to decrease but never fast enough to achieve specialization. This makes it hard for units to settle in one job. Second, the fact that we reset the threshold to the maximum value once the unit changes job means that preference for a job where units cannot become expert is lost as soon as the stimulus disappears. This decision was motivated by the game mechanics, which allow a unit to reset its experience when it wants to switch jobs. We see now that the AI player would benefit from a periodic increase of thresholds unrelated to the job currently being done, rather than a total reset, so units could recover more easily from responding to brief spikes in other stimuli.

6.3.2 Summary of results for the single layer response threshold AI player

First, let's go back to the goals we had fixed for the AI:

1. Victory condition: At least 50% *SoL*. The AI met this goal at every game so the allocation system is very efficient in that area

2. Victory condition: Highest defense possible. The AI builds both defensive buildings and attempts to create as many military units as it can, although it falls short of the Expert player in this area. Still, the AI is able to build a reasonable defense in most cases. There is room for improvements, especially in improving the reliability of the player. And some of the games peaked as high as 87% of what a human can do, so finding a way to get that kind of performance consistently would be great.
3. Correct allocation of workers. We see some good and bad here. Providing enough food to avoid famine works pretty well but the colony could gain from growing faster and adding more units to the work force. Expert units are correctly allocated, while non-experts drift from one job to another because their threshold does not go down fast enough and resets too sharply to provide appropriate specialization.
4. Finding appropriate build order. The evolutionary algorithm can correctly identify pre-requisites in build order and the AI manages to build all important buildings, so this could be considered a success.

6.4 AI player: multi-layer response threshold

We now examine the effect of adding algorithm 2 (5.3.1) to our allocation algorithm. The aim is to restrict the allocation of workers to FOOD, BELLS, or NONE in the early stages of the game, where growing population and developing the production modifier should be prioritized. Indeed if we compare Figure 31 and Figure 33, one of the differences is that the expert player focuses on FOOD and BELLS until his colony reaches a size of 6, before diversifying into other resources. The AI player on the other hand starts allocating workers to LUMBER and HAMMERS starting from size 4, which slows the early growth of the colony.

Using values from Table 5 and Table 6, we run 100 instances of the game with the AI player augmented by the planning stimulus described in algorithm 2 (see 5.3.1). We compile the same statistics as the single layer response threshold AI player (Table 8). As we see in table 9, the average results for the score, military, and *SoL* percentage are significantly higher, according to a two tails paired student-T test.

Table 9 Statistics for different game indicators for the multi-layer AI player. 95% interval of confidence displayed with the mean. p-value for 2-tails paired student-T test comparison with results from Table 8

	Score	Size	Military	*SoL* %	Famine
MEAN	34.98 ± 1.41	14.53 ± 0.84	17.49 ± 0.70	99.17 ± 0.65	0.19 ± 0.10
MEDIAN	36	14	18	100	0
LOW	2	8	1	79	0
HIGH	48	27	24	100	3
p-VALUE	**0.0027**	0.2896	**0.0028**	**<0.0001**	0.8916

We see however that adding another layer of stochasticity increases the volatility of the system, as the low values (resp. high) values are lower (resp. higher) than our first player. During one of the game, the AI player was able to build as many artilleries as the expert player, while during another, it failed miserably and only managed to build one. The same game experi-

