Degree project

Dynamic Strategy in Real-Time Strategy Games

with the use of finite-state machines

Author: Marcus SVENSSON
Supervisor: Johan HAGELBÄCK

Date: 2015-01-30

Course Code: 2DV00E, 15 credits
Level: Bachelor

Department of Computer Science

Abstract

Developing real-time strategy game AI is a challenging task due to that an AI-player has to deal with many different decisions and actions in an ever changing complex game world. Humans have little problem when it comes to dealing with the complexity of the game genre while it is a difficult obstacle to overcome for the computer. Adapting to the opponents strategy is one of many things that players typically have to do during the course of a game in the real-time strategy genre. This report presents a finite-state machine based solution to the mentioned problem and implements it with the help of the existing Starcraft:Broodwar AI Opprimobot. The extension is experimentally compared to the original implementation of Opprimobot. The comparison shows that both manages to achieve approximately the same win ratio against the built-in AI of Starcraft:Broodwar, but the modified version provides a way to model more complex strategies.

Keywords: Real-time strategy, Game AI, Finite-state machine, Starcraft

Preface

I want to thank my supervisor for being extraordinary helpful and making it possible to work with this interesting topic. I also want to express my gratitude to my family, relatives and friends who provided help and pushed me to finish my degree project.

Contents

1 Introduction

This chapter gives a brief summary of what real time strategy games are and some of the challenges that exist when it comes to constructing computer controlled players (bots) within the genre. It also gives an introduction to the subject of the report.

1.1 Background

Real-time strategy (RTS) is a popular game genre that has grown in interest in the academic field of artificial intelligence (AI). The game genre introduces constraints which makes it impossible to use the same solutions that have been applied to board games such as chess. RTS games update the game-state repetitively many times each second and all players can perform actions simultaneously. This combined with the fact that there are many different actions that can be executed allows the game to be in an enormous amount of possible states.

Some of the current research challenges within RTS AI involve: planning, learning, handling uncertainty, spatial and temporal reasoning and domain knowledge exploitation [1]. These are complex problems that are easy for human players to comprehend while it is a big obstacle to overcome for the computer. It is easy to solve single situational scenarios to these challenges, but it becomes incredibly hard to provide a design that can handle a large amount of possible scenarios in a dynamic way.

AI players do have the advantage of being able to execute a huge amount of actions with perfect precision within a short time-frame. Another advantage is the human factor, an AI does not forget to execute any actions during the game.

A strategy is usually denoted as a main plan of what the player should do during the course of the game. Primarily the plan contains infrastructure and military investments but it might as well contain decisions about attack timings and economical expansion. During the course of a game players commonly deviate from the original strategy to be able to respond better to the current game situation. This report focuses on strategy adaptation in RTS games and will use the game Starcraft:Broodwar [2] as a test-bed for research.

1.2 Previous research

Preuss et al, [3] applied fuzzy logic to achieve strategy changes. A set of predefined strategies were used and different variables were defined to describe the current game-state. Each strategy assigned a usefulness value to the defined variables that were used to calculate the most promising strategy at the given time. The authors concluded that the AI did adapt to the current situation and was an improvement compared to a previous static strategy.

There has been some interesting research when it comes to strategy prediction. The topic is of interest due to that if an opponent's strategy can be predicted, that information can be used to adjust your own strategy. Data mining in conjunction with machine learning have been tested in [4]. The authors collected a large dataset consisting of game replays of professional players. Different machine learning algorithms were used on the collected dataset and compared based on recognition rate, performance and resistance to noise. The same dataset was used by Synnaeve and Bessiére [5] who presented a Bayesian model to predict the opponent's opening strategy. The outcome showed promising results as it had a quite high recognition rate and showed strong robustness to noise.

Other research exists that have focused on case-based reasoning [6], [7], [8] and goal-oriented architectures [9], [10]. A comprehensive summary of research in the area of RTS

AI can be found in [1].

1.3 Problem definition

Strategy adaptation in RTS games is one of many issues that AI have to deal with. Experienced players usually start out the game with a predefined strategy which is a description of what units, upgrades and buildings they want to have at certain stages of the game. In the game Starcraft the opponent's race, the map being played and knowledge about the opponent's play-style are factors that are taken into account when a strategical plan is formed. In many cases the player will deviate from the original strategy to be able to handle the current game situation better.

1.4 Purpose

To be able to test strategy adaptation practically, the existing Starcraft:Broodwar bot Opprimobot will be extended with functionality to support it. Today Opprimobot changes strategy in a linear fashion where each state only have a single transition. As an improvement, a finite-state machine will be implemented to be able to model more complex strategies. The report aims to answer the following research question: Is a dynamic strategy version of Opprimobot as good as the present linear strategy version in terms of win ratio? In addition this also raises the sub-question: Under what conditions does the dynamic strategy version of Opprimobot achieve a higher win rate than the present linear strategy version?

1.5 Scope

The AI will make use of the possibility to have complete map information. By ignoring the fog of war, the uncertainty factor is removed and the implementation will not have to deal with possible noise from incomplete game-state data.

1.6 Target group

The primary target group is software developers that work with game AI.

1.7 Social and ethical aspects

Video games are a huge entertainment industry in today's society. Improving the set of tools software developers have in the industry can help in creating better video games in many aspects. Like with many things that are fun there is always a risk of addiction which makes it important to keep things at a moderate level, both for the consumers and producers.

1.8 Outline

The next chapter will introduce the game Starcraft:Broodwar and finite-state machine which is essential to have an understanding of when reading the report. After that there is a method chapter that describes the scientific approach that has been used. This is then followed by Chapter 4 that focuses on the implementation of the solution. The report then presents the results together with an analysis in Chapter 5. To finish off the report, everything is tied together with a discussion part (Chapter 6) and ends with a conclusion in Chapter 7.

2 Background

The following chapter gives an introduction to the game Starcraft:Broodwar and the computational model finite-state machine.

2.1 Starcraft

Starcraft:Broodwar is a famous RTS game that was released 1998 by Blizzard Entertainment [2]. The game takes place in a science-fiction universe and features three playable races: Terran, Zerg and Protoss. Each race promote different ways to play the game as no units or buildings are shared between the races, instead each race has a completely unique set of buildings, upgrades and units. Despite this the game is considered extremely well balanced and has been extensively used in e-sports. Since 2010 annual bot competitions have been held in the game as a way to promote and evaluate the progress of RTS AI research.

