Adaptive Intelligence for Turn-based Strategy Games

Maurice Bergsma Pieter Spronck

Tilburg University / Tilburg centre for Creative Computing
P.O. Box 90153, NL-5000 LE Tilburg, The Netherlands

Abstract

Computer games are an increasingly popular form of entertainment. Typically, the quality of AI opponents in computer games leaves a lot to be desired, which poses many attractive challenges for AI researchers. In this respect, *Turn-based Strategy* (TBS) games are of particular interest. These games are focussed on high-level decision making, rather than low-level behavioural actions. Moreover, they allow the players sufficient time to consider their moves. For efficiently designing a TBS AI, in this paper we propose a game AI architecture named ADAPTA (Allocation and Decomposition Architecture for Performing Tactical AI). It is based on task decomposition using asset allocation, and promotes the use of machine learning techniques. In our research we concentrated on one of the subtasks for the ADAPTA architecture, namely the Extermination module, which is responsible for combat behaviour. Our experiments show that ADAPTA can successfully learn to outperform static opponents. It is also capable of generating AIs which defeat a variety of static tactics simultaneously.

1 Introduction

The present research is concerned with artificial intelligence (AI) for turn-based strategy (TBS) games. The AI for TBS games offers many challenges, such as resource management, planning, and decision making under uncertainty, which a computer player must be able to master in order to provide competitive play. Our goal is to create an effective turn-based strategy computer player. To accomplish this, we employ learning techniques to generate the computer player's behaviour automatically. For this research we are most interested in the '4X' subgenre of TBS games. 4X stands for the primary goals of this type of game, namely Exploration, Expansion, Exploitation, and Extermination.

As TBS games are in many ways related to real-time strategy (RTS) games, the research challenges of these domains are rather similar. Buro and Furtak [2] define seven research challenges for RTS games, six of which are also relevant to the TBS domain. These are (1) Adversarial planning, (2) Decision making under uncertainty, (3) Spatial reasoning, (4) Resource management, (5) Collaboration, and (6) Adaptivity. While these six research challenges are similar for both genres, TBS offers more depth in some of them, such as collaboration and resource management. Comparing both genres, one can observe that, usually, RTS games rely more on fast, exciting action, while TBS games rely more on strategic thinking and careful deliberation. As far as research into these domains is concerned, this means that RTS game research is more dependent on the development of time-efficient algorithms, while TBS games require more intelligent AI techniques.

The outline of this paper is as follows. In Section 2 we provide a definition of the TBS game used for this research. Section 3 explains the AI architecture designed for this game. In Section 4 we describe how spatial reasoning is used in our AI, followed by an overview of the learning algorithm used to generate the behaviour of our AI in Section 5. The experimental setup is explained, and the results are discussed in Sections 6 and 7. Section 8 provides conclusions.

2 Game Definition

For this research, we have developed our own game definition, based on Nintendo's ADVANCE WARS, which is a relatively simple TBS game that still supports most of the major features of the genre. Each of the 4 X's (Exploration, Expansion, Exploitation, and Extermination) are represented in some form in the environment. We named our version SIMPLE WARS. Below, the rules of SIMPLE WARS are explained.

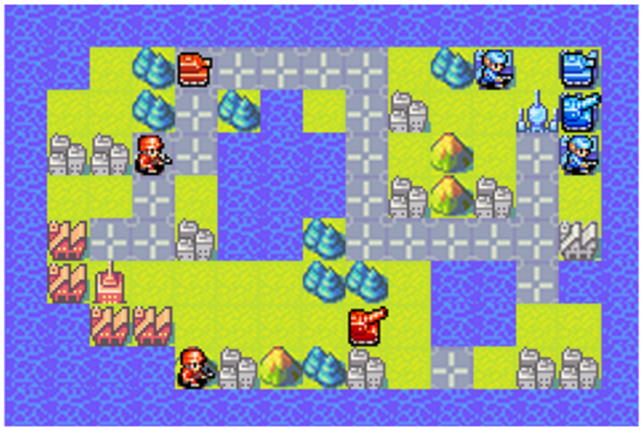
Figure 1: A screenshot of the TBS game under research.