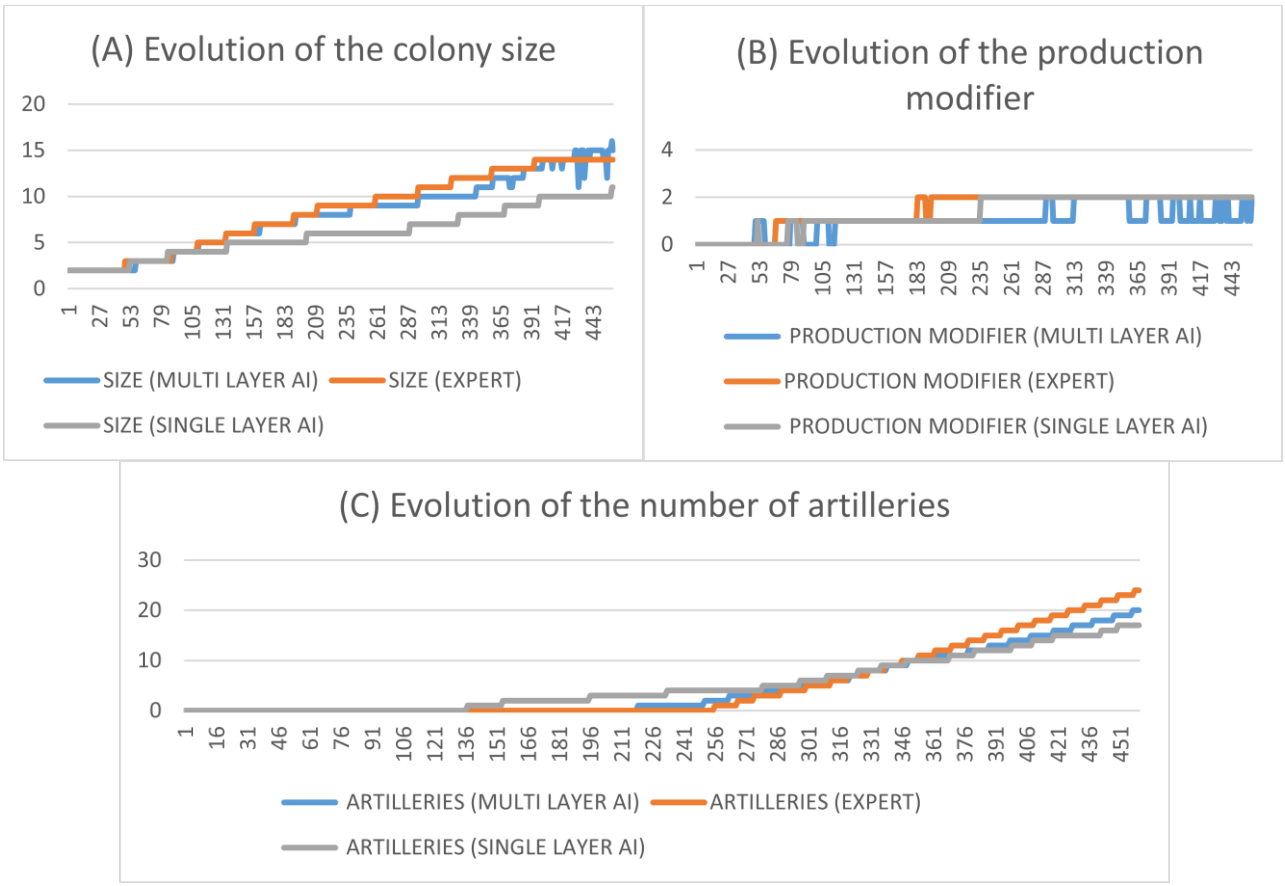

Figure 39 Comparison of the evolution of the colony size, the production modifier and the number of artilleries between the single response threshold AI player, the multi-layer AI player, and the expert player

enced 3 famine episodes, which killed the colony growth for most of the duration of the game.

6.4.1 Upper layer analysis

Figure 39 shows the comparison between all three players. The multi-layer AI player matches more closely the expert player on the colony size and the artillery production. However we see that the production modifier is lagging behind even the one for single layer AI player. This

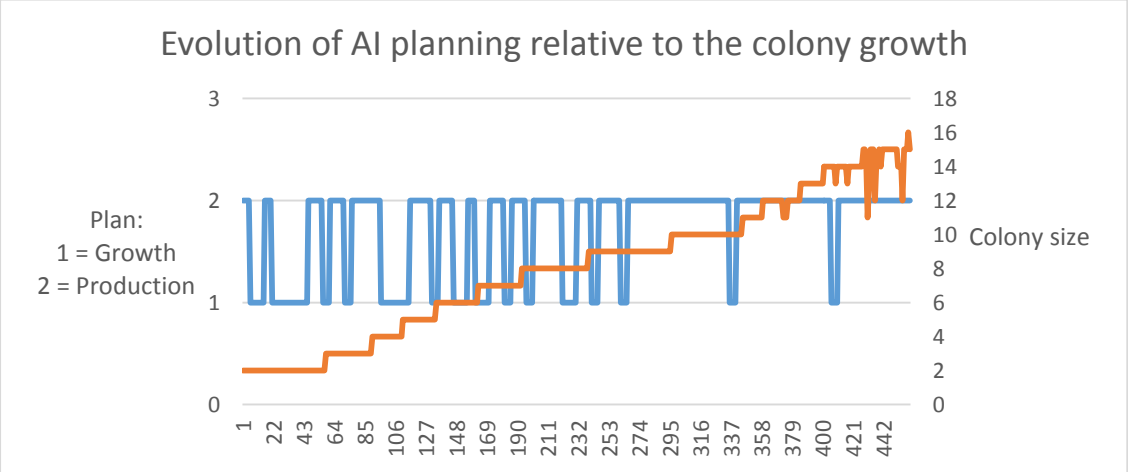

Figure 40 Evolution of planning (left vertical axis) relative to the colony size (right vertical axis)

causes the population to fluctuate at the end of the game because the AI cannot employ all available workers without causing a generalized drop in production. This sub-par production modifier is also what causes the production of artillery to be slower than the expert player, even though it still does better than the single layer AI.

