DATA-DRIVEN RESOURCE MANAGEMENT IN REAL-TIME STRATEGY GAMES

DION CHRISTENSEN, HENRIK OSSIPOFF HANSEN, LASSE JUUL-JENSEN, KASPER KASTANIEGAARD

DAT6, Machine Intelligence

February 2011 – June 2011

Dion Christensen, Henrik Ossipoff Hansen, Lasse Juul-Jensen, Kasper Kastaniegaard: *Data-driven Resource Management in Real-time Strategy Games,* DAT6, Machine Intelligence, © February 2011 – June 2011

SUPERVISORS:
Yifeng Zeng

LOCATION:
School of Information and Communication Technology
Department of Computer Science
Aalborg University

TIME FRAME:
February 2011 – June 2011

ABSTRACT

As more replay data from real-time strategy games becomes available, it might be possible to utilise a data-driven approach in order to streamline resource management in this type of game. This thesis studies the application of a data-driven approach in exploitative and explorative resource management in real-time strategy games. Previous work by the authors is summarised, detailing an algorithm for efficient gathering of resources. The algorithm provides an increase in the amounts of gathered resources, and is shown to be more predictable than the built-in approach used by the test bed. Furthermore, the thesis touches upon base expansion. Based on expert knowledge, 28 features that may be considered when expanding have been identified. Using feature selection methods, subsets containing 15 features are produced. A total of six different sets are tested using both artificial neural networks and decision trees. No subset shows a significant performance gain compared to the full feature set, indicating low noise of the data. The decision models using the feature sets are able to predict base expansions in replay data with a hit rate of up to 64.43%.

Artificial intelligence is the study of how to make real computers act like the ones in the movies.

— Anonymous

Intelligence is what you use when you don't know what to do.

— Jean Piaget

ACKNOWLEDGMENTS

We thank *Bo Hedegaard Andersen* for his enthusiasm and cheerful company during the project period. We would also like to thank *Grace Goodchild* for her input on the various English language curiosities we encountered throughout the scope of the project.

CONTENTS

LIST OF FIGURES

Colour figures are presented in Appendix C in higher resolution.

LIST OF TABLES

LIST OF ALGORITHMS

ACRONYMS

ANN Artificial Neural Network

BWAPI BroodWar Application Programming Interface

BWTA BroodWar Terrain Analyzer

DLL Dynamic Link Library

GPL GNU Public License

MAE Mean Absolute Error

ORTS Open Real-Time Strategy

PFS Perceptron Feature Selection

RTS Real-Time Strategy

SBS Sequential Backward Selection

SFFS Sequential Forward Floating Selection

SFS Sequential Forward Selection

1 INTRODUCTION

The problem domain of this thesis is the feasibility of utilising machine intelligence methods for gathering resources and choosing expansion locations in Real-Time Strategy (RTS) games, using a data-driven approach. An increasing availability of replay data has made RTS games interesting with regard to data-driven approaches. The work done in the area of resource gathering mentioned in this thesis is a reproduction of previous work done by the authors, Christensen et al. [5, 6].

This introduction will act as preliminary reading for understanding the problem domain, and will give a brief introduction to related work within the field of research. Preliminary reading specific to resource gathering will be presented in Chapter 3 and preliminary reading specific to choosing expansion locations will be presented in Chapter 8.

The exploration of the problem domain in this chapter is concluded with the problem statement in Chapter 2.

1.1 REAL-TIME STRATEGY GAMES

In a typical RTS game, two or more players battle each other for map domination by issuing real time commands to multiple units [11]. Some units will be workers, others will be combat units. Worker units are used to build structures and collect the resources needed to build new units and structures. Combat units are used to battle opponents in order to destroy their units and structures to ensure control of the map. Units are usually split up in a range of different types, e.g. airborne, ground and naval, each with their own strengths and weaknesses. In order to win a match in an RTS game, a player needs to make sound strategic decisions, and sustain a healthy income of resources in order to finance their army. A player usually needs to secure other resources than those immediately available. In order to secure new resources, a player needs to seek out a strategic location while maintaining the financial backbone for setting up a defensive perimeter.

1.1.1 *Choice of Real-time Strategy Game*

In order to facilitate experiments, an RTS game must be chosen as test bed. This section will present RTS games commonly used for research purposes, and will be concluded with a selection of the RTS game to be used as a test bed throughout the thesis.

1.1.1.1 *StarCraft: Brood War*

StarCraft is an RTS game developed by Blizzard Entertainment, released in 1998. It is, at the time of writing, considered the best selling strategy game of all time [4] and it is very popular in eSports tournaments. In Korea, professional StarCraft players are in some sense comparable to professional athletes; idolised by fans and receiving six digit pay checks [9]. Late in 1998, a popular expansion pack, *StarCraft: Brood War*, was released, adding new content to the game. A screenshot of StarCraft: Brood War can be seen in Figure 1.1.

StarCraft and its expansion are closed source, but implementation of artificial intelligence methods in the games can be achieved using Dynamic Link Library (DLL) injection using the tool Chaoslauncher[1]. Loadable modules can be built using BroodWar Application Programming Interface (BWAPI)[2] and BroodWar Terrain Analyzer (BWTA)[3]. These tools give developers the ability to view and control units in StarCraft, facilitating artificial intelligence research within the game.

Figure 1.1: An excerpt from a StarCraft match.

1.1.1.2 *Wargus*

Wargus[4] is a modification of Warcraft II to make it compatible with the open-source cross-platform Stratagus engine[5]. Wargus requires a copy of Warcraft II for models and sounds. Opposed to Warcraft II, a player can make adjustments to in-game units, for example increasing their firepower, using the Lua scripting language. Wargus is written in

1 http://winner.cspsx.de/Starcraft/
2 http://code.google.com/p/bwapi/
3 http://code.google.com/p/bwta/
4 http://wargus.sourceforge.net/
5 http://stratagus.sourceforge.net/

C++, and since the source is freely available, the game is completely open to artificial intelligence research. A screenshot of Wargus can be seen in Figure 1.2.

Figure 1.2: An excerpt from a Wargus match.

Wargus has previously been used for research purposes; Chan et al. [3] use Wargus to test an online planner which uses means-ends analysis to pick the best sequence of actions to reach some specified resource goal. Their experiments show that the planner performs at the same level as a competitive human player, and perform significantly better than state of the art planning algorithms.

1.1.1.3 *ORTS*

ORTS[6] is a programming environment developed specifically for research on RTS games. The developers of ORTS state that commercial RTS games are closed source, and only offer implementation of research AI by hacking the games in some way. The source code for ORTS is freely available, licensed under GNU Public License (GPL), and the developers encourage researchers to consider ORTS as test bed for their research. A screenshot showing ORTS can be seen in Figure 1.3.

The developers state that commercial RTS games have security issues, in that games are simulated in full on a peer-to-peer basis, making it possible for malicious players to access the full game state. ORTS is designed as a server/client architecture, where only the individually visible parts of the game state are sent to the game clients.

ORTS is made in C++ and implements a homebrewed scripting language to define the game scenarios playable in the engine [2]. It is considered platform independent, and is known to run without problems on both Linux and Windows. Buro [1] motivates the research of artificial intelligence in RTS games, and suggests that researchers both use and contribute to the ORTS development.

6 http://skatgame.net/mburo/orts/

Figure 1.3: An excerpt from ORTS. The screenshot is taken from the official ORTS website.

1.1.1.4 *Own Real-time Strategy Game*

An RTS game could be created specifically for the work done in this thesis. In this way, there would be full control over what happens, and agents would be easier to implement. It would, however, require a large workload. Since the focus of this thesis is not the development of an RTS game, this seems impracticable.

1.1.1.5 *Summary*

In this section, four choices of RTS games have been presented. Due to the popularity of StarCraft, the possibilities made available by Chaoslauncher and BWAPI, and the great amounts of data available, StarCraft is deemed the most interesting test bed, and subsequent sections of this thesis will revolve around StarCraft. Additionally, StarCraft is non-colliding in terms of agents gathering resources, which makes research on resource gathering a less complex problem to deal with.

The popularity of StarCraft facilitates the use of data-driven approaches, due to an increasing amount of available data. The data is obtainable through replays made available on websites such as iCCup[7], where thousands of replays are open for download, indicating that StarCraft is very prominent in terms of replay data.

7 http://www.iccup.com/

1.2 RELATED WORK

Previous work on resource gathering and expansion strategies in RTS games are limited. This section contains related work on both resource management and expansion strategies, and an evaluation of the usefulness of the work.

1.2.1 *Resource Gathering*

Wintermute et al. [15] create an agent for controlling a player in the open source RTS engine, ORTS. The agent utilises Soar[8] which is a decision-making framework for developing intelligent agents that must choose actions in some specialised environment.

The agent uses a human-inspired mechanism for selecting information, referred to as an attention system. In this system, the position and size of a rectangular view field determines which information is currently available, and a focus point chooses which parts of the view field should currently be attended.

Wintermute et al. [15] state that ORTS is a minimalistic RTS game engine that does not implement abstract command features e.g. attacking and gathering resources. Most commercial RTS games provide these features. In order to provide Soar with a human-level interface, the most common commands are implemented through the use of finite state machines.

Resource gathering is handled through a dedicated coordinator which utilises the high level commands of the human-level interface. The coordinator works by using simple learning through discovering poorly performing gathering routes and switching workers in these routes to potentially better performing ones. The coordinator does not take collisions between workers into account. This is instead handled reactively when two workers are about to collide.

The agent implemented by Wintermute et al. [15] performed well for the resource gathering competition at AIIDE 2006, even though it made no attempts at calculating the optimal solution. Reactive heuristics in combination with a simple learning approach performed well enough to win the competition, collecting 28% more resources than their closest competitor.

1.2.2 *Expansion Strategies*

Weber and Mateas [14] use case-based reasoning for selecting build order strategies using the open source Warcraft II modification, Wargus, as test bed. They affirm that RTS players perform constant reconnaissance in order to counter the strategy of their opponent and argue that scouting is vital in RTS games.

8 http://sitemaker.umich.edu/soar/home

They developed a method which performs scouting through a heuristic and use the intelligence gathered for identifying similar game states. A strategy that has been shown successful against a similar game state is then selected as a response. In order to decide how well one strategy fares against another, several scripts representing different strategies were created and each script was tested by running it against the other scripts.

The case-based selection method was tested against several other AIs, including the built-in AI of Wargus. Experiments were performed with both perfect and imperfect information. The results indicates a better win rate for perfect information.

The primary useful points from Weber and Mateas [14] are that scouting for information is important for winning an RTS game. Though no experiments are done against human players, the experiments indicate that there is a connection between observable actions of the opponent and the future actions of this opponent, making strategy prediction a valid method for computerised opponents, at the very least. It is possible that human players are not predictable to the same degree as the scripted opponents used for this paper. However, it would be interesting to examine which human player strategies (if any) could be predicted from a collection of reference data.

2 PROBLEM STATEMENT

The scope of this thesis is data-driven resource management in RTS games, with a focus on games where agents collecting resources cannot collide with each other, game states cannot be fully observed, and expansions are needed to provide a sufficient flow of resources.

Resource management may be split into two categories; exploration and exploitation. Exploration deals with discovering or creating new resource gathering opportunities. Exploitation deals with collecting resources and optimising the use of the resource sites that have been claimed.

The categories have been synthesised into two tasks, i.e. resource gathering–the act of amassing resources from the resource sites that have been obtained–and base expansion–the act of creating expansions in order to gain access to new resources and further facilitate the collection of resources. In order to efficiently expand to new locations in the game environment, some decision model is needed.

This thesis summarises previous work by the authors, and deals with the problem domain of choosing expansion locations, using a data-driven approach. This may be done by identifying and utilising the features that are considered significant by human players, when choosing expansion locations.

StarCraft: Brood War will be used as test bed for any proposed solutions of both the problem of resource gathering and expansion strategies.

Part I

EFFICIENT RESOURCE GATHERING

3 INTRODUCTION

In RTS games, a player usually does not take direct control of the agents that have been assigned to gather resources. When an agent is ordered to gather resources, the agent will move to the resource site, gather an amount of the resources, return to the nearest resource deposit, and then continue to gather resources until a different order is issued. The build-in method used for gathering resources in StarCraft and StarCraft: Brood War is designed in this way.

In StarCraft, a standard game map contains several clusters of resource sites as seen in Figure 3.1. The crystals seen in the lower left and top right corners, along with the geyser to the right of the crystals in the lower left corner represents the two collectable resources of StarCraft.