The game takes place on a two-dimensional, tile-based map, as shown in Figure 1. A tile is of a predefined type, such as *Road*, *Mountain*, or *River*. Each type has its own set of parameters. These parameters define the characteristics of the tile, such as the number of moves that it takes for a unit to move over it, or the defense bonus that a unit receives while standing on it. By default, movement in the game can occur either horizontally or vertically, but not diagonally. A defense bonus lowers the amount of damage that the unit receives in combat.

Some tiles contain a base, for example a *City*, which provides a certain amount of resources to the player that controls the base, or a *Factory*, which can produce one unit each turn. Each unit requires a specific amount of resources to be built. A newly created unit is placed on the same tile as the *Factory* that produced it. This unit is unable to move in the turn it was created.

As is the case with bases and tiles, units come in several types. Each type has different values for its parameters, which include (1) the starting amount of health points for the unit, (2) the number of moves it can make per turn, (3) the amount of damage it can do to each type of unit, and (4) the actions the unit can perform. Because a unit in the game actually represents a squad of units, and damaging the unit represents destroying a part of the squad, the effectiveness of actions such as *Attack* (attacking another unit) is tied to the number of health points that remains for the unit. Additionally, the effectiveness of an *Attack* depends on the types of units involved.

The three types of units in this game are *Infantry*, *Tank*, and *Anti-Tank* units. Of these, only the *Infantry* unit is capable of moving over *Mountain* tiles. None of these units is able to move over *Sea* tiles. The units follow a rock-paper-scissors approach, which means that each of the three types is, where doing damage to the other two unit types is concerned, stronger than one and weaker than the other (Infantry defeats Anti-Tank, Anti-Tank defeats Tank, and Tank defeats Infantry).

At every turn, a player can build a single unit at each *Factory* under control, and perform (1) a *Move* and (2) an *Attack* or other action for each unit under control. An *Attack* action can be performed without moving, but a unit cannot move after performing this action. A tile can only contain a single unit at any time. This implies that a *Factory* is unable to produce a new unit whenever another unit is located at the same tile as this *Factory*. Additionally, a moving unit is able to pass through a tile occupied by a friendly unit, but not through a tile occupied by an enemy unit.

3 The ADAPTA Architecture

In many commercial strategy games, the AI is implemented using scripts, which quickly become difficult to balance or adapt. In order to keep the size of the AI manageable, the complex task in the script can be decomposed into several subtasks, which operate independently of each other, and concentrate each on performing a specific part of the complex task, without having to take any of the other subtasks into account. Examples of possible subtasks include resource gathering and combat.

While the goals of the subtasks are independent, they all share the same environment, namely the game world. Moreover, they need to share the finite number of *assets* (resources and game objects) that are available. By nature, a subtask only sees a part of the big picture, and is not concerned with the overall path

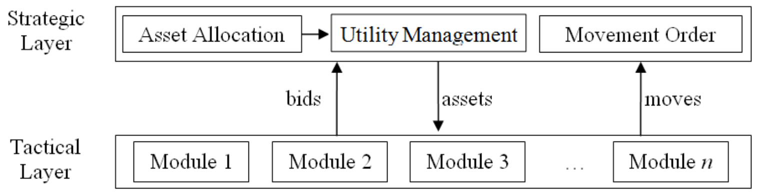

Figure 2: The ADAPTA Game AI architecture.

to victory. Therefore, separate modules are required to keep track of the AI's goals, to determine which goals (and therefore, which subtasks) have priority at a point in the game, and to allocate control over the available assets to the different subtasks. Because these modules keep track of the overall strategy, they are called the strategic modules. The subtasks are called tactical modules, as they are each responsible for one type of tactics. In combination, all these modules make up the ADAPTA architecture, depicted in Figure 2.