Figure 40 shows the evolution of the colony planning according to the colony size. We see that it performs more or less how we expected, however we see that the production plan is still chosen quite often in the early game because the square factor used in the response threshold method (equation (45)) does not provide a sharp enough differentiation for the range of colony size (ideally from 2 to 6) that was chosen. We should thus seriously consider adding a parameter ϕ to increase specialization at low thresholds

$$\mathrm{P}_C = \frac{S_{C,GROWTH}^{\phi}}{S_{C,GROWTH}^{\phi} + \Theta_{C,GROWTH}^{\phi}} \tag{48}$$

6.4.2 Summary of results for the multi-layer response threshold AI player

The addition of another layer presents some advantages in term of average result and the improved player adopts a closer evolution to the expert player than our initial AI. However, this comes at a cost of more instability in the system, which makes sense since we added yet another stochastic process. In light of those results, we do not feel that this kind of system could be expanded upon broadly. Adding more options to the second layer, or adding more layer will most likely make the system more chaotic and at the end will produce totally random results. This is probably why there was never any follow-up to the hierarchical ant model outlined in 2.6.

6.5 AI player: rule based planning

Finally, we look at the performances of our AI player when we augment the stimuli computations in the primary response threshold layer with rule-based planning. Recall that we do that by limiting the generation of stimuli for the production related resources (HAMMERS, LUMBER, TOOLS and ORE) by adding the condition that the colony must have at least size 6 to have a non-zero stimulus for those four goods (equation (47), see 5.3.2).

Table 10 Statistics for different game indicators for the augmented rule-based planner AI player. 95% interval of confidence displayed with the mean. p-VALUES from a two tails paired student-T test compared with results in Table 8 (single layer) and Table 9 (multi-layer)

	Score	Size	Military	*SoL* %	Famine
MEAN	37.32 ± 1.30	17.49 ± 1.03	18.80 ± 0.54	98.22 ± 1.28	0.16 ± 0.09
MEDIAN	38	18	19	100	0
LOW	0	6	3	47	0
HIGH	48	36	24	100	3
p-VALUE (single layer)	**<0.0001**	**0.0004**	**<0.0001**	**<0.0001**	0.7744
p-VALUE (multi-layer)	**0.0059**	**<0.0001**	**0.0007**	0.2029	0.6418

As the bold values indicate in Table 10, augmenting the response threshold algorithm with rule-based planning performs significantly better than both previous AI players in term of score, colony size and military production, according to a two tails paired student-T test. With this player, we're getting close to 80 percent of the expert's production on average, with the best games equaling 100 percent. The only minor glitch is one game that did not achieve the 50 percent *SoL* required to declare independence, and thus ended up with a score of 0. However, since the average *SoL* percentage being 98.22%, this is a very rare occurrence.

Figure 41 shows a comparison between one of the best games played by the rule-based augmented AI and the expert game. There is a very close match in colony growth between both