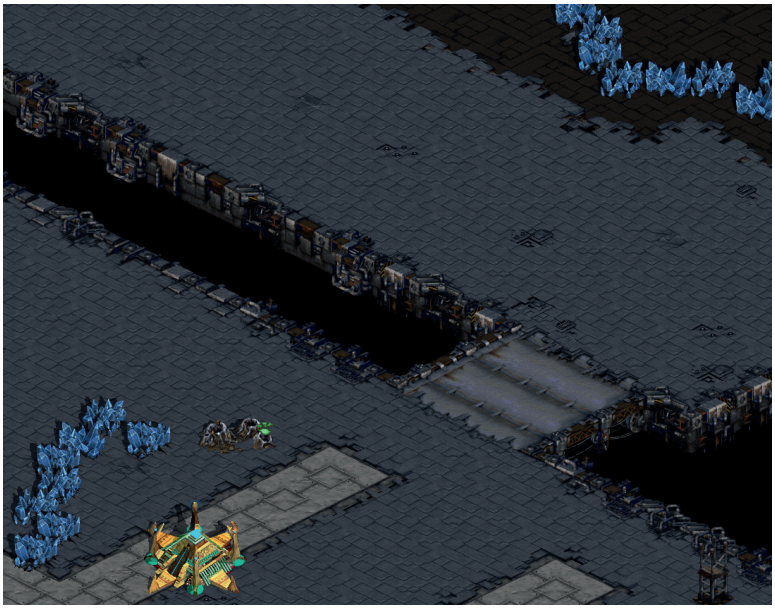


Figure 3.1: Screenshot showing the lower left corner of the official StarCraft: Brood War map Astral Balance.

Players of the game are able to build new resource deposits at will, and they start out with one close to a cluster, like the building in the lower left corner in Figure 3.1.

To avoid agents colliding with each other in the small area between a resource deposit and a cluster of resources, the mechanics in StarCraft is designed to disable collisions for agents gathering resources. As soon as an agent is given an order other than collecting or delivering resources, i.e. to move to a specific area, collisions are re-enabled for that agent.

Only one agent is allowed to gather from a resource site at a time. If a resource site is occupied when an agent arrives to gather resources, the agent will go to an unoccupied resource site in the cluster, or wait until the resource site is available. Choosing a different resource site may lead to the appearance of an agent regretting its previous goal in order to choose a new goal. This behaviour will, in some cases, result in the agent leaving a resource site that is becoming available shortly for a resource site that will be occupied shortly, thus wasting time on travelling. Waiting for availability may cause an agent to waste time waiting instead of moving to another resource site. Any time spent not moving to the correct resource site, or waiting for the wrong resource site, causes a loss compared to the optimal behaviour. Furthermore, the erratic movement may cause the resource income rate to spike or drop–when an agent chooses a new path–making it difficult to predict the amount of resources available at a later point in time.

To avoid these problems, direct control can be applied to the agents, where the future availability of the resource site is considered before moving. Since units gathering resources in StarCraft do not collide with one another, focus may be kept solely at scheduling.

4

ALGORITHM

This chapter introduces the domain of resource gathering formally, and describes an algorithm for resource gathering in this domain. The algorithm utilises a queuing system by using recorded travel times. These are then used for distribution of agents, in order to increase income.

4.1 RESOURCE GATHERING DOMAIN

Given a set of agents $A = \{a_i | i = 1, \ldots, n\}$ and a set of resource sites $M = \{m_j | j = 1, \ldots, l\}$ located in a two-dimensional Euclidean space, each site m_j having an attached amount of resources $r_j \in \mathbb{Z}^+$, choose a subset $S \subseteq G$ of gathering tasks $G = A \times M \times T$, where T is a set of time indices in the domain, such that the total amount of resources $R = \sum_{j=1}^{l} r_j$ is gathered in a minimal time mt.

All agents have a fixed maximum capacity for carrying resources, r_k and will collect $r_a = min(r_k, r_j)$ from a site before needing to unload their cargo at a resource deposit D. The time required for the actual gathering is a constant time C. For each site m_j there exists exactly one resource site queue Q_j containing the agents that will gather from this resource site next. A resource site queue is defined as a totally ordered set of agents $Q_j = \{a_h \in A | h = 1, \ldots, z_j\}$, $\forall a_h \in Q_j \implies a_h \notin Q_k$ where $j \neq k$, such that each agent is assigned to at most one resource site queue at a time. When an agent has finished gathering, it is removed from the queue. The first element $a_1 \in Q_j$ is the agent that may use the resource site first.

Let mt be the total time that is required to execute every gathering task $s_{i,j} = (a_i, m_j) \in S$. Specifically, a gathering task $s_{i,j,t}$ is completed in a round-trip time $rtt_{i,j}$ after agent a_i travels from D to a site m_j, potentially waits in line, collects resources and returns to D.

Equation 4.1 shows a calculation of the round-trip time, which aim to minimise by optimising efficient resource gathering through scheduling.

$$rtt_{i,j} = tt^i_{D,j} + max\left[0, rt_j - tt^i_{D,j}\right] + C + tt^i_{j,D} \,. \tag{4.1}$$

where $tt^i_{D,j}$ is the time required for the agent a_i to travel from depot D to the resource site m_j. rt_j is the remaining time for site m_j to become available after all agents in queue Q_j have completed their work. Thus, the time agent a_i would wait in line is the remaining time for the queue to become empty, rt_j, minus the time that has already

been spent on traveling. The constant collecting time C is also added along with the travel time to return to the resource deposit $tt^{i}_{j,D}$.

By using resource site queues, it is possible to determine the best site to send agents to, in order to minimise the time required for the agent to return to D with an amount of resources. Equation 4.2 states the remaining time for an agent h in queue Q_j to finish gathering the resource m_j. c_h refers to the remaining collection time of the h^{th} agent in the queue, $a_h \in Q_j$ where $0 \leqslant c_h \leqslant C$. Bear in mind that c_h is exactly C in every case other than the first agent in the queue.

$$rt^{h}_{j} = \begin{cases} tt^{h}_{D,j} + c_h & \text{for } h = 1 \\ rt^{h-1}_{j} + max\left[0, tt^{h}_{D,j} - rt^{h-1}_{j}\right] + C & \text{for } h > 1 \end{cases} . \quad (4.2)$$

The required time for agents ahead of a given agent for finishing the collection of resources from a resource site, is included in the recursive definition of rt^{h}_{j}. The total time of Q_j is therefore $rt_j = rt^{z_j}_{j}$, meaning the time required for the last agent in Q_j to finish gathering the resource.

The time required for an agent to finish gathering is dependent on the time required for the preceding agent, as gathering cannot occur before the preceding agent has completed gathering. The time required for the first agent in the queue, rt^{1}_{j} does not depend on any preceding agents, so only the travel time and the remaining gathering time needs to be included.

4.2 ALGORITHM DEFINITION

An algorithm was developed as a practical utilisation of equations 4.1 and 4.2. Minimising the round-trip time for an agent will cause the agent to always pick the resource site that will allow for the fastest delivery of an amount of resources in a greedy fashion. The main mining algorithm can be seen in Algorithm 1, along with the procedure `Work(Q,a)`.

The algorithm is run every time an agent needs to be assigned to a resource site queue.

In line 2, the algorithm iterates through every agent that is not currently assigned to a queue. Lines 3-10 iterates through the set of resource site queues to find the queue that requires the least amount of time for the agent to return with a deposit, and then adds the agent to the queue. $tt^{i}_{x,D}$ is the travel time of the agent a_i, from the resource site in the given iteration to the resource deposit.

The procedure `Work(Q,a)` accepts a queue Q and an agent a as parameters and returns the total time spent on traveling to the resource site, waiting in line and gathering from the resource site. The procedure works by recursively calculating the time required by the

Algorithm 1 Resource gathering algorithm

Declarations
$Q_1, \ldots, Q_l$: Resource site queues.
$tt^i_{x,D}$: Travel time from resource site x to resource deposit.
1: $time \leftarrow \infty$
2: **for all** $a_i \notin \bigcup_{j=0}^{l} Q_j$ **do**
3: **for** $x = 1 \ldots l$ **do**
4: $A \leftarrow Q_x \cup \{a_i\}$
5: **if** $Work(A, a_i) + tt^i_{x,D} < time$ **then**
6: $time \leftarrow Work(A, a_i) + tt^i_{x,D}$
7: $best \leftarrow x$
8: **end if**
9: **end for**
10: $Q_{best} \leftarrow Q_{best} \cup \{a_i\}$
11: **end for**

Parameters
Q: Resource site queue containing a.
a: Agent.
Returns
Time until a is removed from Q.
Declarations
$z = |Q|$: Number of elements in Q.
$p = h$ where $a = a_h \in Q$ and $1 \leqslant h \leqslant z$: Position of a in Q.
$tt^i_{D,j}$: Travel time from resource deposit to resource site j.
12: **procedure** WORK(Q, a)
13: $w \leftarrow 0$
14: **if** $p = 1$ **then**
15: **return** $tt^i_{D,j} + C$
16: **end if**
17: **for** $h = 1 \ldots p - 1$ **do**
18: $w \leftarrow w + Work(Q, a_h)$
19: **end for**
20: **if** $w > tt^i_{D,j}$ **then**
21: $w \leftarrow w - tt^i_{D,j}$
22: **else**
23: $w \leftarrow 0$
24: **end if**
25: **return** $tt^i_{D,j} + w + C$
26: **end procedure**

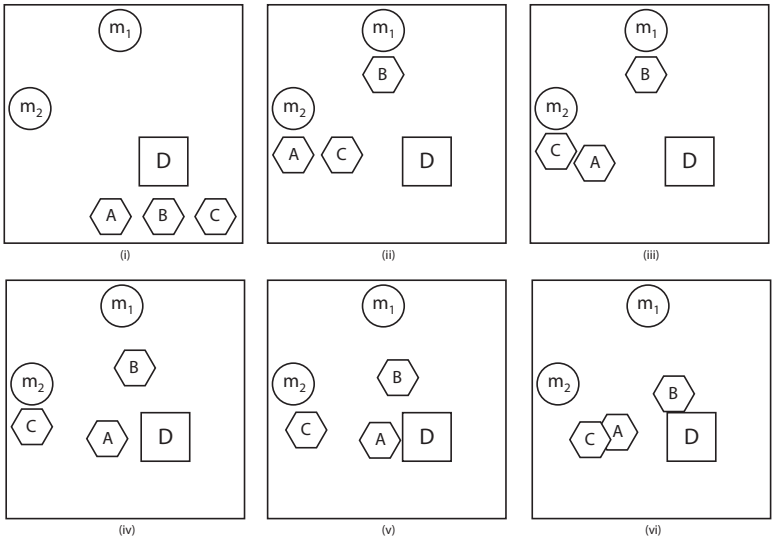


Figure 4.1: (i) None of the agents have moved. A is assigned to m_2, B to m_1, C to m_2. (ii) A is done using m_2, B is gathering m_1, C starts gathering m_2. (iii) B is done using m_1, A is moving to D, C is still gathering m_2. (iv) C is done using m_2, A is still moving to D, B is moving to D. (v) A delivers to D and is assigned to m_2, B is still moving to D, C is moving to D. (vi) B delivers to D and is assigned to m_1, A is moving to m_2, C is still moving to D.

agents ahead in the queue (lines 17-18) in order to calculate the time spent waiting in line.

Since the time spent on traveling is included in the return value in line 25, the travel time has to be subtracted from the waiting time in lines 20-24. $tt_{D,j}$ is the travel time from the resource deposit to the resource site to which the argument queue, Q belongs.

Consider the scenario in Figure 4.1 in which three agents, A, B and C must gather resources from m_1 and m_2 by transporting resources from these positions to the resource deposit D. The actions of the agents resulting from applying Algorithm 1 to this scenario can be observed in Table 4.1. Each queue is initially empty and each agent is initially not assigned to a resource site.

The first three rows in the trace display each agent being assigned to their first resource site. The algorithm takes the distance to be travelled into account for each agent as well as the remaining time for agents that have already been assigned to a resource site. When agents return to D, Algorithm 1 is run again, to assign the agents to their next resource site. Over the course of a session, each agent may be assigned to different resource sites, depending on their availability.

4.3 TRAVEL FUNCTION

As mentioned in Chapter 2, the test bed of the algorithm is StarCraft: Brood War. In this game, agents do not move at a constant speed.