Like many other RTS games Starcraft makes use of a concept known as fog of war that introduces incomplete information where the player is not present. This gives an uncertainty factor that encourage players to perform reconnaissance in order to gain useful information about the enemy. See Appendix A for more detailed information on fog of war in the game.

The core of the game revolves around collecting resources as it is needed to produce units, construct buildings and research upgrades. There exists two types of resources (minerals and vespene gas) that are located in clusters across a map. Maps have a finite amount of resources that can be gathered, therefore a critical part of the game is to secure new resource locations in order to keep or increase the income of resources.

Another important aspect of the game is the units and supply. Each player have a supply counter which denotes used and available supply that is used to limit the size of a player's military force. The used supply is increased when units are produced and decreases when units die. Units can be produced until the supply cost of a desired unit makes the used supply exceed the available supply. Then the player must increase the supply by building supply-increasing buildings or units, but there is a maximum limit at 200 supply. In Figure 2.1 the supply is displayed in the top right corner as 78/100 meaning there is a total of 22 supply left that can be used for units until the limit is reached.

There are a wide range of units which have different properties and abilities, one of the most notable ability being invisibility (which requires specific units or buildings to reveal them). When forces from different players engage in battle, strategical positioning and unit control can impact the outcome of the battle. In the end the main goal in a game of Starcraft is to defeat your opponent(s) by destroying all enemy buildings.

2.2 Finite-state machine

According to Rabin [11], one of most reoccurring software pattern in game AI development is the finite-state machine (FSM). Its vast popularity can be explained by the qualities that FSMs posses e.g. they can be applied to many different AI problems and are easy to understand and implement.

A finite-state machine consists of a finite set of states where only one state at a time can be active. Usually a single state is marked as the starting state and decides where the computation begins. To be able to move from one state to another, a transition has to be defined. Conditions and events are tied to a transition and used to explain when a transition is supposed to trigger. Figure 2.2 displays an example of a finite-state machine.

Figure 2.1: A screenshot of Starcraft:Broodwar.

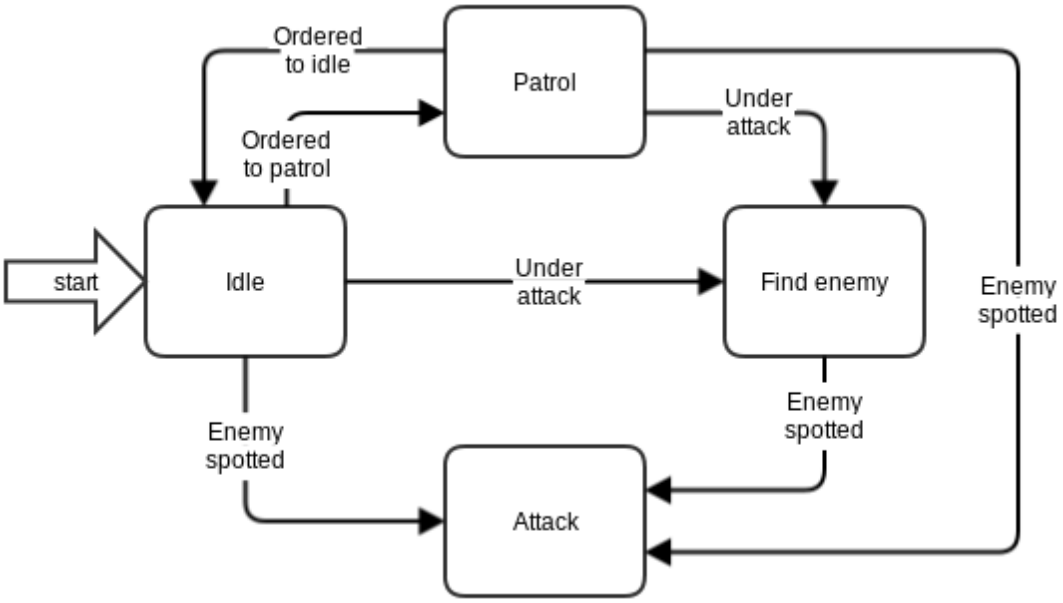

Figure 2.2: Example of a finite-state machine for a game AI that guards an area. The AI toggles between patrolling and idling until it reacts to hostile activity.

3 Method

A quantitative study has been done where a bot using a static strategy has been compared with one that uses a dynamic strategy. Data has been collected through an experiment where the bots have played several matches with the same settings and criteria:

- The bot will play as Terran every game.
- The opponent will be the built-in Starcraft AI playing all three races.
- The games will be played on the maps Match Point, Tau Cross, Andromeda and Python.
- The bot will play 20 matches against each race and repeat this for every chosen map (240 matches in total).

240 games will give a relatively good overview and some insight if there will be any differences depending on the played map or the opponent's race. Each game will produce statistics of the bot's in-game score, the enemy's in-game score and if the game was won or lost. This can later be used to compare how the two Opprimobot versions performed by looking for differences in the collected data. The win rate will give the best image of how well the bot performed as the in-game score depends on several factors where one of the most notable is how long the game lasted. Instead the in-game score accumulated should be seen as an indication of trends on different maps or when playing against a certain race.

There are reasons behind the chosen maps and the decision to play against an AI opponent. Letting the AI play against another AI is preferred instead of playing against a human controlled opponent due to two reasons: the game can be played on a much higher speed (allows the experiment to be done faster) and bots usually have a more static playing style. The maps that were chosen to be played during the experiment are maps that have been used in e-sport. These maps have a symmetrical design which gives both players the same starting conditions. Different strategies and play styles are encouraged depending on the map, this can for instance be that they differ in size, location of resources and the amount of supported players (ranges from 2-4). For more detailed information about the maps see Liquipedia [12].

4 Implementation

This chapter introduces the Starcraft:Broodwar bot Opprimobot (previously known as BTHAI) and its architecture. The modifications and additions which have been done to the strategy system of the bot are explained below.