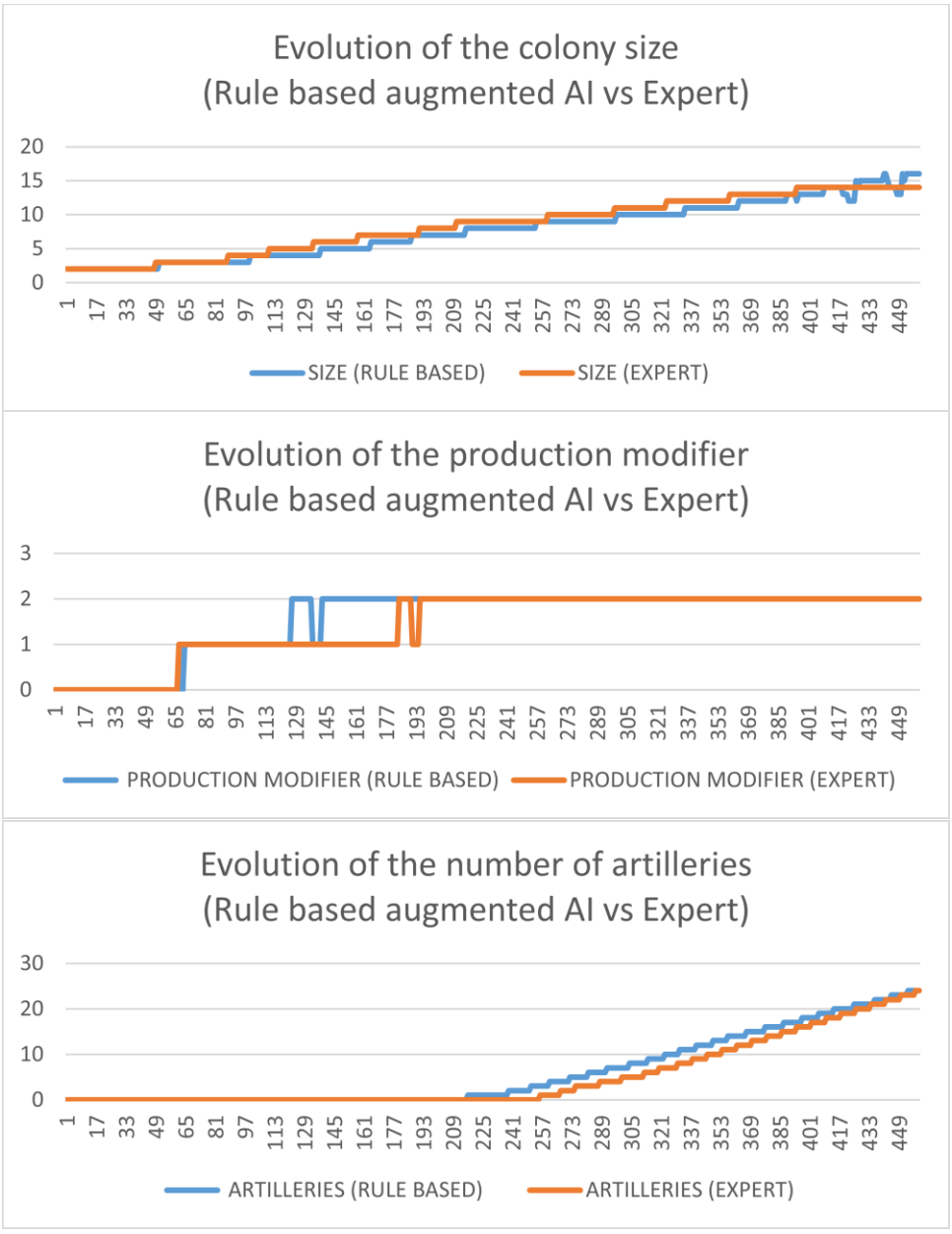

Figure 41 Comparison between the evolution of the colony size, production modifier and number of artilleries produced between our rule-based augmented AI and the expert

players and the production modifier for the AI player even maxes out faster. We see however that while the total number of artilleries produced is equal, the AI player takes more time to build them, which means the production is not fully optimized in the later game. Still, those results are very satisfying and this shows that such simple hybrid models might be a valid technique for developing at least part of an artificial player at a fraction of the efforts required to create a traditional scripted AI player .

6.6 A State of the art AI player for FreeCol

Following our observations of possible improvements in the threshold updates in FreeCol, and the overall quality of the rule-based augmented player, we propose one last implementation of the AI player which we'll call 'state of the art' for lack of better term.

For this player, we use the allocation algorithm (algorithm 1, see 5.1.4) with the rule-based augmented stimuli computations (described in 5.3.2) just like the player in the previous section. We also modify the threshold update rule ((41) see 5.1.3) to reflect our observations about Figure 38.

Let θ_{ug} be the threshold of unit *u* for stimulus S_g. For each turn that unit *u* performs job *g*

$$\theta_{ug} = \begin{cases} 0.1 & if(u\ is\ expert\ for\ g) \\ \max(1, \theta_{ug} - 1) & if\ not \end{cases} \quad (49)$$

Similarly, for each turn that unit *u* does not perform job $k \neq g$

$$\theta_{ug} = \begin{cases} 0.1 & if(u\ is\ expert\ for\ k) \\ \min(10, \theta_{ug} + 1) & if\ not \end{cases} \quad (50)$$

Finally, for each turn that unit *u* is idle, we lower every thresholds to increase their responsiveness to stimuli. For all jobs $g \in G$

$$\theta_{ug} = \begin{cases} 0.1 & if(u\ is\ expert\ for\ g) \\ \max(1, \theta_{ug} - 1) & if\ not \end{cases} \quad (51)$$

With this threshold update rule in place, units should demonstrate job specialization more rapidly and hopefully this will help them become experts. On the other hand, resetting thresholds more slowly when they switch jobs gives them a chance to recover from a brief but unfortunate spike in stimulus.

Before testing this new player, we perform another round of parameter optimization for the scale factors with our evolutionary algorithm. We reuse the same values for chromosome 2 and find the following values for chromosome 1 to be optimal:

Table 11 Optimized values for stimuli scale factors for the state-of-the-art player

Chromosome 1

θ_{FOOD}	0.445	θ_{LUMBER}	0.468	θ_{ORE}	0.580	θ_{NONE}	1.009
θ_{BELLS}	0.487	$\theta_{HAMMERS}$	1.660	θ_{TOOLS}	1.880		

We once again play 100 games with our improved player and we compare the results with those in 6.5 to find out the effect of our updated threshold rule.