Table 4.1: A sample trace for the scenario in Figure 4.1. From the left is the current time, the action of the affected agent, the current state of both resource site queues and the amount of gathered resources.

Time	**Action**	**Queues**	**Resources**
0	A move to m_2	$Q_{m_1} = \emptyset$ $Q_{m_2} = \{A\}$	0
0	B move to m_1	$Q_{m_1} = \{B\}$ $Q_{m_2} = \{A\}$	0
0	C move to m_2	$Q_{m_1} = \{B\}$ $Q_{m_2} = \{A, C\}$	0
190	A move from m_2	$Q_{m_1} = \{B\}$ $Q_{m_2} = \{C\}$	0
206	B move from m_1	$Q_{m_1} = \emptyset$ $Q_{m_2} = \{C\}$	0
270	C move from m_2	$Q_{m_1} = \emptyset$ $Q_{m_2} = \emptyset$	0
280	A deliver & move to m_2	$Q_{m_1} = \emptyset$ $Q_{m_2} = \{A\}$	8
314	B deliver & move to m_1	$Q_{m_1} = \{B\}$ $Q_{m_2} = \{A\}$	16

An agent, given a movement order while standing still, will first accelerate towards a maximum speed and decelerate before reaching its destination.

Since StarCraft: Brood War, is closed source, access to the algorithms used for path finding and acceleration in the game is restricted. To compensate for this, travel times are measured and stored. Every time an agent moves between the resource deposit and a resource site, the time of the travel is recorded. Information on the travel is saved in a lookup table, including the starting position, the destination and the time spent travelling. As the function will only be used to decide the travel time between a resource deposit and the stationary resource site, the amount of starting position/destination pairs is limited, making this approach possible.

Whenever another agent is moving in a path equal to a recorded path, the original estimate of the time needed for travelling is used and potentially revised. Given enough data, all possible paths an agent may use when mining resources is known for a specific map.

The information is gathered offline by using the resource gathering algorithm, where the travel function is defined as a function that returns a previously calculated value. If no data matching the source and destination exists, the function returns a low value to allow the agent to take the path and thereby gather previously unknown data.

It should be noted that this approach necessitates that the environment is either static or that changes does not influence the travel times. This is not the case in StarCraft: Brood War, as new travelling paths may become available when resource sites are depleted. Depleting a resource site causes the area previously blocked by the resource site to become passable. This does not impact the experiments performed for this thesis as the experiments do not run long enough for any resource site to become depleted. In general, due to the nature of resource gathering in StarCraft: Brood War, the resource sites will often deplete at approximately the same time, making the influence of the variable environment negligible.

5 COMPLEXITY ANALYSIS

The time complexity of Algorithm 1, given k resource sites and n workers is as follows: At a given run of the algorithm, the loop starting in line 2 is run a maximum of n times (when no workers have been assigned to a queue). In each of these loops, the loop starting in line 3 is run k times. Finally, in each of these inner loops, at most 2 calls are made to procedure `Work(Q,a)`, as seen in lines 5-6. This makes for a worst case scenario of $n \cdot k \cdot 2 \cdot f(n)$, where $f(n)$ is the time complexity of procedure `Work(Q,a)`, given the maximal possible size of a queue, n.

Other than the loop starting in line 17 of procedure `Work(Q,a)`, the execution time of the lines in procedure `Work(Q,a)` can be bound by a constant factor, since the structure of the rest of the algorithm consist solely of if/else statements. The complex part of procedure `Work(Q,a)` is the recursive call in line 18. To keep the algorithm simple, no unnecessary optimisations have been made, making the running time

$$f(n) = \begin{cases} 0 & \text{for } n = 1 \\ 2^{n-1} - 1 & \text{for } n > 1 \end{cases}.$$

By using simple optimisations, keeping a local variable for the result of the call to procedure `Work(Q,a)` in lines 5-6 of Algorithm 1 and using dynamic programming for the result of the recursive calls in procedure `Work(Q,a)`, the complexity can be brought down to $n \cdot k \cdot n = n^2 \cdot k \Rightarrow O(n^2)$.

6 EXPERIMENTAL RESULTS

The algorithm was implemented and the performance of it compared with the built-in resource gathering method in StarCraft: Brood War. Both methods were used, whilst creating 18 agents on the map *Astral Balance*. Each time an agent was created–or had performed a gathering task–it was assigned to a resource site queue using the algorithm. For the purpose of the test, the tenth agent was ordered, upon creation, to construct a supply depot next to the resource depot to allow for the construction of an additional 8 agents. In order to ensure fairness, all tests were performed in the upper right corner, even though the difference of starting positions for players is usually considered negligible. The starting position is next to a mineral cluster consisting of eight mineral fields, each holding 1,500 minerals.

The performances of both methods are measured by evaluating two properties: the average cumulative amount of resources collected by agents over the course of 8,000 frames, as well as a standard deviation of gathered resources during this time frame.

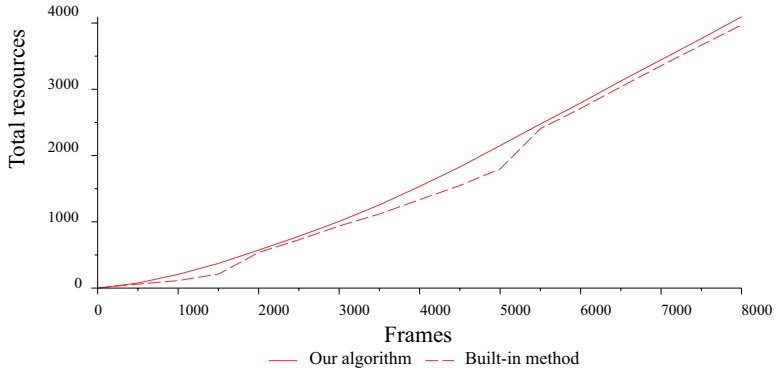


Figure 6.1: Comparison of the two methods, showing the amount of minerals gathered over time.

Figure 6.1 shows a comparison between the two methods with regards to the total amount of collected minerals during the course of 8,000 frames.

The graph shows the average amount of minerals collected at different time intervals during experimental runs of StarCraft: Brood War. The graph is based on data from 10 runs. Clearly, the proposed algorithm increases the income of the player compared to the built-in method. Furthermore, the trend of the curve for the proposed algorithm is easier to recognise than the curve of the built-in method,

indicating a higher predictability. The predictability is further elaborated in Figure 6.2, depicting the standard deviation in the course of the 10 runs. The figure shows that the standard deviation of the minerals collected by both methods grows as time passes. There is a clear tendency that the deviation of the built-in method grows faster than that of the proposed algorithm.

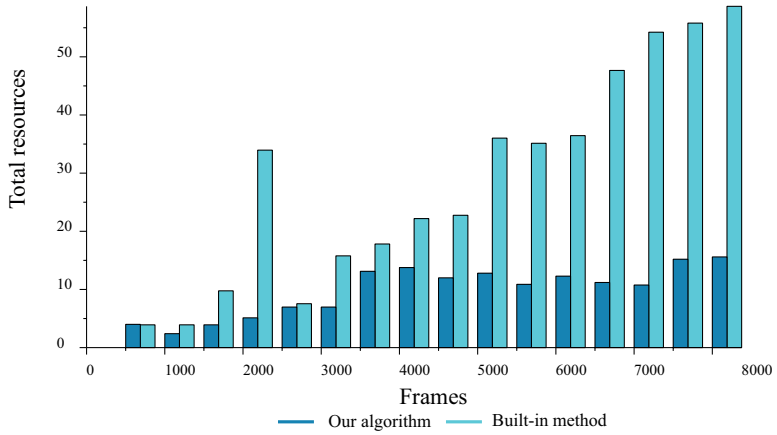


Figure 6.2: Comparison of the standard deviation of the two methods.

7 DISCUSSION AND FUTURE WORK

Experiments show that Algorithm 1 provides an increase in the amount of resources gathered compared to the built-in method of StarCraft. Furthermore the experiments show that the algorithm is more predictable. The predictability can be used to predict the amount of resources collected within a specific time frame.

The increased predictability of income using Algorithm 1 makes it eligible for use in other aspects of an RTS game. An example is to develop a scheduler for construction of, for example, military units. By using the algorithm, it is possible to predict the time at which point the scheduled production line is finished. The algorithm enables the possibility of acquiring an accurate estimate of the gain from utilising resource sites from other areas in the game world. This is interesting for RTS games in which new resource deposits may be created by a player.

The algorithm is general in the sense that it can be applied for RTS games having the following rules.

- Several resource sites are present
- A resource site contains several units of the resource
- Agents move between a resource site and a resource deposit
- Only one agent may use a resource site at a time

The algorithm is *greedy* in the sense that each agent makes decisions that allow for the fastest mineral gain for itself, but not necessarily the best for all agents in a team. It might be beneficial to run the algorithm over a group of agents by having them act as a team and optimising the group income instead of the individual income over time.

Notice that the algorithm depends on the implementation for computing travel times, potentially necessitating a consideration of agent collisions. In some RTS games, the travel time is less significant, due to the distance from a resource site to a resource deposit, the number of agents or other game specifics. In RTS games similar to StarCraft: Brood War we expect to observe similar performance of the proposed algorithm.

The simple approach to calculating travel times, presented in Section 4.3, only works in a static environment which, in our case, is ensured by not running the game long enough for any resource site to deplete. In essence, this keeps the environment static. For a travel time function to work in a dynamic environment, an approach complying with environmental change is needed.

The travel time between positions must either be known or it must be possible to estimate this value. A general algorithm for calculating travel time has not been presented, as the value is very dependent on the environment. If the value of the travel time is not exact, the algorithm does not guarantee the lowest round-trip time for an agent.

Part II

DATA-DRIVEN EXPANSION STRATEGIES

8 INTRODUCTION

As mentioned in Chapter 2, the scope of this thesis is resource management using a data-driven approach. This part of the thesis focuses on base expansion. Chapter 2 states that it is important to expand in order to facilitate the exploitation of resources other than those in the immediate area.

A decision model can be used to model the task of choosing a satisfactory location for base expansion. A variety of decision models exist that may be used to model this problem. The overall theme of this thesis is data-driven resource management, and the chosen decision model should conform to this. The goal is to mimic the actions of a human player choosing a location for base expansion.

A set of input features is needed in order to construct a decision model that is able to determine sound locations for base expansion. These features can be determined through expert knowledge. Though a large number of features may be found, the significance of these may not be easy to identify. Features of low significance may introduce noise to the model. The removal of these features is likely to make the decision model less prone to approximation errors.

There exist a number of methods, which can be used to remove features of low significance, called *feature selection* methods. In this case, the features that should be removed are those which do not contribute to the estimation of relevant expansion locations. The reduction of features provides a less complex structure of a decision model. In the case where the values of some features are estimates, the removal of these is likely to result in a more deterministic decision model output.

Chapter 9 contains the theory of two decision models applicable for the task of choosing an expansion location. Chapter 10 deals with selecting the initial feature set, using expert knowledge. Chapter 11 describes the theory of three different feature selection methods for choosing a set of the most significant features. Chapter 12 deals with the implementation of the three feature selection methods and the initial performance of the attained feature sets. In Chapter 13 the attained feature sets are evaluated. Chapter 14 contains a discussion on the work done in this part of the thesis along with suggestions for future work.

9 THEORY OF DECISION MODELS

This chapter covers the theory of two common decision methods, utilised to solve the problem of choosing a satisfactory location for base expansion. The chapter explains the structure of both methods, how they are trained and how they are used to make decisions. The two decision methods are compared using test scenarios in Chapter 13.

9.1 DECISION TREES

Decision trees represent a function, mapping a vector of attribute values to a single output value [13]. The non-leaf nodes represent the attribute values, while the edges from each of these nodes are labelled with the possible values of the particular attribute. Figure 9.1 shows a decision tree with three attributes; *Resources*, *Food* and *Army size*. The tree is a model, classifying the outcome of an RTS game, given the state of the aforementioned attributes. A path from a root node to a leaf node represents a particular game state, while the leaf node represents the outcome.

Starting with the root node, a decision tree is used by evaluating the variable of a node and following the corresponding label to the next node. This is performed until a leaf node is reached. The reached leaf node is then the output of the decision tree. As an example, given 400 resources, 10 food and an Army size of 12, by following the corresponding edges of the tree (left, left, right), the leaf node *Win* is reached, indicating a victory. This relation is, evidently, a crude simplification of reality and serves only as an example.