4.1 Opprimobot

Opprimobot [13] [14] is a fully-featured Starcraft:Broodwar bot that uses the BWAPI [15] framework to communicate with the game engine. Opprimobot provides a flexible architecture which makes it quite easy to do changes or additions to the implementation. Therefore it is a very useful tool to conduct research on within the field of RTS game AI.

The main architecture of the bot divides classes into the three categories *Agents*, *Managers* and *CombatManagers* as seen in Figure 4.1. Two of the most crucial combat category classes are the commander and squad classes. The commander class has responsibility of many high level decisions that involve attacking, defending and what buildings to construct, upgrades to research and units to produce. The squad class is below the commander class in the hierarchy and has the responsibility of coordinating movement and attacks of the units that belong to the same squad.

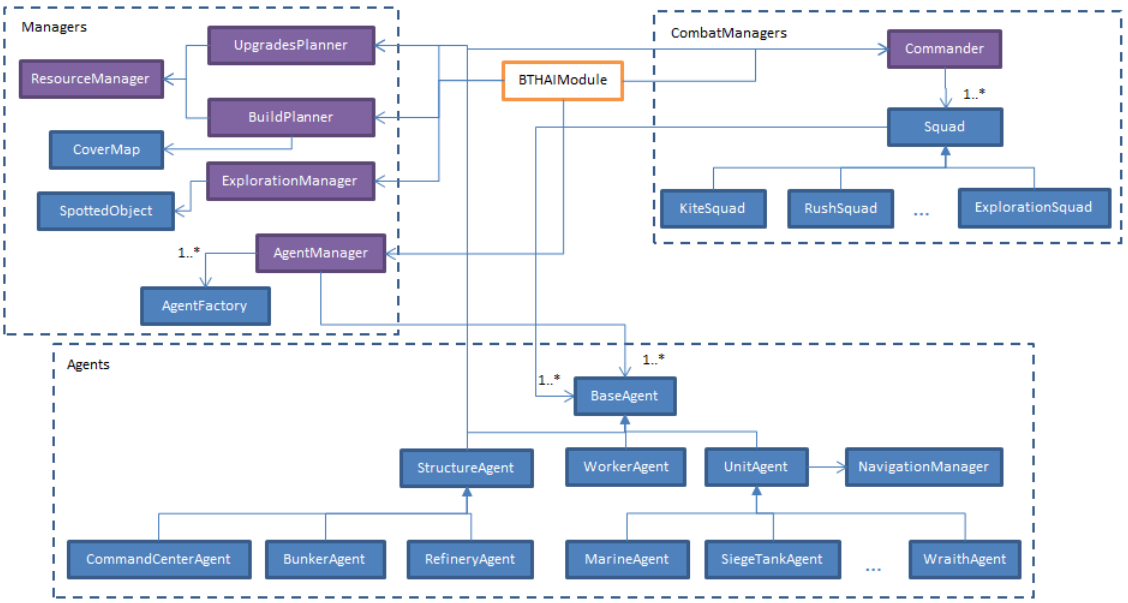

Figure 4.1: Overview of Opprimobot's architecture [16].

Agent classes define behaviour for units and buildings. Some agents are abstract and used to share similar traits, for instance there exists abstract agents of the types: unit, building and worker. There also exists a unique agent for every type of unit and building in the game that is used to specialize behaviour. Last is the manager classes which can be seen as a type of helper classes that provide functionality for e.g. building construction, unit production and map information.

Today the commander class makes use of static strategies which structure reminds of finite-state machines with the restriction that each state can not have more than one transition. For a graphical representation see Figure 4.2. In the implementation this is represented as a set of hard-coded if-statements which have conditions of when a transition to a new state occurs. Each transition is followed by a code block that contains

instructions that should be done when the transition happens. Readers who are interested in more information about Opprimobot can find this in [13] and [16].

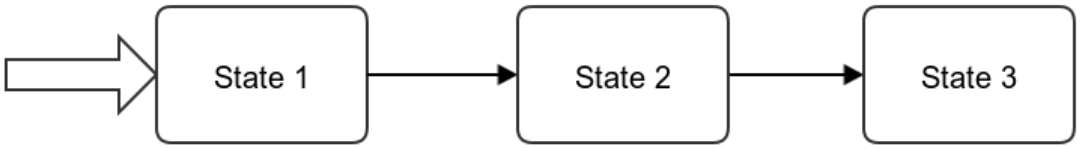

Figure 4.2: Model of how Opprimobot's static strategy work.

4.2 Adapting Opprimobot to dynamic strategies

As an improvement to Opprimobot's strategy handling it has been extended with a more general construction in the form a finite-state machine. The main improvement is an extended commander class that manages a finite-state machine. Its most important task is to keep track of when a state transition happens. When a transition occurs, information stored in the new state is read and redirected to different parts of the system which can process the data. Figure 4.3 shows an example of how a strategy can be designed with the modifications. For further reading on the implementation see Appendix B which contains a few code examples.

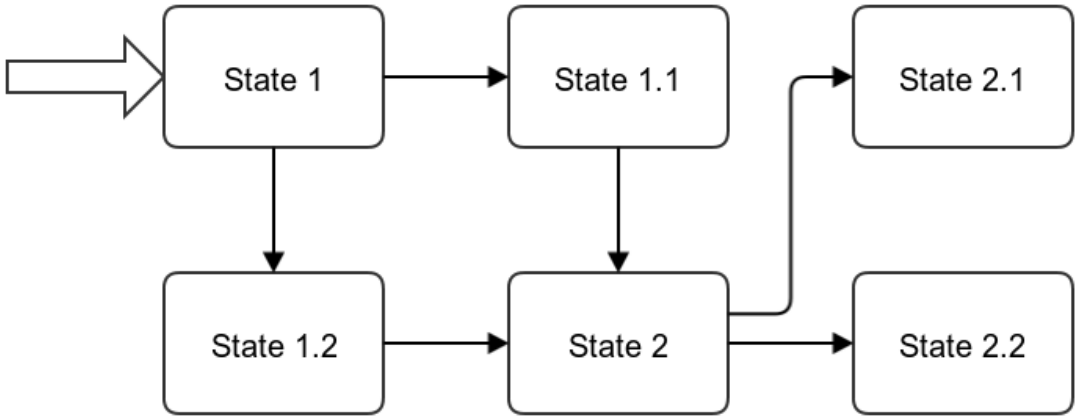

Figure 4.3: Example model of a strategy in the modified strategy system which uses a FSM architecture. The system allows a state to have multiple transitions which makes it possible to create more advanced strategies.