In the ADAPTA architecture, the strategic AI acts as an arbitrator between the different tactical modules. The Asset Allocation Module decides which tactical module gets control over which assets. This is achieved through auctioning. Tactical modules generate bids, which consist of one or more assets that a module wants to use, and a utility value that the module assigns to these assets (e.g., it assigns a high utility value to assets which it considers to be very useful for achieving its goals). The Asset Allocation Module uses these bids to find the allocation which maximises 'social welfare.' Bids are generated by the various tactical modules, which are not concerned with the bids of competing modules. Therefore, the strategic layer contains a Utility Management Module which weighs each bid's utility value according to the tactical module that generated it and the overall goal of the game. After the assets have been allocated, the Movement Order Module decides in which order the actions generated by the tactical modules are executed.

A tactical module is required to perform three tasks, namely (1) Bid Generation, (2) Utility Calculation, and (3) Action Generation.

1. A bid generator is responsible for submitting a set of bids to the Asset Allocation module. These bids should represent the optimal (or near optimal) actions for a subset of the available assets, according to a certain tactic. The ADAPTA architecture does not impose a specific method to generate the set of bids. Rather, the approach is entirely up to the designer of the module.

2. A utility function calculates a numerical utility value given a certain set of assets. This utility is used by the strategic layer to determine asset allocation. Utilities should provide an accurate measurement of the relative effectiveness of a certain tactic, compared to other tactics generated by the same module.

3. After all tactical modules have submitted their bids and the asset allocation is performed, assets are assigned to the tactical modules that submitted the winning bids. The modules generate game actions for the assets assigned to them. These actions are submitted (along with the utility of the associated bid) to the Movement Module in the strategic layer, which executes them in a certain order.

4 Spatial Reasoning

In this section we will take a closer look at the tactical layer of the ADAPTA architecture. As defined in the previous section, a tactical layer can contain any number of tactical modules, and a module can be responsible for any task. This chapter will focus on the creation of a single tactical module, which will serve as an example of how tactical modules fit into the ADAPTA architecture. This example module is named the Extermination module, as it is responsible for handling combat between units. In our environment, this amounts to assigning Move and Attack actions to each unit under the command of an AI.

In our research, we use influence maps [5, 6], to determine the optimal tile to perform an action. This means that the higher the influence is at a tile, the more preferable it is for a certain unit to move to this tile. For TBS games, an influence map is typically calculated just before making a decision, so that it will, due to the turn-based nature, always reflect the current game state.

The influence map is calculated separately for each tile, as follows:

$$I(x,y) = \sum_{o \in O} p(w(o), \delta(o,x,y)),$$

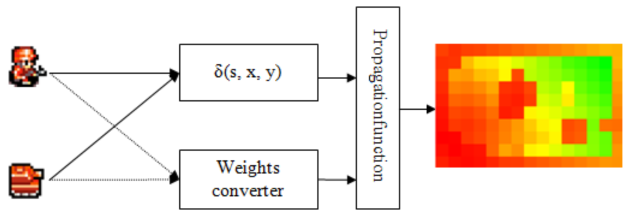

Figure 3: The calculation of an influence map.

where O is the set of all objects used for this influence map, $p(W, d)$ is a *propagation function* of weight vector W (values for attributes of the object which are used by the propagation function; often just one value for each object) and distance d, $w(o)$ converts object o into a weight vector, and $\delta(o, x, y)$ is a distance function calculating the distance from object o to tile (x, y). Simply put, the propagation function will ensure that the influence of an object is higher for tiles that are close to the object, and lower further away from the object. A graphical representation of the calculation of an influence map is shown in Figure 3. In this case, the influence map concerns moveable unit types, and may represent a concept such as 'offensive power.'

The behaviour of an influence map is defined by the two aforementioned functions, (1) the distance function δ and (2) the propagation function p. The distance function can be a general distance metric, such as Euclidean or Manhattan distance, or a domain-specific function, such as the number of steps that a unit needs to get to the target tile. The propagation function defines the influence that an object has on each tile. It is a function of the distance between the game object and current tile. Note that, for the propagation function, values below zero are set to zero, because the object has no influence in the corresponding tile rather than negative influence.