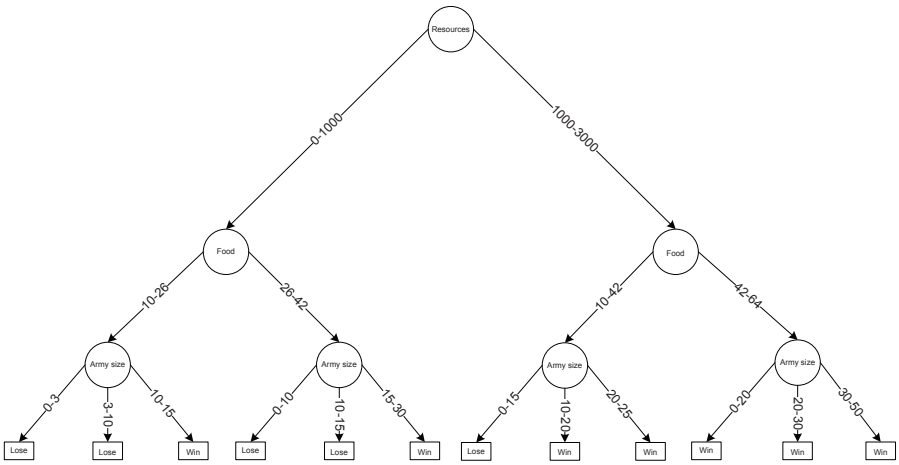


Figure 9.1: A simple decision tree showing a connection between resources, food, army size and victory.

Decision trees are easy to read. The simplistic nature of the trees makes them fast and efficient to evaluate. The problem of learning an optimal decision tree is, however, known to be NP-complete [7], necessitating training solutions that approximate the optimal decision tree for large problems.

9.1.1 *Decision Tree Learning*

This section describes decision tree learning and is based on Russell and Norvig [13, Pages 697-704]. The complete set of all possible decision trees made from a set of attributes is very large, making it infeasible to iterate all combinations to find the tree that most closely resembles a given function. Since it is not always possible to simply construct the decision tree with expert knowledge, a learning algorithm is necessary.

A tuple (X, y), where X is a vector of attribute values and y is the target output, is called an example. Using the decision tree learning algorithm described in Russell and Norvig [13, page 702] it is possible to create a decision tree using a set of examples. The learning function can be seen in Algorithm 2.

The general idea of the learning function is to generate the tree through recursive depth-first traversal, starting with the root node. The set $examples$ is initially the full set of examples and is reduced at each level when split by an attribute test until a stopping criterion is reached. $attr$ is the set of attributes that have not yet been used for a test and is initially the full set of attributes. Each call to the learning function generates a single node where $parentExamples$ contains the examples used by the parent node. The first call to the learning function generates the root node, making $parentExamples$ initially empty.

Lines 2-9 in Algorithm 2 contains the stopping criteria of the learning algorithm:

1. If $examples$ has become empty, there are no examples for this path in the tree. In this case, the target output with the highest occurrence in $parentExamples$ is returned, as that is considered most likely to be the correct output.

2. If the elements of $examples$ all have the same target output y, then the output for every sub-path of this path yields the same output. When this is the case, there is no point in creating a sub tree. Instead a leaf node is generated by returning the target value of $examples$ (which are all the same).

3. If $attributes$ is empty, there are no attributes left to use for constructing further nodes. In this case, it is not possible to create a sub-tree even though the members of $examples$ may

Algorithm 2 The decision-tree learning algorithm. Figure 18.5 in Russell and Norvig [13, page 702]

Parameters
examples : Remaining examples.
attr : Set of attributes not yet tested.
parentExamples : Set of examples used by the current parent node.
Returns
A decision tree.
Declarations
PLURALITYVALUE(*examples*) : Accepts a set of examples and returns the target output y with the highest occurrence.
IMPORTANCE(a, *examples*) : Accepts an attribute a and a set of *examples*. Returns a value indicating the gain from using a for splitting *examples*.

1: **function** TREELEARN(*examples*, *attr*, *parentExamples*)
2: **if** *examples* is empty **then**
3: **return** PLURALITYVALUE(*parentExamples*)
4: **end if**
5: **if** all *examples* have the same target output y **then**
6: **return** y
7: **end if**
8: **if** *attr* is empty **then**
9: **return** PLURALITYVALUE(*examples*)
10: **end if**
11: $A \leftarrow \arg\max_{a \in attr}$ IMPORTANCE(a, *examples*)
12: *tree* $\leftarrow$ a new decision tree with root test A
13: **for all** states v_k of A **do**
14: $exs \leftarrow \{e : e \in examples$ **and** $e.A = v_k\}$
15: *subtree* $\leftarrow$ TREELEARN(*exs*, *attr*$\backslash A$, *examples*)
16: add a branch to *tree* with label $(A = v_k)$ and *subtree*
17: **end for**
18: **return** *tree*
19: **end function**

not share the same target output. The target output with the highest occurrence in *examples* is returned as this is correct in the majority of observed cases.

Lines 11-12 of Algorithm 2 finds the best attribute A for the current *examples*, through the `Importance`-function discussed in Section 9.1.1.1. A *tree* is then generated with A as root. Lines 13-17 iterates through all possible states of A, and for each state adds a branch to *tree* by calling `TreeLearn` recursively, using the examples belonging to the state, the remaining attributes and *examples* as the parent examples.

9.1.1.1 *Importance Function*

The importance function is required for the decision-tree learning algorithm to determine which attribute should be used at a node to split the example set. The desired attribute is the attribute that will provide the largest information gain or predictive power. An often used measure for unpredictability is called entropy. For the sake of simplicity, assume that all target outputs are binary.

Entropy B of a binary random variable with q probability of being true, is calculated as:

$$B(q) = -(q \cdot \log_2(q) + (1-q)\log_2(1-q)) \ .$$

meaning high entropy for probability 0.5 and low entropy for probability values 0 and 1. An attribute, A, may be evaluated by observing the amount of entropy left after testing the attribute. The remaining entropy is:

$$\mathrm{Remainder}(A) = \sum_{k=1}^{d} \left(\frac{p_k + n_k}{p+n} B\left(\frac{p_k}{p_k + n_k} \right) \right) \ .$$

where d is the number of states of A, p_k is the number of positive examples for state k and n_k is the number of negative examples for state k. The difference between the entropy before testing attribute A and after, is the reduction in entropy or information gain:

$$\mathrm{Gain}(A) = B\left(\frac{p}{p+n} \right) - \mathrm{Remainder}(A) \ .$$

Information gain is a valid choice for the `Importance`-function in Algorithm 2, as the goal is to reduce entropy by the use of attribute tests.

9.2 ARTIFICIAL NEURAL NETWORKS

ANNs are inspired by the biological networks of neurons found in the brain. ANNs may be used for solving problems without the need for constructing an elaborate model of the environment. An ANN consists of a layer of input nodes connected to layers of connected artificial neurons where each connection has an associated weight, which is updated through training. This section will refer to both artificial neurons and input nodes as nodes. ANNs are used by providing input values for the input nodes, propagating this information throughout the network and reading the output.

9.2.1 *Artificial Neuron*

An artificial neuron is a conceptual part of a neural network which encapsulates a function. An ANN contains at least one layer of one or more artificial neurons where the outermost layer is referred to as the output layer. Layers between the layer of input nodes and the output layer are called hidden layers. Connections between layers depend on the network type but for a feedforward network each layer is totally connected with the preceding layer.

The input of an artificial neuron is the output of each node having a directed connection to the particular artificial neuron. A common propagation function for artificial neurons is the weighted sum of the inputs.

$$in_j = \sum_{i \in I} w_{i,j} \cdot out_i \, . \tag{9.1}$$

where in_j is the network input for the artificial neuron j, in this case calculated by the weighted sum. I is the set of all nodes in a network and $w_{i,j}$ is the weight from an input node or artificial neuron, i, to the artificial neuron denoted as j. Weights between nodes that are not connected may be considered zero or non-existent. out_i is the output of the node i, meaning it is the calculated output when i is an artificial neuron and merely the input value when i is an input node.

9.2.2 *Activation Function*

The output of an artificial neuron is rarely just the result of the propagation function, as a network modelled this way is not more expressive than a network without hidden layers. This is because a propagation function is a linear function and the application of a linear function on another linear function is also a linear function. Using a non-linear activation function for determining the output of an artificial neuron makes non-linearity possible, resulting in a more expressive network.

The output of an artificial neuron using an activation function, g, becomes

$$out_j = g(in_j) \, .$$

where in_j is the artificial neuron input described in Equation 9.1.

Equation 9.2 is an example of an activation function; this particular function is a step function. A step function returns one specific value when the input is less than the threshold of the function and another value when the input is higher than the threshold. In Equation 9.2 the threshold value is one half and the possible outputs are one and zero.

$$\sigma(t) = \begin{cases} 0 & \text{for } t < \frac{1}{2} \\ 1 & \text{for } t \geqslant \frac{1}{2} \end{cases} \, . \tag{9.2}$$

In the case of a binary activation function, an artificial neuron is sometimes said to *fire* when the output is one and to not fire when the output is zero. Using a binary activation function causes the changes in weights and input to reflect rather coarsely, as a small change may push the activation function beyond the threshold, causing a large change in the output.

The activation function does not have to be binary, as another function returning a value that scales non-linearly with the input is also acceptable. Popular activation functions include the Sigmoid function:

$$P(t) = \frac{1}{1 + e^{-t}} \, .$$

The Sigmoid function produces a value between zero and one, based on the input. The property of the Sigmoid function compared to a binary function is that a small change in the weight or input has only a small effect on the output.

The suitability of an activation function depends on the problem being solved. A binary activation function may be preferable when dealing with a problem area with hard limits, such as learning the binary *AND function*. Binary logic functions have well defined outputs with hard limits for each set of input as the result is always either true or false.

Figure 9.2 shows an ANN consisting of two input nodes as well as a single artificial neuron which have learned the binary *AND* function. The activation function being used is the step function from Equation 9.2. The inputs of the network is a vector of two numbers that is either one (true) or zero (false). The weights adjust the value from the inputs such that the threshold is reached when and only when both inputs have the value one. A network with all the inputs connected directly to the outputs is called a single-layer network or a perceptron network [13].

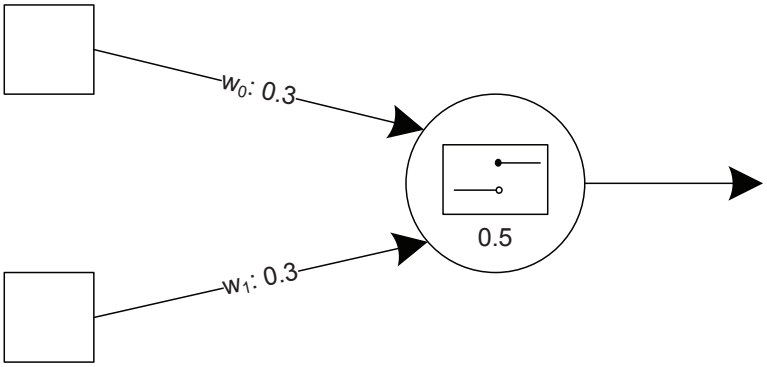


Figure 9.2: A neural network modeling the binary *AND function*.

9.2.3 *Learning*

A strong feature of ANNs is their ability to approximate a function by learning. This is usually done by changing the weights of the edges in the network, influencing the output of the network. There are several methods for learning, though this section will only describe supervised learning. In supervised learning, the network is instructed through the use of input/output vectors, where the outputs are the desired responses to the particular inputs. A pair of two vectors x, y where y is the target response of input x will be referred to as a *sample*.

A popular method for supervised learning is the backpropagation method. With this method, the difference/error of the expected output and the actual output propagates backwards through the network. In order to apply backpropagation, the outputs of all neurons are first calculated, given x as the network input. The error at each artificial neuron, j, in the output layer is calculated as

$$\Delta_j = g'(in_j)(y_j - out_j) \ .$$

where Δ_j is the error at neuron j, g' is the derivative of the activation function g, and y_j is the j^{th} component of the output vector, meaning the target output of artificial neuron j. $(y_j - outj)$ is the difference between the current output of artificial neuron j and the target output. The general idea is that by multiplying this value with the derivative of the activation function applied to the artificial neuron input, the error becomes dependent on the amount of impact a change in the artificial neuron input will have.