4.2.1 States

Each state in a FSM contains a list of transitions as well as a set of instructions. The instructions can for instance be to modify squads, queue buildings and launch an attack. Modifications done to squads involve adding and removing units from its setup and changing variables that control its behaviour. In the proposed implementation the instructions

focus on desired squad composition, upgrades, research and which buildings to construct. The upgrade and building instructions are redirected to managers that can queue the requests while instructions on unit composition are sent to a squad. For simplicity only a single squad is used that units can be appended to, but it can be extended to handle multiple squads and more actions than appending.

4.2.2 Transitions

Transitions have been implemented to allow the finite-state machine to change state and use conditions to choose between multiple transitions. The conditions are based on supply count and enemy buildings and units. The supply count condition only checks if the current supply is greater or equal to the given number. It can be used as either a way to stay at states for a longer period of time or transition to other states without the need of a condition that a certain unit or building has been spotted.

The other condition type with enemy buildings and units can be used as a way to respond to the opponent strategy. It is allowed to add multiple conditions of this type but they will be checked together as an or-statement, thus if one of the conditions is true it will be accepted. The implementation also provides the choice of using either *greater than or equals to*, *equals to* or *lesser than or equals to* as well as the number it should check against. It would be possible to introduce other conditions such as resource count and finished upgrades. A complete condition for a transition could for instance be formulated as: Supply count >= 80 AND (enemy building x count == 1 OR enemy unit y count >= 10).

5 Results and analysis

As stated in the method chapter each bot played 240 games as Terran on four different maps and against the three playable races in Starcraft (20 games for each unique combination). Table 5.1 and 5.2 show the win rate that the two bots achieved in the experiment.

	Terran	**Zerg**	**Protoss**	**Overall**
Match Point	100%	85%	100%	95%
Tau Cros	100%	85%	85%	90%
Andromeda	100%	70%	55%	75%
Python	100%	80%	65%	82%
Overall	100%	80%	76%	85%

Table 5.1: Win rate that the original Opprimobot (static strategy) achieved against the built-in Starcraft:Broodwar AI.

	Terran	**Zerg**	**Protoss**	**Overall**
Match Point	90%	80%	90%	87%
Tau Cros	95%	75%	85%	85%
Andromeda	85%	70%	65%	73%
Python	95%	70%	75%	80%
Overall	91%	74%	79%	81%

Table 5.2: Win rate that the extended Opprimobot (dynamic strategy) achieved against the built-in Starcraft:Broodwar AI.

Some trends can be seen when looking at how both AI performed on different maps as well as against different races. The static strategy bot had a better win rate on every map compared to the dynamic one. Both bots followed the same pattern when it comes to the overall win rate for each map. Starting from the map with the highest win rate the order was: Match Point, Tau Cross, Python and Andromeda. Match Point is the only map with two starting locations which means that the enemy starting location is always known. The two maps that the bots performed worst on are larger maps with open spaces in the middle of the map. This somewhat indicates that the bots performed worse on larger maps and also when there were more than one possible starting location for the opponent.

When it comes to how the bots performed against different races there are some interesting results. Both played best against Terran but the static strategy bot managed to achieve a perfect 100% win rate against Terran. However the dynamic strategy bot managed to win 3% more games against Protoss compared to the one with a static strategy. An important note against Zerg is that the built-in Starcraft:Broodwar AI sometimes randomly utilizes an extremely fast military attack named 4pool. This occurrence is also mentioned in [9]. Both the static and dynamic strategy bot can not deal with this due to the design of the used strategies as well as how the AI functions overall. This rush seems to occur approximately three times out of 20 games and explains some of the losses against Zerg.

An important aspect of the result is that a limited amount of games have been played. This raises the question if the same similarities and differences would remain if more

games were played in the experiment. For instance would the bots still perform worse on the larger maps and would the static strategy bot keep the 100% win rate against Terran. The played maps and the design of the used strategies could also have an impact on the result. For more data see Appendix C that contains charts based on the in-game score from the played games.

6 Discussion

To answer my research question the dynamic strategy bot had a 4% lower win ratio than the original one in the experiment. This difference is of no statistic significance as shown in Table 6.1 which presents a statistic test on the result of the experiment. The answer to the sub question is that the dynamic strategy bot only achived a higher win ratio under one condition, when playing against Protoss it achieved a 3% higher win rate compared to the static strategy bot. This tells that the proposed solution is able to perform slightly better or at least as good as the previous static strategy. The actual differences in win ratio between the two bots can most likely be explained by the design of used strategies in the experiment. The used strategy for the dynamic system was designed from scratch and is not built on the one that the original Opprimobot used. A guess would be that the actual design of a strategy has a larger impact on the playing-performance of the bot compared to the modifications that were done.

Games played by A	240
Win rate A	0.85
Games played by B	240
Win rate B	0.81
Alpha	0.05
P-value	0.12

Table 6.1: Pairwise two-sample t-test between proportion on the overall win rate. Alpha denotes the level of significance and the value 0.05 has been used. Since the P-value is greater than the Alpha value there is no statistic significance.

When it comes to how well the proposed solution actually deals with the problem it is able to do this to some extent, but it is far from being full-fledged as it does not have the ability of complex reasoning. Some of the most notable shortcomings that appear are:

- Economy is not taken into account and with a limited income investments should be prioritized. Currently Opprimobot uses separate data structures for buildings and units which do not cooperate, instead the first data structure that can afford an investment is given the right to do so. Another issue is when the bot has a high income it can not spend resources in the same rate as they are collected. Then it would be a great idea to create additional production and research facilities to better match the income as it would make good use of resources that otherwise would pile up.