Because different types of game objects have different effects on the game, the calculation of influences should be performed differently for each type. Additionally, by summing both positive and negative influences for different types of objects in a single influence map, information may be lost. For these reasons, multiple influence maps should be maintained for different types of objects. These influence maps can then be analysed on their own as well as combined with others by a *layering algorithm*. A layering algorithm is a function that combines the contents of a number of influence maps and combines them into a single, new influence map. Figure 4 is an extension of Figure 3 which illustrates the process of layering two different influence maps, namely the one from Figure 3, and one concerning stationary units, which may represent a concept such as 'production capacity.'.

In our research, the layering algorithm is a neural network, which uses all influence maps that were the result of propagation functions as inputs. As output, it generates two influence maps simultaneously, one of which indicates the preferability of moving to each tile, and the other doing the same for attacking each tile. In this way, the task of the Extermination module is reduced to selecting the tiles with the highest influence from all possible targets for the Move and Attack actions.

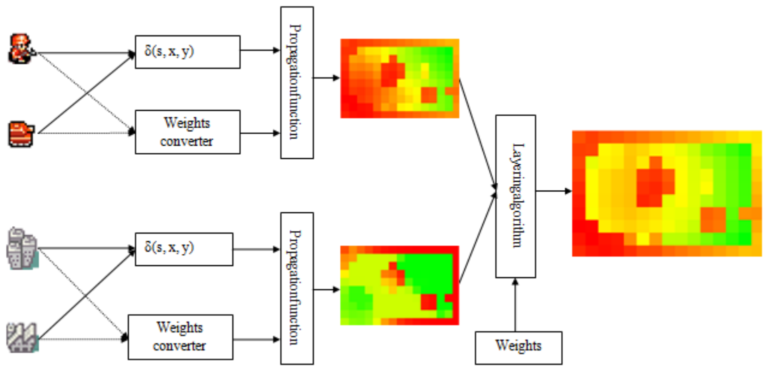

Figure 4: Layering two influence maps.

5 Adaptivity

The behaviour of an AI using the spatial reasoning approach described in the previous section depends on the choices for the propagation functions and layering algorithm, as well as their weights. To generate effective behaviour for such an AI, we use a learning algorithm. In order to keep the size of the search space manageable, we limit ourselves to generating the weights, while the choices of all functions remain fixed.

The learning algorithm we used is an evolutionary algorithm (EA) [3, 7]. In our approach, candidate solutions, or *individuals*, are represented by adding the weight strings of each neural network together, forming a single string of numbers. At each generation, the effectiveness (or *fitness*) of each of the individuals must be determined according to a certain *fitness measure*. Because the individuals define the behaviour of a game AI, their fitness can be determined by letting these AIs play the game that they are intended for. In order to attain consistent fitness values, all individuals are made to play against the same opponent(s). An individual's fitness is defined as

$$F_i = \frac{\sum_j R_{ij} + \sum_j R_{ji}}{2 \cdot |R_i|},$$

where R_{ij} is the numerical value resulting from a game between players i and j, where player i has starting position 1 and player j has starting position 2. The genetic operators chosen for the EA are capable of operating on neural networks. The operators are able to change single weights, entire nodes, or complete networks. A description of each of the operators, as well as a detailed overview of the EA used, is given by Bergsma [1].

6 Experiment 1: Iterative Learning

For our experiments, we implemented a research environment in which the game SIMPLE WARS can be played between different AIs.

It is impractical for the ADAPTA AI to learn against human players. We decided to determine the ability of the ADAPTA AI to defeat a certain tactic by letting it play against other AI implementations, such as scripts. We implemented a simple rush tactic to serve as an initial opponent. This tactic always makes its available units move towards the nearest enemy unit, and attacks whenever possible. In practice, for many commercial RTS games, rush tactics are considered to be a very strong, sometimes optimal approach to win the game [4].

Because the goal of this experiment is to train the Exterminate module, the game rules are focussed on combat. The goal of the game is to destroy 10 enemy units. This is also the total amount of units a player is allowed to build during the game. Because the game is played on a small map, both players are only allowed to control 4 units at any time. Both players start off without any units, but with 4 Factories, each of which is able to build all three unit types. Aside from the 10-unit limitation, both players are given sufficient resources to build 10 units at the start of the game.