The error for each artificial neuron, i not in the output layer, is calculated as

$$\Delta_i = g'(in_i) \sum_j w_{i,j} \cdot \Delta_j \ .$$

so that the error of an artificial neuron depends on the error of every other artificial neuron that is influenced by this artificial neuron. The

weight, $w_{i,j}$ is included as a factor in this, such that the amount of error propagating back depends on the amount of error, the artificial neuron is responsible for.

When the Δ-values for all artificial neurons in the network have been calculated, each weight is updated as seen in Equation 9.3 where $w'_{i,j}$ is the new value of $w_{i,j}$.

$$w'_{i,j} = w_{i,j} + \alpha \cdot out_i \cdot \Delta_j \ . \tag{9.3}$$

α is the learning rate, which is usually a value between zero and one. The learning rate determines how much will be learned from updating the weights using this sample. Note that a single iteration of weight updates, using a sample x, y, is not guaranteed to adjust the network in a way, where the correct output will be achieved for the input vector x. The purpose of the update is to have the network approach the correct function. This is achieved by several iterations of learning as seen in the simplified algorithm, Algorithm 3.

Algorithm 3 Learning with backpropagation

```
1: while Stopping criterion is not reached do
2:     for all Input/output pair (x, y) do
3:         Calculate outputvector given x as input
4:         Calculate Δ-values by backpropagation
5:         for all Weight w do
6:             Update w using Δ
7:         end for
8:     end for
9: end while
```

As it is visible in the algorithm, several examples are used for instructing the network. The criterion used as predicate for the while-loop depends on choice. It may be as simple as a fixed run of $10,000$ iterations or it may be that the network must provide correct output for some percentage of test-data. There is no criterion that can be said to be the best as it will often depend on the network and the size of the problem.

9.3 SUMMARY

Decision methods have been successfully applied for selection of strategies [14, 12]. Randall et al. [12] utilises ANNs in the ship battle RTS game *DEFSIM*. Each ship is equipped with an ANN and tasked with making cooperative strategic decisions, including choosing destinations and targets to attack. To achieve victory, it is necessary for friendly ships to move in groups to maximise their collective firepower, requiring the ships to predict where other friendly ships will move.

As stated in Section 2, the scope of this thesis is data driven resource management. The use of replays should facilitate sufficient data for training decision models and enable the models to mimic the behaviour of human players.

10 DATA EXTRACTION

In order to create a decision model for choosing expansion locations, it is necessary to determine a set of features that potentially influences what location a player chooses for expansion. The idea formed in this chapter uses a set number of features determined by expert knowledge. The features are to be used for a data-driven approach, based on data extracted from several StarCraft replays. Chapter 11 focuses on selecting relevant feature subsets based on the data extracted in this chapter, while Chapter 12 focuses on the finalised decision model, based on the extracted and refined feature subsets. Some of the features proposed in this chapter are based on previous work by the authors [6].

10.1 FEATURE PROPOSITION

It may be argued that several factors are involved in choosing expansion locations. Such factors may include the amount of resources a location has to offer, as well as the possibility of meeting an opponent while traveling to the location. Some factors might not be obvious at first, since it is hard to include the human theory of minds, i.e. that players starts to think about how the opponent thinks, and the recursion that may follow.

The goal of this chapter is to propose a list of features a player may consider, when picking a location for expanding. These features are to be used to accurately predict how desirable an expansion location is. In turn, the model might be used to predict where an enemy is expanding. The features presented in this section are based on expert knowledge by the authors, and some features may be irrelevant. The goal has been to make a list that is as exhaustive as possible, and then select a subset of these features using feature selection (see Chapter 11).

For a summarised version of the features presented here, as well as their domain, see Appendix A.

Features relative to the location itself

AMOUNT OF GAS The amount of gas available on site.

AMOUNT OF MINERALS The amount of minerals available on site.

NUMBER OF CHOKEPOINTS The number of chokepoints in the area. A chokepoint is a connection between two regions. A region is a partition of the map containing no static obstacles, such as rivers,

hills and cliffs. In general, the fewer chokepoints in an area, the less entrance points exist for enemy ground units.

PRESENCE IN SHORTEST PATH BETWEEN ENEMY/OWN BASE A key point to surviving and making surprise attacks on the enemy is to remain hidden for as long as possible. If an expansion is placed in an area that needs to be passed through on the way between two opposing base locations, there is a chance that the expansion will be discovered. On the other hand, the area may also be a key strategic position to put up main defences, given that it is the only route between two opposing base locations.

DISTANCES FROM BASES TO THE SITE Various distances from both own and enemy bases to the site. The lower the distance from the nearest enemy base, the faster the opponent can bring reinforcements to invading forces. In the same way, the lower the distance from the nearest ally base, the faster reinforcements can be brought to the defenders. The distances are split up in four categories:

OWN/ENEMY FLYING DISTANCE The flying distance from the nearest enemy/ally base to the site.

OWN/ENEMY GROUND DISTANCE The ground distance from the nearest enemy/ally base to the site, taking into consideration the environmental obstacles that may be on the way.

OPEN SIDES IN THE AREA In StarCraft, every map has a fixed size on both axes, with an impenetrable wall that no units may pass. In effect, an area close to one or more of these axes has fewer sides to defend from enemy forces.

MAP POSITION A map position is 1 out of 9 positions. The rectangular map is divided into 9 equally sized cells, being either north-west, north, north-east, west, middle, east, south-west, south or south-east.

Features relative to the map and general status of the map

NUMBER OF OWN/ENEMY EXPANSIONS The number of enemy expansions and the number of own expansions may reveal something about the status of the match. Roughly speaking, the more expansions a player has, the higher their income and hence the better their position in battle.

NUMBER OF POSSIBLE EXPANSIONS Every map in StarCraft is different. The amount of sites to set up a base for resource gathering varies from one map to another. This number may directly affect the importance of other factors like the number of own/enemy

expansions, since two expansions on a map with 16 expansion sites might be less important than the same on a map with four expansion sites.

HIGHEST PRESENCE IN SHORTEST PATH In order to determine the significance of the presence in the shortest path between enemy and own base, a normalisation factor is needed. The choice for this feature is the *highest* presence possible between the choices of expansions.

GAME TIME IN FRAMES The amount of time that has passed since a match began may be important for several reasons. For example, since defensive structures require time and resources to build and maintain, an early expansion may increase the income of a player, but will leave both the initial base and the expansion vulnerable until defences have been set up.

BEST NUMBER OF GAS/MINERALS The highest number of gas/minerals possible for all choices of expansions. Both may be used as normalisation factors for other features.

OWN/ENEMY RACE Each race in StarCraft has its own strengths and weaknesses, so the presence of different enemy and ally races may prove an important factor when deciding where to expand.

MAP SIZE The dimensions of the map may affect the pace of the game; a small map is likely to make early attacks viable, while a large map may encourage the construction of large armies.

OWN/ENEMY AIR UNITS The number of own/enemy air units may affect the accessibility of areas, and in turn influence the desirability of these areas.

Features based on various scores (own/enemy)

StarCraft has various statistics for each player in the form of scores, indicating how well they are doing. The preferred expansion sites of a player may depend on whether they have the upper hand in a match or are fighting for survival.

UNIT SCORE Each unit is given a score in StarCraft. The score depends on their price and their place in the tech tree. The unit score is the accumulated score of all previously constructed units.

BUILDING SCORE Each building is given a score in StarCraft. The score depends on their price and their place in the tech tree. The building score is the accumulated score of all previously constructed units.

RAZING SCORE When a player eliminates an enemy building, a score is gained. The razing score is the accumulated score for all buildings eliminated by a player.

10.2 DATA EXTRACTION APPROACH

This section provides a description of the approach used for data extraction. The replay format of StarCraft and the limitations created by this format is described in Section 10.2.1. The framework utilised for extracting data–as well as the approach–is described in Section 10.2.2.

10.2.1 *Replay Format*

The replay format of StarCraft contains all static information about the map, along with the names and races of the players in the match. The representation of the events of a match is an ordered set of actions performed by each player, along with the time of each action relative to the start of the match. Anything that happens during the match that is not a player-made action is not contained within the replay format.

Each action can be thought of as a representation of a decision made by a player. As an example, imagine a player deciding to construct a supply depot. The player selects a worker unit, chooses a building to construct, as well as a location. When this has been done, the worker unit will move to the location and attempt to construct the building. The player actions of this example are the selection of the worker unit, and the command to construct a supply depot at a particular destination.

Depending on the state of the game at the time of this action sequence, as well as other actions being performed, the construction of the supply depot may either fail or succeed. If part of the build site is blocked when the worker arrives at the destination, the build will fail–as will the build if the worker is destroyed by an enemy, or if the worker is given a new command. The success or failure of an action is not in itself an action performed by a player, meaning that this information is not included in the replay data. Other commands are no different, in that no information is provided indicating whether the command has succeeded or failed.

When a replay is executed through StarCraft, the initial state of the game is known. Using the initial state, the player events and the game engine, StarCraft is able to simulate a match completely. The way StarCraft simulates replays does not facilitate skipping to a specific time in the match. Every action up until a specific point in time must be executed one frame at a time.

As the raw replay data lacks the information of certain interesting events–such as the destruction of units–it is infeasible to use the raw replay files for extracting data, as it would require complete knowledge

of the closed StarCraft game engine. An alternative is to use the game engine of StarCraft to run the replays and extract data in real time, which is the approach used in this thesis.

10.2.2 *Broodwar API*

BWAPI[1] is an open source framework for creating AI modules for StarCraft: Brood War. Through the use of DLL injection, this framework makes it possible to subscribe to in-game events. Events are fired in both live games and replays, and it is possible to access static map data as well as iterate through units to gain additional information.

The implementation of the data extraction method involves saving a vector of feature data, for every possible build site, every time an expansion is being constructed by any player within the game. Each vector contains information specific to the build site as well as the overall state of the game and most important, whether the build site was chosen for expansion or not. This is indicated by the vector containing the value BUILD when the vector represents the build site that was chosen for this game state and NOTBUILD when the build site was not chosen for this game state.

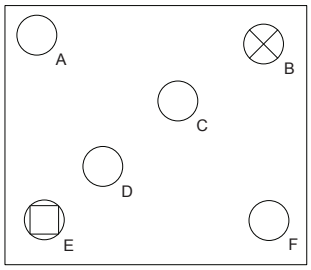


Figure 10.1: An example containing expansion sites.

The points A through F in Figure 10.1 represents positions that are lucrative for expanding. The cross and the square represents existing bases for player one and player two respectively. At the beginning of the match, both players are assigned one base each. First player one is considered; a vector of feature data is constructed for the build site, E as this is the position of player one's base. This vector contains the value BUILD. For each potential build site a vector is generated, each containing the value NOTBUILD. In this example a vector is generated for each of the following expansion sites: A, C, D, F. The vector for expansion site E has already been made, but note that no vector is made for expansion site B. The reason for this is that B is not considered a valid expansion site, because there already is a base on that position. Any occupied expansion site is ignored, regardless of whether the owner is the current player or an enemy.

1 http://code.google.com/p/bwapi/

The result of this approach is a set of feature vectors, where one of the features reveals whether the game state represented by the other features resulted in placing an expansion at the particular expansion site or not.

11 FEATURE SELECTION THEORY

This chapter introduces three methods for selecting the most influential features observed in a domain; Sequential Forward Selection (SFS), Sequential Backward Selection (SBS) and Sequential Forward Floating Selection (SFFS). Feature selection methods are useful when constructing decision models, as the implicit simplification of a model may increase the overall performance. If learning needs to be applied to a model, reducing the number of input features is likely to result in faster training time. The feature selection methods discussed in this section are sequential, as they iteratively add or remove features in a feature set. Sequential feature selection methods are the most commonly used methods for feature selection [8].

The methods always yield the same solution given the same input and a non-stochastic performance measure of feature sets. The forward method starts with an initially empty feature set, and adds the most influential features, while the backward method starts with the full set, and removes the least influential feature. SFFS is a further development of the SFS method that potentially explores more permutations of features, by allowing features to be removed from the set after they have been added.

All three methods require a performance function that accepts a set of features and evaluates the performance of this set. The choice of performance function depends on the domain. The methods do not examine all possible subsets of the features, nor do they guarantee the optimal solution. An exhaustive search of all subsets requires a search through 2^k permutations, where k is the total number of features.