- There is no real prediction of future enemy strategies or events, it is only capable of handling information based on the present time. For instance if an aggressive attack from the enemy is expected it is usually a good idea to prepare for it by prioritizing defensive investments. The same can be said about preparing detection for invisible units. The proposed implementation can do this to some extent by relying on indications from enemy buildings, but in the long run it does not provide a reliable solution.

- The last shortcoming is adaptation to the current situation. This can for example be to adjust the army composition by prioritizing and increasing the number of

wanted anti-air units and structures due to that the enemy has many flying units. The implementation can handle this to some extent but it is not dynamic or adaptive enough to be a reasonable solution. Instead fuzzy logic based on economical and military information would probably solve this sub-problem fine, similar to what Preuss et al. did in [3].

The mentioned problems are tied closely to the work done in strategy adaptation and should be considered when designing an AI. A FSM still could be a feasible solution. Instead of using raw instructions and numbers in the FSM, more abstract data would probably provide an easier model for an AI to reason about.

7 Conclusion

This report presented a finite-state machine solution as a way to achieve adaptive strategy. The game Starcraft:Broodwar and the bot Opprimobot have been used to be able to practically test the idea. Complete map information was given to the bots to remove the uncertainty factor and allow the finite-state machine to always choose the right transition. Results show that the original bot and the extended version achieved approximately the same win rate against the built-in AI of the game. The proposed solution works at least as well as the original static strategy of Opprimobot and provides a way to model more complex strategies. At the same time the solution is far from perfect as it keeps some of the problems that are tied to a static strategy and introduces new ones due to the finite-state machine design.

Further research could include information gathering. The goal of this would be to gather information about the opponent that later can be used for decisions in the higher level parts of the AI. This involves ways to schedule scouting with different tasks and also improving unit control so the scouting has a higher chance of succeeding. There are also different categories of information to keep track of such as economical, technological and military.

Another suggestion is a continuation of the work presented in this report. As mentioned in the discussion chapter the implementation lacks the ability of reasoning. This does however link to information gathering and domain knowledge, without any information there is really nothing the AI can reason about. Another option would be to extend the system with learning which would allow it to automatically extend or modify an existing finite-state machine.

References

[1] S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft," *IEEE Transactions on Computational Intelligence and AI in games*, 2013.

[2] Starcraft:Broodwar. [Accessed 22-Jan-2015]. [Online]. Available: http://en.wikipedia.org/wiki/StarCraft:_Brood_War

[3] M. Preuss, D. Kozakowski, J. Hagelbäck, and H. Trautmann, "Reactive Strategy Choice in StarCraft by Means of Fuzzy Control," in *Proceedings of the 2013 IEEE Symposium on Computational Inteligence in Games (CIG)*, 2013.

[4] B. G. Weber and M. Mateas, "A Data Mining Approach to Strategy Prediction," in *Proceedings of 2009 IEEE Symposium on Computational Intelligence and Games (CIG)*, 2009.

[5] G. Synnaeve and P. Bessière, "A Bayesian Model for Opening Prediction in RTS Games with Application to StarCraft," in *Proceedings of 2011 IEEE Symposium on Computational Intelligence and Games (CIG)*, 2011.

[6] D. C. Cheng and R. Thawonmas, "Case-Based Plan Recognition for Real-Time Strategy Games," *Proceedings of the Fifth Game-On International Conference*, 2004.

[7] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "Case-Based Planning and Execution for Real-Time Strategy Games," in *Proceedings of the Seventh International Conference on Case-Based Reasoning*, 2007.

[8] K. Mishra, S. Ontañón, and A. Ram, "Situation Assessment for Plan Retrieval in Real-Time Strategy Games," 2008.

[9] B. G. Weber, M. Mateas, and A. Jhala, "Applying Goal-Driven Autonomy to StarCraft," 2010.

[10] A. Dahlbom and L. Niklasson, "Goal-Directed Hierarchical Dynamic Scripting for RTS Games," 2006.

[11] S. Rabin, *Introduction to Game Development*, 1st ed. Boston: Charles River Media, 2005.

[12] Liquipedia. [Accessed 22-Jan-2015]. [Online]. Available: wiki.teamliquid.net/starcraft/Main_Page

[13] Opprimobot. [Accessed 22-Jan-2015]. [Online]. Available: code.google.com/p/bthai/

[14] J. Hagelbäck, "Potential-Field Based navigation in StarCraft," in *Proceedings of 2012 IEEE Conference on Computational Intelligence and Games (CIG)*, 2012.

[15] Brood War API. [Accessed 22-Jan-2015]. [Online]. Available: github.com/bwapi/bwapi/

[16] J. Hagelbäck, "Multi-Agent Potential Field based Architectures for Real-Time Strategy Game Bots," Ph.D thesis, Blekinge Institute of Technology, 2012.

A Appendix - Fog of war in Starcraft

Fog of war limits what information is available on the map for a player. The limitation of information gives a strategical depth to the game as the player has to gain needed information to be able to know what the opponent is doing. With insufficient information on the enemy, the player is left guessing and more likely to perform bad decisions. To make information available in an area, the player must have a building or unit nearby or use an ability that grants vision.

An area can be in one of three states: unexplored, previously explored and currently being explored. Unexplored areas show as a pitch black colour on the map. Areas that previously have been explored are grayed out and show buildings in their last known animation state. When an area is currently being explored, the player can see and target enemy units and buildings. A comparison of the two later can be seen in Figure A.1.

Figure A.1: A display of fog of war.

In the left picture of Figure A.1, the player has vision over an enemy area. An enemy building is being targeted and enemy worker units can be seen. This information is not available in the right picture where fog of war is in effect.

Starcraft also implements a ground level feature which is tied to the fog of war. This feature limits the vision for ground units to the same ground level and levels below that one. Important to note is that flying units ignore this feature. If a player does not have vision over an attacking unit or building from a higher ground level it will be revealed for a few seconds. Figure A.2 displays the difference between two ground levels with a ground unit.

Figure A.2: Visibility differences between two ground levels.

In the left picture of Figure A.2, a single unit is below a higher ground level which it does not have vision of. When standing on the higher ground level as displayed in the right picture, the player has vision over both height levels.

B Appendix - Code examples

The following listings show some of the key components of the implementation. Listing B.1 and B.2 show how a state and transition is created while Listing B.3 and B.4 display how a transition is triggered and handled.