Our goal is to improve iteratively the performance and behaviour of the ADAPTA AI, as well as assessing to which extent the results of the learning algorithm generalise over different opponents. Therefore, after the ADAPTA AI has learned to convincingly defeat the initial opponent, we started a new run with the learned AI now as the new opponent. Again, when this opponent was convincingly defeated, we replaced it with the newly generated AI. This process was repeated until ten new AIs were generated.

Table 1 lists the final minimum, average, maximum, and opponent fitness values for each iteration. The Rush AI is used as the opponent in the first iteration, the AI generated against the Rush AI is used as the opponent in the second iteration, etc.

From Table 1 we conclude that, at each iteration, the learning algorithm is successful in generating a solution which outperforms its opponent. This can be derived from the fact that the maximum fitness for each iteration is positive. To compare the individual results, all eleven AIs (the Rush AI and the ten generated AIs) were made to play against each other. For each of these AIs, their fitnesses against emphevery opponent are averaged in Figure 5 (this is different from the average listed in Table 1, which is the average fitness achieved against only its training opponent). Here, iteration number 0 represents the Rush AI.

Figure 5 shows clearly that the overall performance of the AIs does not increase with each iteration. Even though each AI outperforms the AI from the previous iteration, the average fitnesses may go up or down. This fact suggests that the results do not generalise well over the opponents from previous iterations.

For a better understanding of these results, we considered the types of tactics that the generated AIs use. We found that the AIs can be divided into three categories, each of which uses simple tactics.

Iter.	Minimum	Average	Maximum	Opponent
1	-0.293	0.129	0.473	-0.129
2	-0.275	0.059	0.326	-0.059
3	-0.060	0.155	0.486	-0.155
4	-0.416	-0.027	0.286	0.027
5	-0.021	0.318	0.591	-0.318
6	-0.252	0.141	0.500	-0.141
7	-0.533	0.057	0.357	-0.057
8	-0.022	0.025	0.302	-0.025
9	-0.425	0.053	0.300	-0.053
10	-0.212	0.111	0.457	-0.111

Table 1: Final fitness values for Experiment 1.

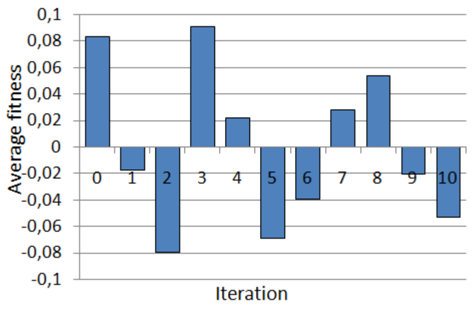

Figure 5: Average fitnesses of all AIs.

1. The Defence tactic. This tactic keeps units close to each other at the base, and only attacks when provoked. It is used by AIs #1 and #5.

2. The Base Offence tactic. This tactic entails rushing the enemy bases with all units, and attacking enemy units whenever possible. It is used by AIs #2, #4, and #7.

3. The Unit Offence tactic. This tactic is similar to the Base Offence tactic, but it moves towards enemy units instead of bases. It is used by AIs #3, #6, #8, #9, and #10.

Figure 6 shows the performances for each of these types of tactics against each of the others, as well as against the Rush tactic. Each of the four graphs contains the average fitness values for the corresponding tactic against each of the four tactics. From this figure we can observe the following. The Rush AI outperforms the Base Offence and Unit Offence tactics, but is outperformed by the Defence tactic. In turn, the Defence tactic is outperformed by the Base and Unit Offence tactics. This implies that none of the tactics dominates the others. It seems that, as is the case with the units, a rock-paper-scissors relationship exists at the level of strategies.

As a side note, we add that no Fog of War (FoW) was used for these experiments. While experiments with FoW were performed, their results were similar to the results described here. A possible explanation for this is that, for this environment, FoW has little effect on the game. Details concerning the FoW experiments are given by Bergsma [1].