Feature selection methods have previously been applied in the game domain. Yannakakis [16] uses SFS and Perceptron Feature Selection (PFS) to select a minimal subset of the most influential features from a set of 71 features, in order to make an accurate model of user preferences in a game. Yannakakis [16] reiterates that SFS is commonly used, since it yields high performance with minimal feature subsets in a wide range of feature selection problems [17, 18]. The PFS method is similar to SBS with a slight change. Instead of looking at the performance of the feature set, a perceptron is used, with an input for each feature. The perceptron is then trained, and the features with a weight below some threshold are removed. This continues until no weights are below the threshold. Yannakakis [16] shows that SFS yields a higher performance than PFS.

11.1 SEQUENTIAL FORWARD SELECTION

This section explains how the SFS method is used to perform feature selection. The method works by starting with the empty set, and adding the most influential feature one at a time, until some stopping criterion is satisfied.

Let X be an initially empty set of features, and let x_i denote the i^{th} feature in X. Let D be the set of features not added to X, and let d_i denote the i^{th} feature in D. Let the performance function $J(X)$ accept a set of features and return a value indicating the performance of the set. Selection of the most influential features, using SFS is described in Algorithm 4

Algorithm 4 The Sequential Forward Selection Algorithm

while Not Stopping Criterion **do**
  $x_{best} \leftarrow \{d_0\}$
  for all $d_i \in D \backslash \{d_0\}$ **do**
    if $J(X \cup \{d_i\}) \geqslant J(X \cup x_{best})$ **then**
      $x_{best} \leftarrow \{d_i\}$
    end if
  end for
  $X \leftarrow X \cup x_{best}$
  $D \leftarrow D \backslash x_{best}$
end while

Algorithm 4 describes how the method runs until some stopping criterion is met, and iteratively adds the most influential feature. The element d_i in D with the highest performance when combined with X is added to X and removed from D. Examples of stopping criteria include a decrease in performance or simply a maximum number of selected features.

11.2 SEQUENTIAL BACKWARD SELECTION

This section explains the SBS method which, given a feature set, iteratively removes a feature until some stopping criterion is met.

Let X be an initially full set of features, and let x_i denote the i^{th} feature in X. Let the performance function $J(X)$ accept a set of features and return a value indicating the performance of the set. Selection of the most influential features using SBS is described in Algorithm 5.

Algorithm 5 describes how the SBS method runs until some stopping criterion is met, by iteratively removing the worst feature from the set. The worst feature is the feature that, when removed from the set, provides the best performance of the new set. The feature is identified and removed, and the remaining set is evaluated.

Algorithm 5 The Sequential Backward Selection Algorithm

while Not Stopping Criterion **do**
  $x_{worst} \leftarrow \{x_0\}$
  for all $x_i \in X \backslash \{x_0\}$ **do**
    if $J(X \backslash \{x_i\}) \geqslant J(X \backslash x_{worst})$ **then**
      $x_{worst} \leftarrow \{x_i\}$
    end if
  end for
  $X \leftarrow X \backslash x_{worst}$
end while

11.3 SEQUENTIAL FORWARD FLOATING SELECTION

This section explains the SFFS method proposed by Pudil et al. [10]. The SFFS method is an extension of the SFS method. SFFS allows already included features to be excluded, if they are later deemed less important given a new context. The method floats between subsets of the full feature set by adding and removing features. This allows for measuring the performance of a potentially greater amount of subsets than the SFS method.

Let X be an initially empty set of features, and let x_i denote the i^{th} feature in X. Let D be the features not added to X, and let d_i denote the i^{th} feature in D. Let the performance function $J(X)$ accept a set of features and return the performance of the feature set. The SFFS method is expressed as pseudo-code in Algorithm 6

In lines 2-9 of Algorithm 6, the method locates the feature belonging to D that will yield the highest performance when added to X. When the feature has been found, the two sets are updated accordingly. In lines 10-18, the method locates the least influential feature in X, and if this is not the feature just added, removes the feature by updating the sets accordingly. If a feature was removed and X contains more than two features after this operation, the method will call the procedure, `FurtherExclusion`, in line 20, which potentially excludes additional features, if the performance of X is increased by doing so.

The procedure `FurtherExclusion` in Algorithm 6 excludes features in an iterative manner. Line 26 ensures that this procedure will remove features until there is no gain in removing features, or until the set contains only two features. Lines 29-37 locates the feature which, when removed, will increase the performance of X the most, and update the sets accordingly.

Algorithm 6 The Sequential Forward Floating Selection Method

while Not Stopping Criterion **do**
  $x_{best} \leftarrow \{d_0\}$
  for all $d_i \in D \backslash \{d_0\}$ **do**
    if $J(X \cup \{d_i\}) \geqslant J(X \cup x_{best})$ **then**
      $x_{best} \leftarrow \{d_i\}$
    end if
  end for
  $X \leftarrow X \cup x_{best}$
  $D \leftarrow D \backslash x_{best}$
  $x_{worst} \leftarrow \{x_0\}$
  for all $x_i \in X \backslash \{x_0\}$ **do**
    if $J(X \backslash \{x_i\}) \geqslant J(X \backslash x_{worst})$ **then**
      $x_{worst} \leftarrow \{x_i\}$
    end if
  end for
  if $(X_{best} \neq X_{worst})$ **then**
    $X \leftarrow X \backslash x_{worst}$
    $D \leftarrow D \cup x_{worst}$
    if $|X| > 2$ **then**
      FURTHEREXCLUSION(X, D)
    end if
  end if
end while

procedure FURTHEREXCLUSION(X, D)
  $r \leftarrow$ true
  while $|X| > 2 \wedge r$ **do**
    $x_{worst} \leftarrow \emptyset$
    $r \leftarrow$ false
    for all $x_i \in X$ **do**
      if $J(X \backslash \{x_i\}) \geqslant J(X \backslash x_{worst}) \wedge J(X \backslash \{x_i\}) \geqslant J(X)$ **then**
        $x_{worst} \leftarrow \{x_i\}$
        $r \leftarrow$ true
      end if
    end for
    if r **then**
      $X \leftarrow X \backslash x_{worst}$
      $D \leftarrow D \cup x_{worst}$
    end if
  end while
end procedure

12 FEATURE SELECTION EXPERIMENTS

This part of the thesis has presented several tools and techniques that may contribute in solving the problem of constructing a decision model for expansion strategies in RTS games. This chapter describes how to utilise the tools and techniques presented to attain reduced feature sets. This is achieved through experiments, in which three different feature selection methods are applied to a set of features. The goal of the experiments is to generate a decision model for each feature selection method of less complexity–and ideally better prediction–than the decision model using the full feature set. Chapter 13 describes evaluation scenarios using the reduced feature sets attained in this chapter.

12.1 IMPLEMENTATION OF FEATURE SELECTION METHODS

Three different feature selection methods have been implemented for these experiments, all of which are described in Chapter 11. All implementation and experiments are made using MATLAB[1]. Implementation of the feature selection methods all use ANNs to evaluate sets of features. This is done by training an ANN using backpropagation, where each input node corresponds to a feature from the specified feature set. The network is evaluated using Mean Absolute Error (MAE) as the performance measure. MAE for an ANN is defined as:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|a_i - t_i| \,.$$

where n is the total number of entries, a_i is the actual value output by the ANN for an entry i and t_i is the target value expected for an entry i. The lower the MAE is for a given feature set, the better the feature set is perceived. Normal settings are used for the ANNs as defined by MATLAB. All three implementations have been set to select feature sets containing a total number of 15 features.

12.1.1 *Data Set Used for Experiments*

A large data set containing samples of the extracted features has been assembled as described in Chapter 10. The data set contains a total of 28 features with 81.844 entries of data. The data is divided into

1 http://www.mathworks.com/products/matlab/

60% training data, 10% validation data and 30% testing data. The validation data is used for measuring performance during training. During feature selection, division of the data into the three categories is handled by the MATLAB implementation using random division.

12.2 FEATURE SET PERFORMANCE

This section details the performance of each feature set gained through experiments with the three feature selection methods. A full trace is given for each method, containing added/removed features, along with the corresponding MAEs of the given feature set at specific iterations of the method.

12.2.1 *Sequential Forward Selection*

Table 12.1 shows the full trace of running SFS on the sample set, selecting 15 features from the full feature set. As it can be seen, the MAE exhibits an overall decrease as more features are added, which means an increase in performance.

Table 12.1: Trace of the Sequential Forward Selection method using the feature set containing 28 features.

Iteration	Feature added	MAE
1	OwnGroundDistanceNormalized	0.1577993
2	EnemyGroundDistanceNormalized	0.1464801
3	GameTime	0.1405489
4	NumberOfOwnExpansions	0.1353540
5	NumberOfPossibleExpansions	0.1340203
6	OpenSides	0.1324332
7	NumberOfGas	0.1340026
8	OpponentUnitScore	0.1304485
9	CurrentPresenceInShortestPath	0.1296689
10	OpponentRazingScore	0.1309179
11	OwnFlyingDistanceNormalized	0.1294599
12	NumberOfEnemyExpansions	0.1269970
13	NumberOfChokepoints	0.1272868
14	EnemyFlyingDistanceNormalized	0.1243576
15	BestResources	0.1270499

12.2.2 *Sequential Backwards Selection*

Table 12.2 shows the full trace of running SBS using the sample set, removing 13 features to yield a feature set of 15 features. As it can be seen, as with SFS, the MAE exhibits an overall decrease. The end performance is slightly better than that of SFS.

Table 12.2: Trace of the Sequential Backwards Selection method using the feature set containing 28 features.

Iteration	Feature removed	MAE
1	EnemyFlyingDistanceNormalized	0.1273334
2	HighestPresenceInShortestPath	0.1405049
3	MapPosition	0.1296795
4	OpponentRazingScore	0.1297889
5	OwnAirUnits	0.1347829
6	Race	0.1316123
7	OpenSides	0.1290031
8	OpponentUnitScore	0.1303906
9	GameTime	0.1393028
10	BestGas	0.1276848
11	MapSize	0.1323159
12	BestResources	0.1296734
13	OpponentAirUnits	0.1256307

12.2.3 *Sequential Forward Floating Selection*

Table 12.3 shows a partial trace of running SFFS using the sample set, selecting 15 features from the full feature set. The trace shows the iterations in which a feature is added without later being removed, as well as the MAE for the feature set at that point. For a full trace see Appendix B.

12.3 SUMMARY

Through experiments, a feature set for each of the three feature selection methods has successfully been found. Table 12.4 shows the difference between the three feature sets, as well as a performance comparison of the feature sets. A checkmark (✓) denotes that the specified feature is included in a feature set. The performance measure used in this chapter is MAE, being the average distance between the target value and the actual value. Table 12.4 shows that the difference between the MAE of the three methods is relatively low. While MAE as

Table 12.3: Partial trace of the Sequential Forward Floating Selection method using the feature set containing 28 features.

Iteration	Feature added	MAE
1	OwnGroundDistanceNormalized	0.1573412
5	EnemyGroundDistanceNormalized	0.1480936
43	NumberOfGas	0.1277823
50	HighestPresenceInShortestPath	0.1277823
51	NumberOfOwnExpansions	0.1277823
58	OwnFlyingDistanceNormalized	0.1279682
60	BestGas	0.1279682
64	NumberOfEnemyExpansions	0.1279682
65	OpponentAirUnits	0.1279682
68	NumberOfPossibleExpansions	0.1279682
72	OpponentBuildingScore	0.1279682
77	OpenSides	0.1279682
78	OwnUnitScore	0.1279682
79	NumberOfMinerals	0.1279682
80	BestMinerals	0.1279682

a performance measure generally shows how well a model performs, it does not necessarily reveal how accurate the model is.

Table 12.4 also contains a custom feature set. This feature set is based solely on expert knowledge by the authors. A performance of the feature set is not given, since no feature selection has been performed in order to produce the set. Furthermore a set containing only the features common to the other feature sets is tested. This feature set is also shown in Table 12.4. As with the custom set, there is no performance listed with the common feature set, since no feature selection was applied.

A detailed performance review of all feature sets–including *Custom*, *Common* and *Full*–can be found in Chapter 13.

Table 12.4: Overview of the obtained feature sets. A checkmark (✓) denotes that the specific feature is included in a feature set.