Listing B.1: A state being created and filled with strategical information. The integer parameter in the addToUnitPlan function represents the amount of the specific unit that should be added.

```
StrategyState *state = new StrategyState();

// Building information
state->addToBuildPlan(UnitTypes::Terran_Engineering_Bay);
state->addToBuildPlan(UnitTypes::Terran_Missile_Turret);
state->addToBuildPlan(UnitTypes::Terran_Barracks);

// Upgrade information
state->addToBuildPlan(UpgradeTypes::Terran_Infantry_Weapons
   );
state->addToBuildPlan(UpgradeTypes::Terran_Infantry_Armor);

// Unit information
state->addToUnitPlan(UnitTypes::Terran_Marine, 25);
state->addToUnitPlan(UnitTypes::Terran_Medic, 5);
state->addToUnitPlan(UnitTypes::Terran_SCV, 1);
state->addToUnitPlan(UnitTypes::Terran_Siege_Tank_Tank_Mode
   , 6);
```

Listing B.2: Example of a transition being defined between two states. The transition requires the player to have at least 50 supply and includes two additional conditions, the opponent must have at least six Protoss Zealot or two Protoss Gateway.

```
StateTransition *t = new StateTransition(nextState, 50);

t->addCondition(UnitTypes::Protoss_Zealot, 6,
   StateCondition::GREATER_OR_EQUALS);

t->addCondition(UnitTypes::Protoss_Gateway, 2,
   StateCondition::GREATER_OR_EQUALS);

state->addTransition(t);
```

Listing B.3: Functions that checks if a transition occurs. If a transition's conditions are met a new state is returned.

```
vector<StateTransition*>* transitions = currentState->
   getTransitions();

// Loop through transitions
for (unsigned int i = 0; i < transitions->size(); i++)
{
   StateTransition* transition = transitions->at(i);
   StrategyState* state = transition->getState();
   // Check supply
   if (transition->getSupply() > currentSupply)
      continue;

   vector<StateCondition>* conditions = transition->
      getConditions();
   // No conditions
   if (conditions->size() == 0)
      return state;

   // Loop through conditions for the transition
   for (unsigned int j = 0; j < conditions->size(); j++)
   {
      StateCondition condition = conditions->at(j);
      int unitsFound = EnemyInfo::getUnitCount(condition.
         unit);

      // Check if the condition is true
      if (condition.type == StateCondition::EQUALS &&
         unitsFound == condition.quantity)
         return state;

      else if (condition.type == StateCondition::
         LESSER_OR_EQUALS && unitsFound <= condition.
         quantity)
         return state;

      else if (condition.type == StateCondition::
         GREATER_OR_EQUALS && unitsFound >= condition.
         quantity)
         return state;
   }
}

return currentState;
```

Listing B.4: Function that reads the strategical information from a state and redirects to other parts of the system. The code is called once for the starting state and every time a transition occurs.

```
// Add new items to the buildplan
vector<BuildplanEntry>* plan = currentState->getBuildPlan()
   ;
for (unsigned int i = 0; i < plan->size(); i++)
{
   BuildplanEntry entry = plan->at(i);

   // Build it at the current supply
   if (entry.supply == -1)
   {
      entry.supply = currentSupply;
   }
   buildplan.push_back(entry);
}

// Append units to squad
vector<UnitSetup>* unitplan = currentState->getUnitPlan();
UnitSetup us;

for (unsigned int i = 0; i < unitplan->size(); i++)
{
   us = unitplan->at(i);
   mainSquad->addSetup(us.type, us.no);
}
```

C Appendix - Charts of in-game score

This appendix contains charts based on how much in-game score that was earned in the experiment. The blue colour represents the static strategy and the red colour its opponent while the yellow represents the dynamic strategy bot and green its opponent. In Figure C.1 the summarized scores of all played games are shown where each staple represents 240 games.

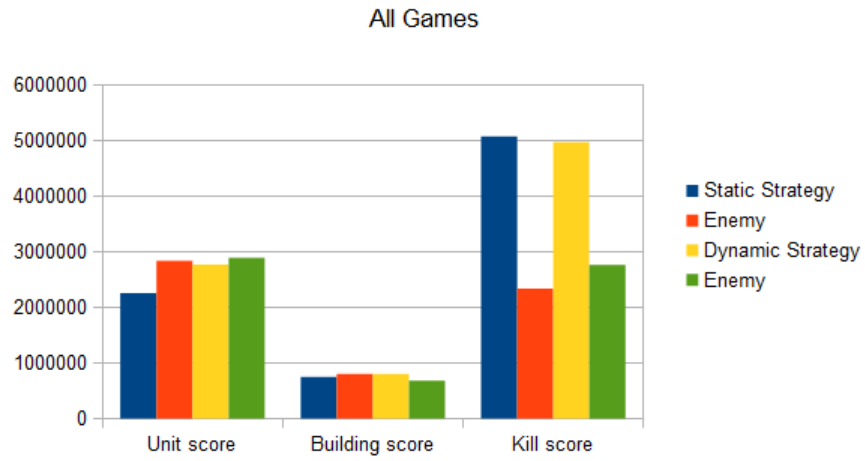

Figure C.1: Total in-game score accumulated over all games.

C.A Aranged by opponent race

The following charts show the total accumulated score based on the opponents race. A single staple in the charts represents 80 games.