Table 12 Statistics for different game indicators for the state-of-the-art AI player. 95% interval of confidence displayed with the mean. p-VALUES from a two tails paired student-T test compared with results in Table 11(rule-based augmented AI player)

	Score	Size	Military	*SoL* %	Famine
MEAN	41.98 ± 1.22	20.21 ± 0.94	20.99 ± 0.55	100 ± 0.92	0.04
MEDIAN	42	20	21	100	0
LOW	26	12	13	100	0
HIGH	52	31	26	100	1
p-VALUE (rule-based)	**<0.0001**	**<0.0001**	**<0.0001**	**0.0067**	0.0226

We see in Table 12 that our intuition about the behavior of the thresholds was the correct one, as this new improved player shows drastic improvements on every levels. It is both more reliable as none of the low values are critically bad and it has higher averages on all but the last indicator which was already good for the other players. It even manages to outperform the number of artilleries built by the expert player, showing true optimization capabilities. While such a level of play does not happen every game, the state-of-the-art player manages to equal the expert's performance 13% of the time.

Figure 42 and Figure 43 indeed show that the workers demonstrate more stability in their work, and the colony seems to reach equilibrium in the later part of the game. This allows us to assume the system would stay stable even during a longer game, and the newly created units would not affect the allocation performances.

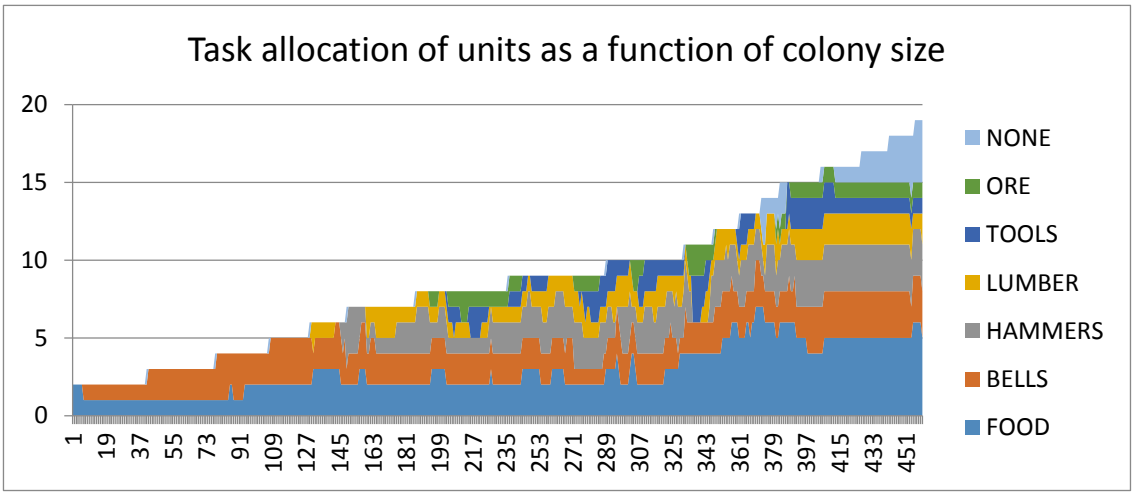

Figure 42 Allocation of units to the different tasks as a function of the colony size. One can notice only very limited variations after turn 400