	SFS	SBS	SFFS	Custom	Common
NumberOfGas	✓	✓	✓		
NumberOfMinerals		✓	✓	✓	
NumberOfChokepoints	✓	✓		✓	
NumberOfPossibleExpansions	✓	✓	✓		
NumberOfOwnExpansions	✓	✓	✓	✓	✓
NumberOfEnemyExpansions	✓	✓	✓	✓	✓
CurrentPresenceInShortestPath	✓	✓			
HighestPresenceInShortestPath			✓		
GameTime	✓			✓	
BestMinerals		✓	✓		
BestGas			✓		
BestResources	✓				
OpenSides	✓		✓		
OwnFlyingDistanceNormalized	✓	✓		✓	
OwnGroundDistanceNormalized	✓	✓	✓	✓	✓
EnemyFlyingDistanceNormalized	✓			✓	
EnemyGroundDistanceNormalized	✓	✓	✓	✓	✓
Race				✓	
MapPosition					
OpponentAirUnits			✓		
OwnAirUnits					
MapSize					
OpponentUnitScore	✓				
OpponentBuildingScore		✓	✓		
OpponentRazingScore	✓			✓	
OwnUnitScore		✓	✓		
OwnBuildingScore		✓			
OwnRazingScore		✓			
Performance of feature set (MAE)	0.1270499	0.1256307	0.1279682	N/A	N/A

13 FEATURE SELECTION EVALUATION

Feature selection methods have been evaluated so far solely by using MAE–obtained by evaluating logged data from replays–as detailed in Chapter 12. The focus of this section is the utilisation of the proposed feature sets for predicting player expansions. This is achieved through the use of decision models, which are evaluated through various methods.

13.1 USING ARTIFICIAL NEURAL NETWORKS

ANNs have been used during feature selection and is also used for evaluating the final feature sets. For each of the final feature sets, an ANN is trained with the features specified by the set. The ANN is evaluated and the performance of the ANN is considered a measure of the particular feature set.

Whenever a game state for a specific expansion site is evaluated through an ANN, the output is a number signifying the likelihood of the site being the current target of a player's expansion. The target data for training has an output of 1.0 for sites observed to be built upon and an output of 0.0 for sites that were not chosen when another was.

The result of this approach is that all game time values the network has been trained with results in at least one build action. In other words, the network has been trained to determine *where* a player will build their expansion, given that they are building one at the given state provided as input to the network. The network is not trained to determine *when* a player will construct an expansion–this is considered a separate concern.

The ANNs are trained with MATLAB, using the collected data. 60% of the data is used for training, 10% for validation and 30% for testing the resulting network. Contrary to Chapter 12, division of data is not randomly chosen by MATLAB, but is divided manually in order to secure consistent tests.

13.2 USING DECISION TREES

The feature sets are also evaluated using decision trees. The purpose is to gain a broader view of the feature sets by evaluating the sets with another decision model. The reason for using decision trees is that it is a fairly well-known, simple and accessible method.

The decision trees are trained with MATLAB using the same data as used for training ANNs. Training and validation data is combined, resulting in 70% of the data being used for training and 30% for testing the resulting trees. The output is classified as one or zero, indicating whether the expansion site is chosen or not. Note that decision trees do not output continuous values; data is instead used to identify a correct leaf node which contains the value to output.

For comparison purposes, the decision trees and ANNs are tested using the same performance measures.

13.3 FURTHER TESTING

In order to evaluate the feature sets from Chapter 12, the feature sets are used for training several ANNs and decision trees.

The decision models constructed using the feature subsets provided by the feature selection methods are evaluated in various ways. The following methods of evaluation have been chosen:

MEAN ABSOLUTE ERROR Using the same evaluation method as used for the feature selection performance function, introduced in Chapter 12.

HIT RATE The hit rate is the amount of correctly identified expansion sites, given that only the site with the highest network output is considered. Using this method, the values of the network output for the incorrect sites does not matter as long as the network output for the correct site is higher than the values of all other sites being considered.

Consider the sample in Table 13.1a. S is a potential expansion site along with the state of the game at the moment of consideration. $\mathrm{Net}(S)$ is the network output for S and $\mathrm{Target}(S)$ is the actual/correct data that has been observed. The sample counts as a *hit* when measuring hit rate, as the network output for the correct expansion site, S_2, is higher than the network output for all other possible expansion sites. The sample in Table 13.1b is considered a miss, as S_6 has the highest network output, yet S_7 is the correct expansion site. If the samples from Table 13.1a and Table 13.1b is the complete sample set, then the hit rate is $\frac{1}{2} = 50\%$.

Formally, for each possible build-site $b_i \in B$, the probability of choosing b_i as the build-site is calculated as

$$P(b_i) = \begin{cases} 1 & \text{for } \mathrm{out}(b_i) = \max\limits_{b \in B} \mathrm{out}(b) \\ 0 & \text{for } \mathrm{out}(b_i) \neq \max\limits_{b \in B} \mathrm{out}(b) \end{cases}.$$

Meaning that if the network output of b_i, $\mathrm{out}(b_i)$, is the highest output for all possible build-sites, the probability of choosing b_i is one,

	Net(S)	Target(S)
S_1	0.2	0
S_2	0.9	1
S_3	0.2	0
S_4	0.5	0

(a) A sample in which the network output closely approximates the target.

	Net(S)	Target(S)
S_5	0.2	0
S_6	0.7	0
S_7	0.6	1
S_8	0.1	0

(b) A sample in which the network output is far from the target data.

and if this is not the case, the probability is zero. The hit rate is the percentage of correctly identified build sites using this method. Note that in the cases where there is more than one possible expansion site yielding the maximum value, it is considered a conflict, which will be counted as a miss as it is not possible to identify the correct expansion site.

DISTRIBUTION The distribution is the amount of correctly identified expansion sites on average, given that the normalised network output for an expansion site is treated as the probability for choosing that particular expansion site. The difference from the hit rate is that when calculating the distribution, the difference between network outputs matter.

In the sample of Table 13.1a, the correct expansion site is S_2, the distribution for the sample is $\frac{0.9}{0.2+0.9+0.2+0.5} = 0.5$. Even though S_2 is the highest network output for the sample, it is not considered a total hit, as the probability for choosing this expansion site is 50% when using the distribution. The distribution for the sample in Table 13.1b is $\frac{0.6}{0.2+0.7+0.6+0.1} = 0.375$, so even though S_7 is not the highest network output, there is still a 37.5% chance of selecting S_7 as the expansion site. If the samples from Table 13.1a and Table 13.1b is the complete sample set, the distribution is $\frac{0.5+0.375}{2} = 43.75\%$.

The distribution is calculated as follows:
For each possible build-site $b_i \in B$, the probability of choosing b_i as the build-site is

$$P(b_i) = \frac{out(b_i)}{\sum_{b \in B} out(b)}.$$

Where $out(b_i)$ is the network output for expansion site b_i and B is the set of all possible expansion sites. In this approach, the output value for the correct build site is normalised with the total output for all available expansion sites.

Table 13.1: Results for each set chosen by feature selection using ANNs.

Feature Set	Hit Rate	Distribution	MAE
SFS	62.60%	51.46%	0.1182
SBS	64.66%	51.24%	0.1146
SFFS	64.43%	48.55%	0.1193
Common	60.61%	43.13%	0.1361
Custom	62.47%	49.22%	0.1232
Full Feature Set	64.26%	53.03%	0.1136

13.4 RESULTS

This section will present and discuss test results for ANNs and decision trees.

13.4.1 *Artificial Neural Network Results*

The results of the experiments using ANNs can be seen in Table 13.1. The first column is the feature set used during the training of the ANN, e.g. the feature set marked as **SFS** is the feature set resulting from the SFS feature selection method. The remaining columns pertain to the evaluation methods discussed in Section 13.3.

All feature sets described in Chapter 12 are included in Table 13.1, as well as the results for random values and the full feature set for comparison. The MAE values are lower than the ones presented in Chapter 12. This is because the output layer of the ANNs has been limited to provide only output values in the range $0.0 \leqslant \text{output} \leqslant 1.0$. This change makes the possible distances obtainable through MAE lower, but still has the same overall significance.

When using ANNs, the full feature set has the potential to be at least as good as any subset of the full feature set. The weights for any non-influential feature may be set to a zero value or near-zero value, meaning that the feature will have little or no influence on the result. However, during training of the ANN, the weights of non-influential features are not always reduced sufficiently, which may result in some amount of noise.

As it can be observed in Table 13.1, the most reduced feature sets are performing close to that of the full feature set. This indicates that the exclusion of the features not chosen by the feature selection algorithms have not significantly reduced the performance of the ANNs. Still, ANNs based on the features from the two smallest sets, *Custom* and *Common*, have the worst hit rates at 62.47% and 60.61% respectively. This, coupled with the relatively low distribution and high MAE values, suggests that the methods lack some of the important features that

Table 13.2: Results for each set chosen by feature selection using decision trees.

Feature Set	Hit Rate	Distribution	MAE
SFS	43.54%	48.72%	0.1053
SBS	44.17%	49.54%	0.1076
SFFS	44.74%	49.59%	0.1050
Common	45.53%	47.89%	0.0990
Custom	43.87%	49.23%	0.1061
Full Feature Set	44.47%	49.79%	0.1068

are included in the other methods. *Custom* covers 11 features, while *Common* covers only four.

13.4.2 *Decision Trees Results*

Using the same feature sets as in Section 13.4, Table 13.2 contains the results by the decision trees. The hit rate is lower than that of the ANNs while the distribution is about the same. One reason for this might be that the feature sets were selected using ANNs and the selected features are better suited for that type of decision model. A reason that may be considered more likely is that the number of possible outputs for the decision tree is limited. Each leaf node maps to exactly one output, limiting the range of outputs to no more than the number of leaf nodes. Note also that the values of the leaf nodes are always either one or zero. Because of this, conflicts are likely to occur. The percentage of conflicts using the decision trees is on average 37.42%, while the average percentage of conflicts when using ANNs is less than 0.003%.

In general, the results of the decision trees based on the various feature sets are very similar, with a span between the highest and lowest hit rate of 2.01%. Oddly enough, the tree based on the fewest features, *Common*, has the highest hit rate and lowest MAE, conflicting directly with the results of the ANNs. This may be due do coincidence, based on the low span between highest and lowest hit rate, or due to the high amount of conflicts using decision trees. If there is a pattern in the conflicts, the fact that fewer features are included in the *Common* feature set may influence the end result. Another reason may be the simplified structure of the decision trees, when compared to ANNs. For simplicity, all decision trees have been trained with binary splitting on each feature, so some of the more complex connections between the variables are possibly lost.

In order to form a baseline for comparison, a random approach, shown in Table 13.3, has been evaluated as well. During the evalu-

Table 13.3: Results for choosing at random, used as a baseline for comparison.

	Hit Rate	Distribution	MAE
Random	14.71%	14.70%	0.4980

Table 13.4: Training times for ANNs and decision trees, in seconds.

	SFS	SBS	SFFS	Full	Custom	Common
ANNs	173.1	186.7	261.1	233.0	115.9	69.2
Trees	2.7	5.6	4.4	7.7	3.8	1.9

ation of the random approach, instead of using a trained network, a function returning a random value in the range zero to one is used. The purpose is to discover which rates it is possible to reach using only random chance. Note that the hit rate for the random approach is around 15%. This is due to the fact that for each expansion created, several expansion sites were not chosen and so the chance of getting a hit depends on the amount of possible expansion sites. The performance of the random baseline indicates that the ANNs and decision trees trained using feature sets are performing at a higher level than would be expected, if there was no connection between the choice of expansion site and the selected features.

Table 13.4 and Table 13.5 shows average training and running times for ANNs and decision trees, given various feature sets. In spite of the substantial difference in training times between the two methods, every ANN was trained in less than five minutes. Since training occurs offline, such training times are deemed fully acceptable.

The average running times also shows a substantial difference between the two methods. Since the methods only need to be run every time a base location is needed, and since it can be run asynchronous with the game, the running times of both methods are useable in live gaming sessions.

13.5 SCENARIO

This section describes a scenario in StarCraft in which a decision model is used to predict the expansion location of a player. The section contains reasoning based on experience for whether or not the output

Table 13.5: Running times for ANNs and decision trees, in seconds.

	SFS	SBS	SFFS	Full	Custom	Common
ANNs	0.069	0.068	0.069	0.068	0.068	0.069
Trees	0.003	0.003	0.003	0.002	0.003	0.002

of the decision model is sound. As a complete human analysis of a neural network containing 10 hidden nodes and 15 features is a very complex job, this section is based primarily on the opinions of the authors.