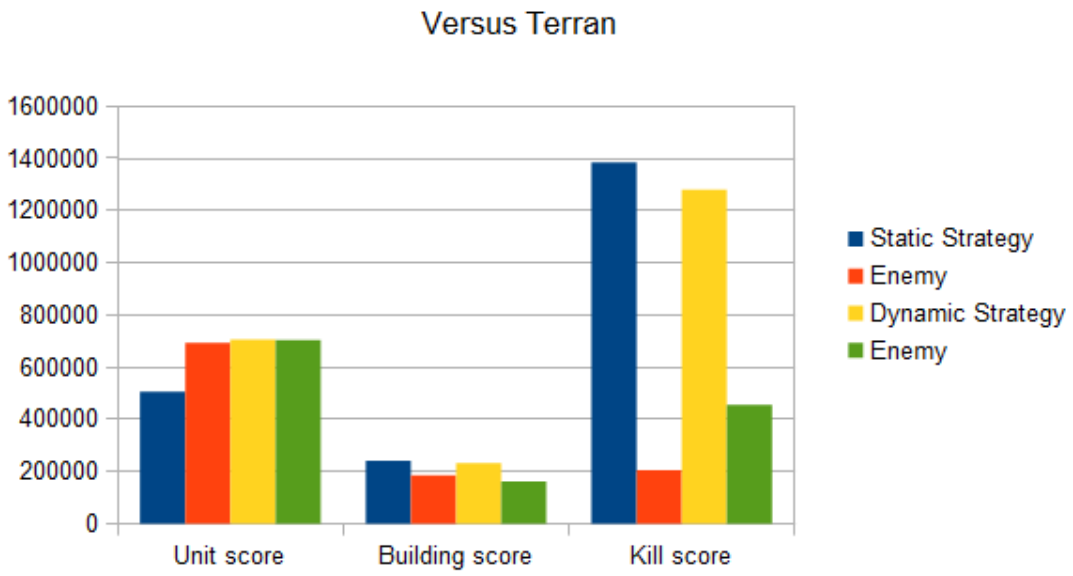

Figure C.2: Total in-game score when playing against Terran.

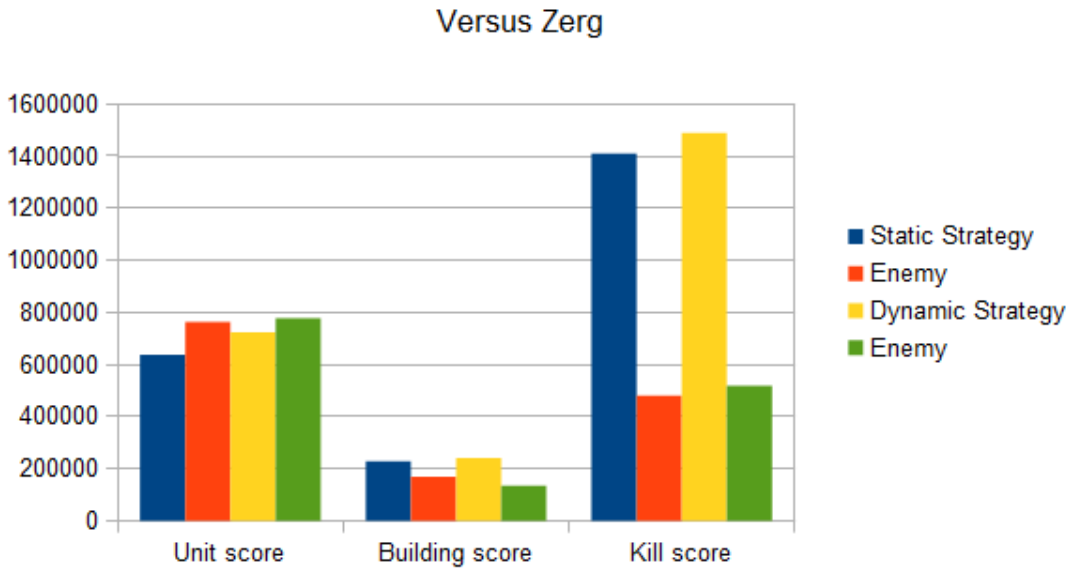

Figure C.3: Total in-game score when playing against Zerg.

Figure C.4: Total in-game score when playing against Protoss.

C.B Arranged by map

The following charts display the total accumulated score based on the map that was played. Every staple in the charts represents 60 games.

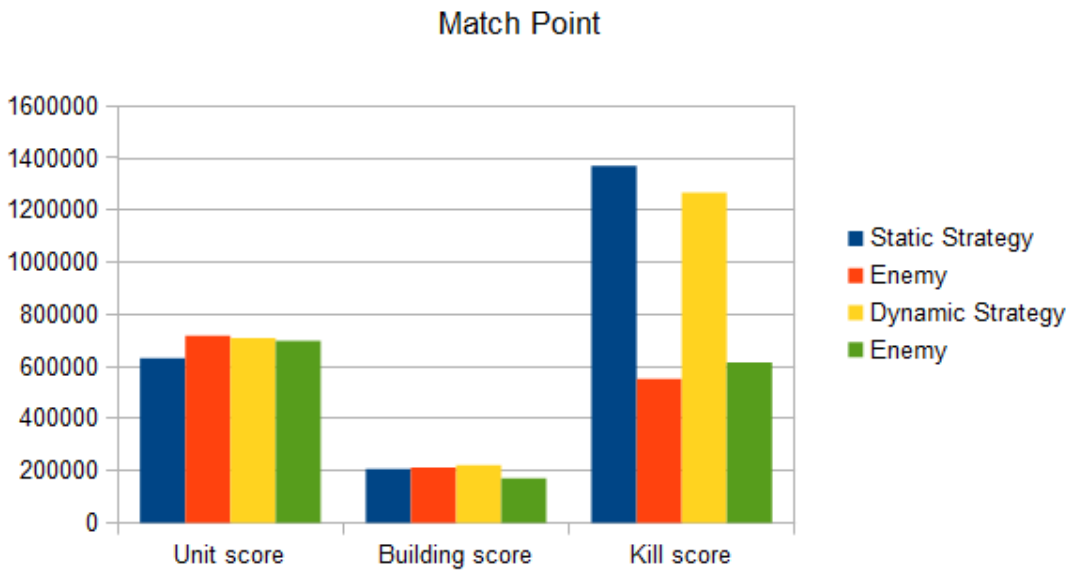

Figure C.5: Total in-game score accumulated when playing on the map Match Point.