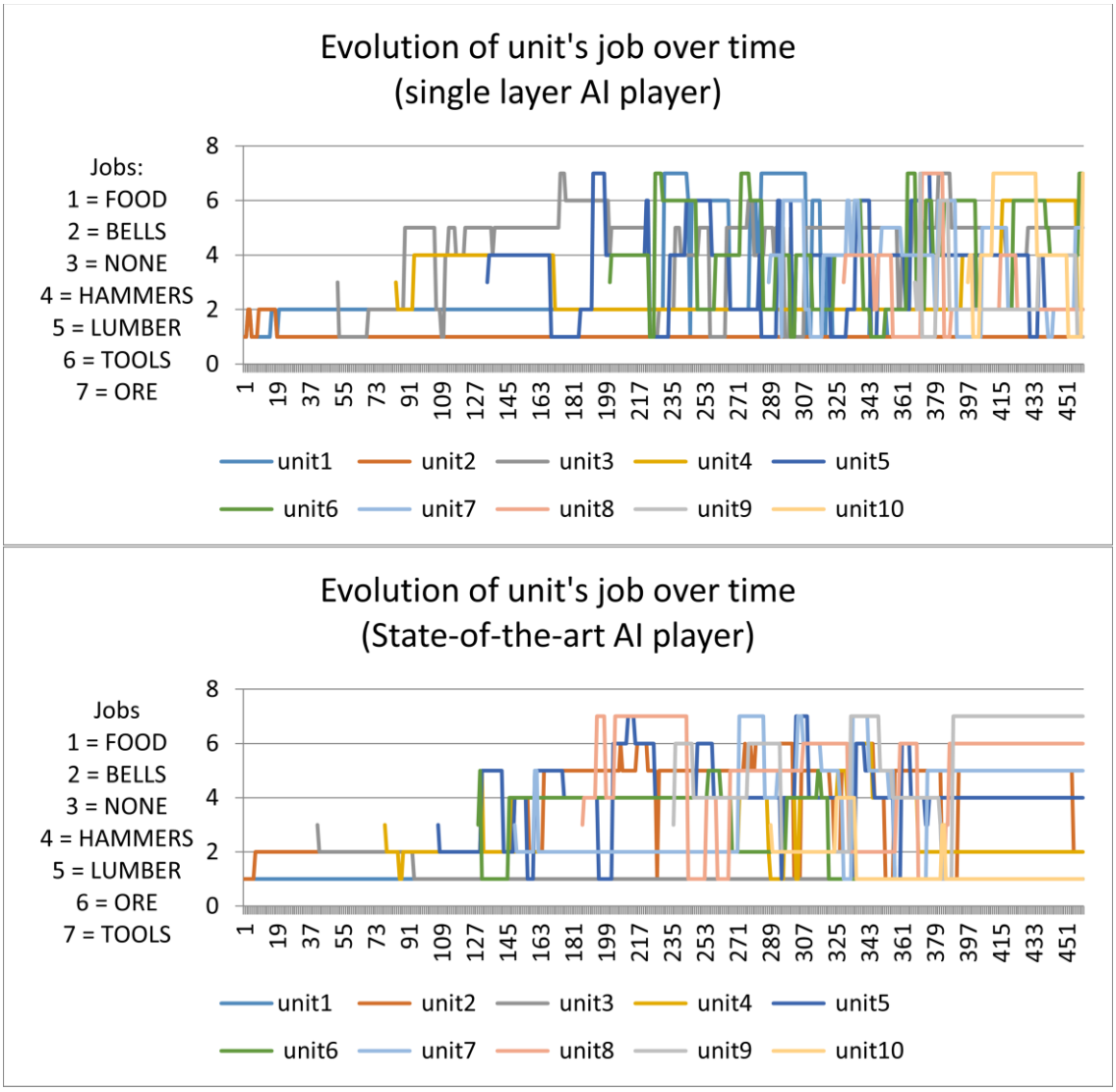

Figure 43 Comparison of the stability of worker's tasks over time, between the original single response threshold player (top) and the state-of-the-art AI player (bottom)

We end our experiments with this ultimate AI player. Looking back over the goals that we fixed for our AI player in 4.2, we feel confident that this final version meets all of them successfully. Indeed if we look at the computer game industry, most AI players performing at human level are scripts that range in the thousands line of codes to handle every possible situations, when it does not simply cheat to play at that level. Here, our player fits in a couple hundred lines, is easily customizable, and could even adapt to the experience of the human opponent by choosing sub-optimal scale factors if the adversary does not play at expert level.

Chapter 7. Conclusion

Intelligence comes in many forms, and while humans are undoubtedly the pinnacle of intelligent beings, insect-inspired swarm intelligence has been a very successful subfield of AI. Ant colony optimization for example has produced state of the art algorithm in many combinatorial optimization problems. With this work we add another field, computer games AI, where swarm intelligence could shine.

Regrettably, computer games so far have not received the attention they deserve by the academic field. Yet, the challenges found in games are perfect to test the application of advanced academic AI research. Computer games provide a ready-made digitalized framework and propose challenges that would require developing an intelligence closer to the capacities of humans. This was the elusive initial goal of AI pioneers like Turing, and which often times tends to be side-stepped in favor of solving yet another NP-hard industrial problem. But now that the computer games field has matured beyond geekdom to become mainstream entertainment, and that the game industry has grown to represent considerable financial resources; implementing smart, efficient, intelligent algorithms in games is an open door to a profitable and exciting career.

7.1 Contributions

Our contributions are threefold. First, we showed that using the threshold response method, with very simple rules, an efficient resource management system can emerge, even with intricate constraints such as the ones present in a strategy game. Without any fine tuning, our basic AI player was able to perform at around 66% of an expert player on average, over a full game episode.