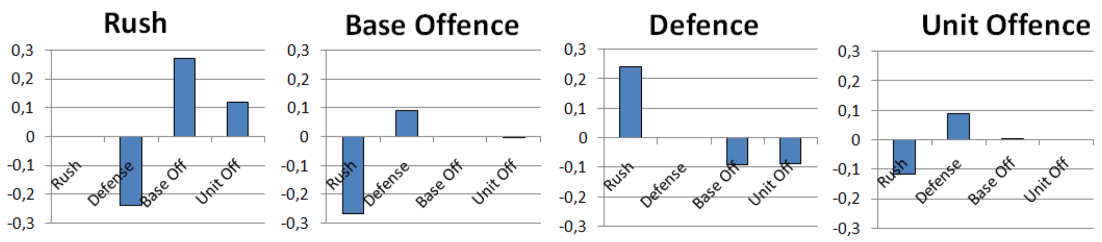

Figure 6: Fitnesses between each type of tactic.

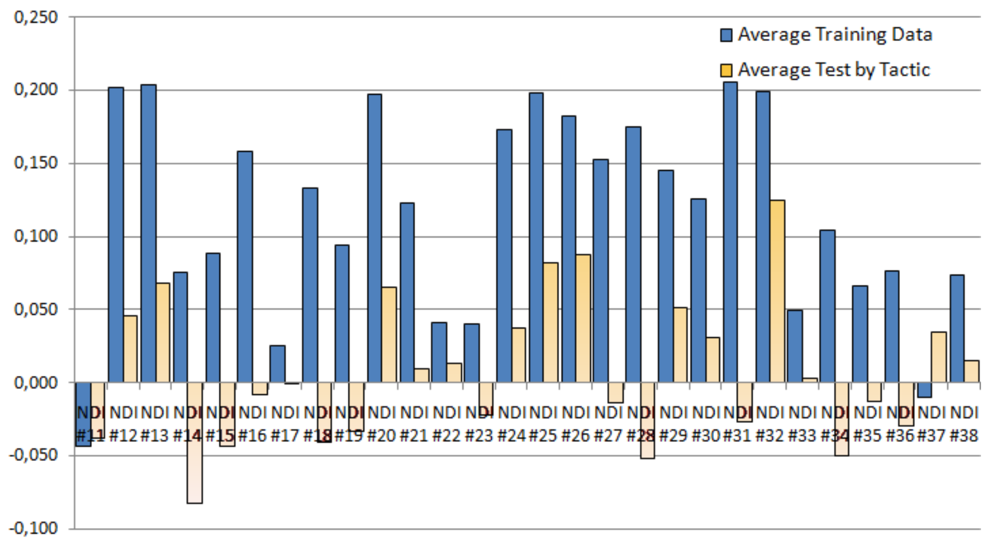

Figure 7: Average fitness of each non-dominated individual against the training set and the test set. The latter is weighed according to the prevalence of each tactic in the test set.

7 Experiment 2: Multi-Objective Learning

Because each generated AI's performance was strongest against their direct opponent, it is expected that an AI which is learned against multiple opponents will be relatively strong against each of them. If each of these opponents uses a different tactic, the AI must learn to outperform each of these tactics simultaneously, which potentially results in a generalised AI.

In the second experiment we decided to generate an AI against three of the different AIs from the first experiment, namely #0 (the Rush AI), #1 (a Defence tactic) and #3 (A Unit Offense tactic). The approach we used for this experiment is based on the SPEA2 algorithm [8]. It uses the concept of Pareto dominance to establish an ordering between different individuals. An individual is Pareto dominated if another individual's fitness values are at least equal to the first individuals fitness values, and at least one of these values is greater. Using this concept, non-dominated individuals can objectively be considered better than their dominated counterparts.

The SPEA2 approach differs from regular EAs in two important ways. Firstly, aside from the regular population, an archive containing non-dominated individuals is maintained. Each new population is now generated using the individuals from not just the population, but also from this archive. Secondly, two factors are used to determine the scalar fitness value of individuals, namely (1) the number of individuals which dominate this individual and (2) the location of this individual in the fitness space.