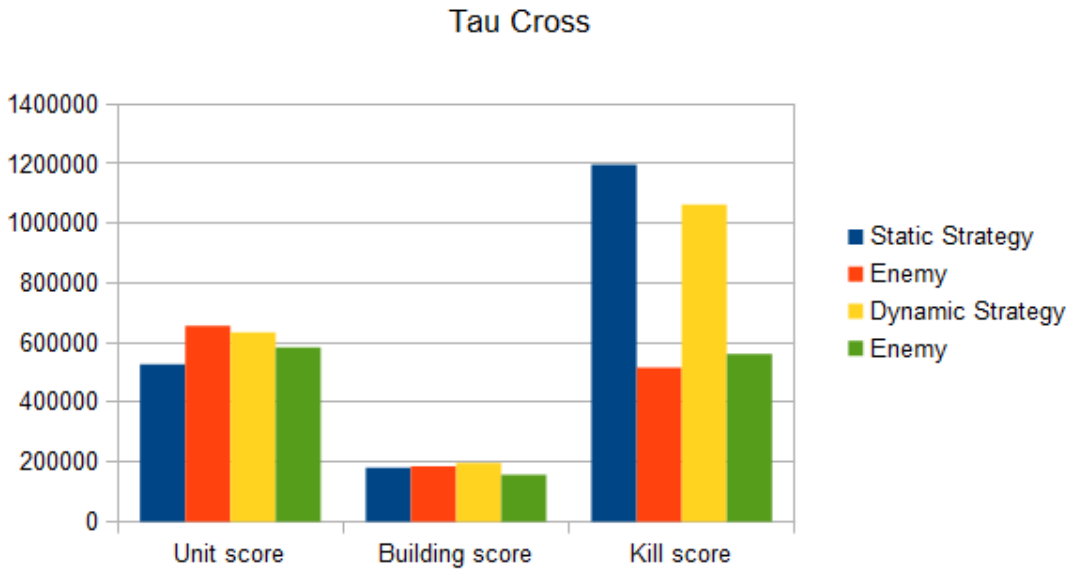

Figure C.6: Total in-game score accumulated when playing on the map Tau Cross.

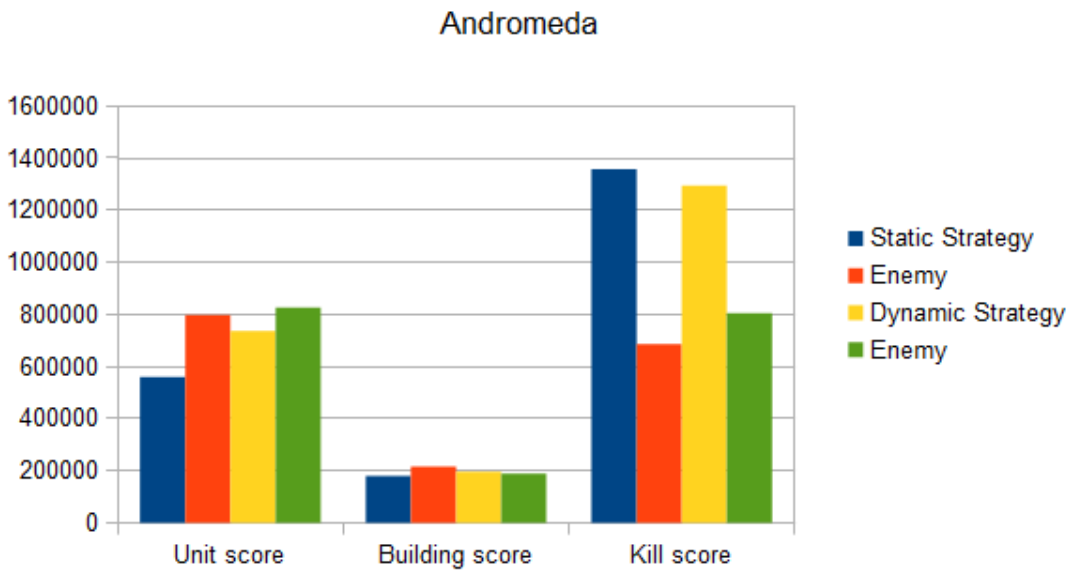

Figure C.7: Total in-game score accumulated when playing on the map Andromeda.

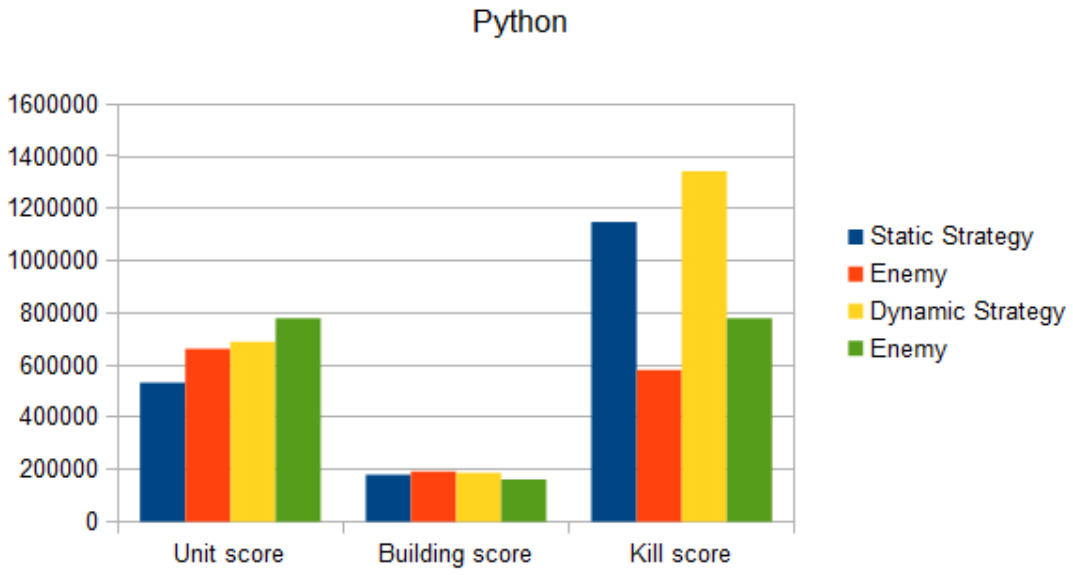

Figure C.8: Total in-game score accumulated when playing on the map Python.

Lnu.se

Linnæus University
Sweden

Faculty of Technology
SE-391 82 Kalmar | SE-351 95 Växjö
Phone +46 (0)772-28 80 00
teknik@lnu.se
Lnu.se/faculty-of-technology?l=en