Second, we showed that while adding another response threshold layer to emulate planning capabilities brings better average results, the variation in the player's performances from one game to another increases beyond that of our initial player. Thus it will make such an approach inappropriate for most processes with low risk tolerance, and it shows this kind of architecture might be a dead end.

Finally, we showed that with a few minor modifications, such as adding rule-based constraints to our stochastic algorithm, we obtained an AI player that can play the resource management part of a FreeCol game at near expert level, with excellent results on average and good results in the worst case.

Although applications of the response threshold algorithm have been developed academically before, it is, to our knowledge, only the second time it has been applied to a real problem, and the first time where multiple agents have to choose between multiple tasks at each time step. Beyond the obvious use in computer games where this problem is very common, we believe it could be applied to a wide range of task allocation problems where worker allocation has to be done in an efficient manner (automatic scheduling for managers, weapons-target allocation, etc…). But even just in the computer game field, it could potentially be a big improvement to traditionally cheating or rigidly scripted artificial player, which are very hard to develop in an efficient manner.

7.2 Future work

While it would probably be hard to improve the quantitative results of this method for this specific problem, improvements to the average results could come from examining in details why the player does not perform that well in some game, and see if we can reduce or eliminate such occurances. Furthermore, we only tackled a sub-part of the game, so it would be very interesting to see if the algorithm can be scaled up to coordinate multiple colonies, handle trade and the other parts of the game that we didn't touch, or if like our attempt to create a multi-level system, too much stochasticity ends up in chaos.

A References

Amato, C., & Shani, G. (2010). High-level Reinforcement Learning in Strategy Games. *9th International COnference on Autonomous Agents and Multiagent Systems: Volume 1* (pp. 75–82).

Bellman, R. (1957). *Dynamic Programming* (p. 366). Princeton University Press. Retrieved from http://books.google.com/books?id=fyVtp3EMxasC&pgis=1

Bengio, Y. (2010). Introduction to Deep Learning Algorithms — Notes de cours IFT6266 Hiver 2010. Retrieved August 14, 2012, from http://www.iro.umontreal.ca/~pift6266/H10/notes/deepintro.html

Bengio, Y., Lamblin, P., Popovici, D., & Larochelle, H. (2007). Greedy Layer-Wise Training of Deep Networks. *NIPS 2006* (pp. 153–160).

Bergsma, M., & Spronck, P. (2008). Adaptive Spatial Reasoning for Turn-based Strategy Games. *4th Artificial Intelligence and Interactive Digital Entertainement Conference* (pp. 161–166).

Blum, C. (2004). *Theoretical and Practical Aspects of Ant Colony Optimization* (p. 294). Amsterdam: IOS Press.

Bonabeau, E., Sobkowski, A., Theraulaz, G., & Deneubourg, J. (1998). Adaptive Task Allocation Inspired by a Model of Division of Labor in Social Insects. *Biocomputing and Emergent Computation*, (8), 36–45.

Bonabeau, E., Theraulaz, G., & Deneubourg, J.-L. (1996). Quantitative study of the fixed threshold model for the regulation of division of labour in insect societies. *Proceedings: Biological Sciences*, *263*(1376), 1565–1569.

Campos, M., Bonabeau, E., Théraulaz, G., & Deneubourg, J.-L. (2000). Dynamic Scheduling and Division of Labor in Social Insects. *Adaptive Behavior*, *8*(2), 83–96.

Champandard, A. J. (2011). This Year in Game AI: Analysis, Trends from 2010 and Predictions for 2011 | AiGameDev.com. Retrieved July 31, 2012, from http://aigamedev.com/open/editorial/2010-retrospective/

Cicirello, V. A., & Smith, S. F. (2001). Wasp-like Agents for Distributed Factory Coordination. *Autonomous Agents and Multi-Agent Systems*, *8*(3), 237–266.

Hinrichs, T. R., & Forbus, K. D. (2007). Analogical Learning in a Turn-Based Strategy Game. *Twentieth International Joint Conference on Artificial Intelligence* (pp. 853–858).

Hinton, G. E., Osindero, S., & Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, *18*(7), 1527 – 1554.

IBM. (2012, April 11). IBM Watson. Retrieved from http://www-03.ibm.com/innovation/us/watson/

Laird, J. E., & van Lent, M. (2001). Human-Level AI's Killer Application, Interactive Computer Games. *AI Magazine*, *22*(2), 15–26.