At the end of the second experiment (which we ran only once due to the time-intensive nature of the task), the learning algorithm had generated 28 non-dominated individuals. To determine the quality of these AIs, they were played against the test data, which consists of the previously generated AIs which were not used as training data (AIs #2 and #4 through #10). The resulting averages of the fitness values against both the training and the test sets are displayed in Figure 7. The test set averages are weighted according to the prevalence of their corresponding tactic within this set.

Unsurprisingly, the results against the test set are generally lower than those against the training set. However, in some cases the differences are exceedingly large. For example, the AI with the highest average fitness against the training set, Non-Dominated Individual (NDI) #31, has a negative average fitness against the test set. The fitness values for this individual imply that a single AI using a certain observed tactic, is not necessarily representative for other AIs using the same tactic. The test results in this figure also show a clear favourite; NDI #32 outperforms the other AIs by a wide margin.

To obtain a more accurate measure of the generated AIs' performances, they were also made to play each other in a tournament. Again, NDI #32 proved to be the best-performing AI.

Analysing the behaviour of NDI #32 shows an interesting tactic. It does not simply rush toward enemy units or bases, but it is not strictly defensive either. For the first few turns, this AI does not move its units, until any attacking enemy units move into attack range. Then, the AI sends all of its units towards these enemy units. Because the enemy's units move at different rates and therefore do not arrive simultaneously, this results in a material advantage as well as the initiative of attack. This behaviour shows that the learning algorithm is able to generate an AI which not only outperforms multiple tactics, but does so using a new, somewhat more complicated tactic, instead of an improved version of the previously generated tactics.

8 Conclusions

The goal of this research was to generate automatically an effective TBS AI player. To accomplish this, our approach focussed on spatial reasoning, through the use of influence mapping. In this work, we have extended the influence mapping approach by implementing an improved layering algorithm, and by creating a learning algorithm which generates the weights, and therefore the behaviour, of the influence maps automatically. We have also shown that influence maps can be used to determine directly the behaviour of an AI player.

Moreover, in order to decompose the complex task of creating a TBS AI player, we proposed the ADAPTA architecture. This architecture makes it possible to concentrate AI design on a single subtask. This promotes the possibility to implement learning AIs.

In our experiments, we chose one subtask, namely an Extermination module which is aimed at combat. For this subtask we learned new AIs using an evolutionary algorithm. The results achieved showed that the ADAPTA AI is able to generate tactics that defeat all single opponents. Moreover, by learning against multiple opponents using different tactics simultaneously, an AI was created which was able to play at least equally well and outperform most of the previously generated AIs.

For future work, we intend to explore the capabilities and limitations of the auctioning mechanism in the strategic layer of the ADAPTA architecture, as this was not included in the present research. Furthermore, we intend to explore whether the successes achieved with learning of the Extermination module can be repeated for the other modules.

Acknowledgements

This research was funded by a grant from the Netherlands Organization for Scientific Research (NWO grant No 612.066.406).

References

[1] M. Bergsma. Adaptive spatial reasoning for turn-based strategy games. Master's thesis, Maastricht University, 2008.

[2] M. Buro and T. Furtak. RTS games as test-bed for real-time AI research. *Workshop on Game AI, JCIS 2003*, page 20, 2003.

[3] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, 1989.

[4] M. Ponsen, H. Muñoz-Avila, P. Spronck, and D.W. Aha. Automatically generating game tactics with evolutionary learning. *AI Magazine*, 27(3):75–84, 2006.

[5] P. Tozour. Influence mapping. *Game Programming Gems*, 2:287–297, 2001.

[6] P. Tozour. Using a spatial database for runtime spatial analysis. *AI Game Programming Wisdom*, 2:381–390, 2004.

[7] X. Yao. Evolving artificial neural networks. In *Proceedings of the IEEE*, volume 87, pages 1423–1447, 1999.

[8] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical Report 103, Swiss Federal Institute of Technology, Zurich, Switzerland, 2001.