Liu, N., Abdelrahman, M., & Ramaswamy, S. (2005). A genetic algorithm for single machine total weighted tardiness scheduling problem. *INTERNATIONAL JOURNAL OF …*, 1–8. Retrieved from http://www.lifl.fr/~derbel/ueOC/5/genetic.pdf

Morley, C. (1996). *Painting trucks at General Motors: The effectiveness of a complexity-based approach. In: Embracing Complexity: Exploring the Application of Complex Adaptive Systems to Business* (pp. 53–58). Ernst & Young. Retrieved from http://130.203.133.150/showciting;jsessionid=3F4B0AA98748CCA9F649643644D19FCD?cid=2171110

Nau, D., Cao, Y., Lotem, A., & Muftoz-avila, H. (1999). SHOP : Simple Hierarchical Ordered Planner. *Sixteenth International Joint Conference on Artificial Intelligence* (pp. 968–973).

Nitsche, M., Fitzpatrick, R., Ashmore, C., Kelly, J., & Margenau, K. (2006). Designing Procedural Game Spaces : A Case Study. *FuturePlay*, *14*(10), 187–193.

Nouyan, S., Ghizzioli, R., Birattari, M., & Dorigo, M. (2005). An insect-based algorithm for the dynamic task allocation problem. *Künstliche Intelligenz*, *19*(1), 25–31. Retrieved from http://www.metaheuristics.org/~mbiro/paperi/IridiaTr2005-031r001.pdf

Poe, E. A. (1836). Maelzel's Chess-Player. *Southern Literary Messenger*, 318–326.

Ranzato, M. A., Poultney, C., Chopra, S., & Lecun, Y. (2007). Efficient Learning of Sparse Representations with an Energy-Based Model. *NIPS 2006* (pp. 1137–1144).

Robinson, G. E., Page, R. E., & Huang, Z.-Y. (1994). Temporal polyethism in social insects is a developmental process. *Animal Behaviors*, *48*, 467–469.

Simon, H. A. (1962). The Architecture of Complexity. *Proceedings of the American Philosophical Society*, *106*(6), 467–482.

Smith, A. (1776). *The wealth of nations* (p. 900). London: Methuen & Co. Ltd.

Sánchez-ruiz, A., Lee-urban, S., Muñoz-avila, H., Díaz-agudo, B., & González-calero, P. (2007). Game AI for a Turn-based Strategy Game with Plan Adaptation and Ontology-based retrieval. *ICAPS 2007 Workshop on Planning in Games*.

Takahashi, D. (2000). Artificial Intelligence Gurus Win Tech-Game Jobs. *The Wall Street Journal*, p. 14.

Takahashi, Dean. (2011). With online sales growing, video game market to hit $81B by 2016 (exclusive) | VentureBeat. Retrieved August 14, 2012, from

http://venturebeat.com/2011/09/07/with-online-sales-growing-video-game-market-to-hit-81b-by-2016-exclusive/

The Top 25 PC Games of All Time - PC Feature at IGN. (n.d.). Retrieved August 15, 2012, from http://pc.ign.com/articles/082/082486p1.html

Theraulaz, G, Bonabeau, E., & Deneubourg, J.-L. (1998). Response threshold reinforcements and division of labour in insect societies. *the Royal Society*, 1–18. Retrieved from http://rspb.royalsocietypublishing.org/content/265/1393/327.short

Theraulaz, Guy, Bonabeau, E., & Deneubourg, J.-L. (1995). Self-organization of Hierarchies in Animal Societies : The Case of the Primitively Eusocial Wasp Polistes dominulus Christ. *Journal of Theoretical Biology*, *174*, 313–323.

UdeM. (2011). Montreal's new research chair for artificial intelligence. *UdeM Nouvelles*. Retrieved from http://www.nouvelles.umontreal.ca/udem-news/news/20110315-a-research-chair-in-artificial-intelligence-in-montreal.html

Warfield, I., Hogg, C., Lee-Urban, S., & Munoz-Avila, H. (2007). Adaptation of hierarchical task network plans. *Twentieth International FLAIRS conference*.

Wender, S., & Watson, I. (2008). Using reinforcement learning for city site selection in the turn-based strategy game Civilization IV. *2008 IEEE Symposium On Computational Intelligence and Games*, 372–377. doi:10.1109/CIG.2008.5035664

Wilson, E. O. (1984). The relation between caste ratios and division of labor in the ant genus Pheidole (Hymenoptera: Formicidae). *Behavioral Ecology and Sociobiology*, *16*(1), 89–98. Retrieved from http://www.springerlink.com/content/u5067p1121415116/

Withers, G. S., Fahrbach, S. E., & Robinson, G. E. (1993). Selective neuroanatomical plasticity and division of labour in the honeybee. *Nature*, *364*, 238–240.

Wolf, T. D., Jaco, L., Holvoet, T., & Steegman, E. (2002). A nested layered threshold model for dynamic task allocation. *Third International Workshop on Ant Algorithms* (pp. 290–291). Retrieved from http://www.springerlink.com/index/7VB3JL2F3DDHPMBV.pdf