The scenario has been chosen from one of the replays in the set, used during testing. In this scenario, two players compete on a map which supports up to four players. This section covers a single expansion decision of player A. The game state representation for each possible expansion site at the time of expansion can be seen in Table 13.6. Each column maps to an expansion site and each row represents a feature with the exception of the last row which displays the output value of a neural network trained with the feature set found by SBS in Chapter 12. As SBS is used for this example, all features that are not in the SBS feature set have been omitted.

Figure 13.1 shows an overview of the game at the moment of the expansion. Player A has two bases marked A, while player B has three bases, marked B. As it can be observed on Table 13.6 player A has not destroyed any of player B's buildings, however, player A performs active scouting and therefore knows the location of player B's three bases.

The last row in Table 13.6 shows that, according to the decision model, player A's highest preference is Site 2, with an output value of 0.5279. This is also the actual location chosen by the player. Site 2 and Site 5, are both rated significantly higher than the remaining sites. By observing their values in relation to the other sites, it would seem that variables representing distances are an important factor. From the authors' perspective, it seems like a sound strategy to construct expansions close to existing bases when possible. Doing so makes it possible to construct the expansion swiftly, as the worker required to construct the resource deposit need not travel very far. Bases that are closely clustered are usually also easier to defend due to low travelling distances for allied forces. A low distance also facilitates moving workers from one site to another at a minimal risk. Both flying distance and ground distance seems important; however, the data indicates that ground distance is the more important of the two. Testing shows that the minerals of sites also influence the choice of expansion site, but due to the relatively similar values for the sites of this particular map, the influence is negligible.

The variables pertaining to the general game state, such as NumberOfOwnExpansions, OwnUnitScore and OwnRazingScore may influence the impact of other variables, however since this scenario covers only a single expansion choice, such connections may be difficult to deduce.

Table 13.6: Table containing the game state representation of the scenario.

	Site 1	Site 2	Site 3	Site 4	Site 5	Site 6	Site 7	Site 8	Site 9
NumberOfGas	5000	5000	5000	5000	5000	6192	6192	5000	5000
NumberOfMinerals	13500	12000	13500	10500	12000	9507	9507	12000	10500
NumberOfChokepoints	1	2	1	2	2	2	2	2	2
NumberOfPossibleExpansions	14	14	14	14	14	14	14	14	14
NumberOfOwnExpansions	2	2	2	2	2	2	2	2	2
NumberOfEnemyExpansions	3	3	3	3	3	3	3	3	3
CurrentPresenceInShortestPath	0	0	0	0	0	4	4	0	0
BestMinerals	13500	13500	13500	13500	13500	13500	13500	13500	13500
OwnFlyingDistanceNormalized	0.7281	0.3998	1	0.7177	0.3227	0.5114	0.4949	0.9664	0.7262
OwnGroundDistanceNormalized	1	0.4404	0.9921	0.7615	0.5090	0.5156	0.5088	0.9584	0.7636
EnemyGroundDistanceNormalized	0.7482	0.9739	1	0.5052	0.7722	0.5132	0.5201	0.4983	0.7671
OpponentBuildingScore	2215	2215	2215	2215	2215	2215	2215	2215	2215
OwnUnitScore	3950	3950	3950	3950	3950	3950	3950	3950	3950
OwnBuildingScore	2300	2300	2300	2300	2300	2300	2300	2300	2300
OwnRazingScore	0	0	0	0	0	0	0	0	0
Model output	0.0278	0.5279	0.0469	0.0116	0.3487	0.0484	0.0538	0.0030	0.0342

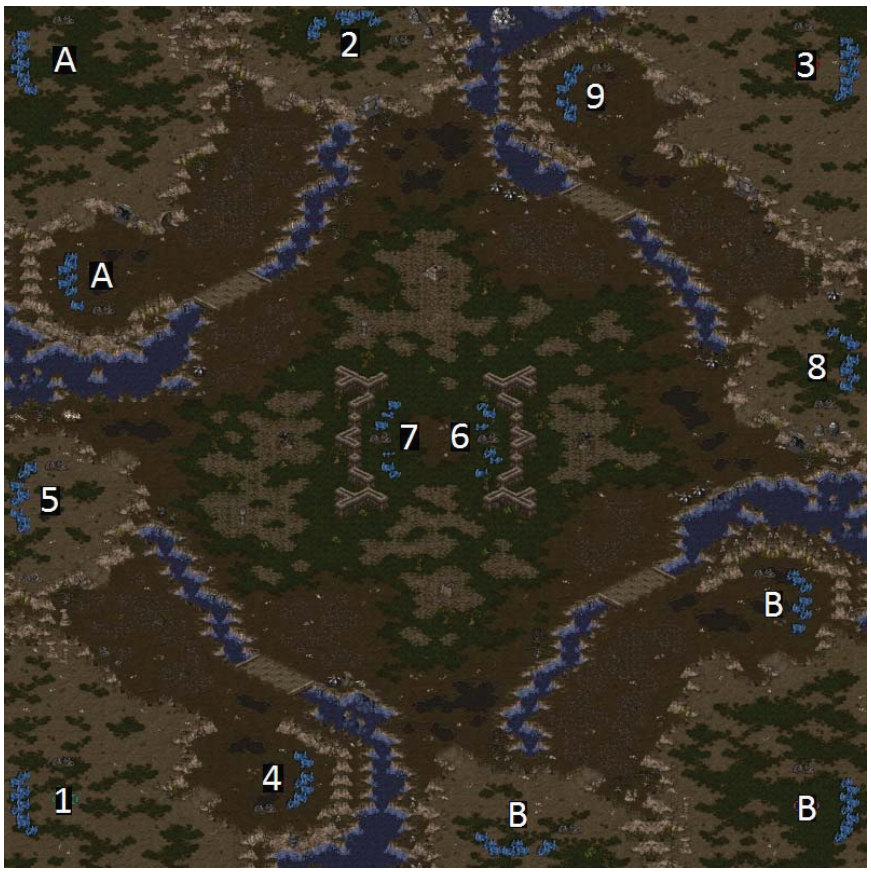


Figure 13.1: A visual representation of the scenario.

14 DISCUSSION AND FUTURE WORK

This part of the thesis has touched upon the subject of utilising machine intelligence methods for choosing expansion locations in RTS games, using a data-driven approach. The main idea is to mimic human behaviour by collecting data from replays played by human players. The test bed chosen for the thesis, StarCraft, has thousands of replays available on websites such as iCCup[1], facilitating data for training decision models for choosing expansion locations. Using expert knowledge, 28 features that may be considered by players when expanding, has been identified and sample data has been gathered from the replays.

Communication with StarCraft has been achieved through DLL injection, by using Chaoslauncher[2] along with BWAPI[3] and BWTA[4]. Using three feature selection methods, three sets of 15 variables were found. Also added were a set containing the common features from the three sets, a custom set containing the features considered most important by the authors, and a full feature set. Two decision methods have been used for implementing several decision models to allow an agent to mimic the behaviour of human players when expanding, based on the state of up to 28 features. ANNs were chosen, as they have previously been proven successful in strategy prediction and decision trees were added for comparison.

The six sets were used for training six ANNs and six decision trees. Finally, a method returning a valid random output was used as a baseline for comparison. No one feature selection method shows substantially better results than the others. The results of the full feature set indicate a low amount of noise among the data.

The best hit rate of 64.43% was achieved with an ANN based on the features of SBS, while the lowest MAE, at 0.1136, was achieved with an ANN based on the full feature set. The highest distribution of 53.17% was achieved using a decision tree based on the custom feature set.

Every combination of features and decision models have a hit rate higher than 50%, substantially better than the random baseline, having a hit rate of 14.71%. Since data extraction and training is done offline, and online use of the decision models is not computationally heavy, the methods are useable in live gaming sessions.

1 http://www.iccup.com/
2 http://winner.cspsx.de/Starcraft/
3 http://code.google.com/p/bwapi/
4 http://code.google.com/p/bwta/

14.1 FUTURE WORK

This section covers possible directions for future work of the data-driven expansion approach presented in this thesis, along with ideas to speed up the various feature selection methods.

14.1.1 *Predicting Expansions*

The current decision models are trained for selecting the most likely expansion site of a human player at the moment of the game state. Using a method for approximating a future game state, the models may be used for predicting where an opponent will expand. A similar method may be applied for approximating the location of an already placed expansion.

Analysing the amount of players expanding in certain time intervals may help determine when an opponent is likely to expand, while the state of the variables at the time of expansion may help make a better approximation. Such information can in turn be used by an agent to know where and when to scout for the opponent, and help approximate the best locations to make expansions of its own.

14.1.2 *Dealing with Unknown Game States*

In general, many of the features that are used to feed the decision models need to be collected or approximated in an online setting, if the target game only allows for imperfect knowledge. Since accessing the state of the features may require some degree of scouting, and since the scouting method presented in this section relies on the state of some variables itself, it may be beneficial to adopt a method for approximating the initial values.

14.1.3 *Player Modelling*

As the decision models are currently tuned to make approximations based on the behaviour of a generalised player model, it is possible that a better performance may be gained from classifying individual players. One way of doing so is the use of online learning through a reinforcement learning technique. In this case, the output of decision models may be used as a basis for initialising a Q-table for Q-learning. The table may then be updated in the usual Q-learning style, using whether or not the correct base location is targeted as reward.

14.1.4 *Alternative Decision Models*

Decision models like Bayesian networks and influence diagrams, handles actions under uncertainty more naturally than ANNs and decision trees. These may be considered a solution to the problem of imperfect knowledge. However, Bayesian networks and influence diagrams requires some degree of expert knowledge to design, taking away the advantage of the current decision models. Given a thorough analysis of the problem domain–or the opinions of expert StarCraft players–an influence diagram created and trained on the basis of the same features as the ANNs, could make for some interesting comparisons.

14.1.5 *Use in Full Scale Real-time Strategy Games*

In a full scale RTS game, the information gained from scouting the opponent and choosing expansion locations may be used by other modules of an agent to gain knowledge on which base locations should be defended and where to attack the opponent. In general, given the great amounts of replay data available, a data-driven approach similar to the one used for expanding may be used for strategy predictions, along with counter strategies.

14.1.6 *Alternative Performance Measures for Feature Selection*

The current performance measure used for feature selection, presented in Section 12.1, seeks to reduce the overall distance between the results of the decision models and the actual data. One of the main uses of the decision models is to estimate a good expansion location given a set of features. As mentioned in Section 13.3, a logical way to choose an expansion location is to pick the location with the features that makes for the highest value in the decision model. A correct guess is then considered a hit, while a wrong guess is considered a miss.

Using the hit rate of a decision model as the performance measure when performing feature selection might help adjust the model to increase the number of times a correct location is found.

14.1.7 *Improving Feature Selection Methods*

This section presents several ideas on reducing MAE and speeding up execution of the chosen feature selection methods using expert knowledge.

14.1.7.1 *Modified Starting Sets*

A natural way to improve a selection method is to modify the starting sets using expert knowledge by either adding or removing–depending on the method–features that are sure to be selected. As an example, if n features are deemed significant and are added to the set, SFS will be able to skip the first n iterations. Alternatively, to avoid depending too heavily on the accuracy of expert knowledge, the features may be partitioned into groups of varying importance. The groups may be used for speeding up the feature selection methods by allowing each iteration of the feature selection method to consider only features from the best group most of the time. For example, using ϵ-greedy selection; with probability ϵ choose from any group and with probability $1 - \epsilon$ choose from the best non-empty group.

14.1.7.2 *Feature Dependencies*

Using expert knowledge, features that are related to one another may be identified. The information may in turn be used to include related features, every time a feature is identified for inclusion by a feature selection method.

Part III

APPENDIX

A

FULL LIST OF FEATURES

Overview table for features included for feature selection. Some ranges have approximated maximal values, less than the theoretical maxima. As an example, *NumberOfMinerals* stems from a 'normal' maximum of $50,000 \cdot 8 = 400,000$ even though the theoretical maxima is much larger, and depends on the number of patches that can fit on the entire map.

Table A.1: Summary of the 28 features used in the thesis, including the full name, the domain of the features as well as a short description.

Name	Domain	Description
NumberOfGas	$[0, 100000] \in \mathcal{N}$	Gas on location
NumberOfMinerals	$[0, 400000] \in \mathcal{N}$	Minerals on location
NumberOfChokepoints	$[0, 10] \in \mathcal{N}$	Number of entrances to region
NumberOfPossibleExpansions	$[0, 20] \in \mathcal{N}$	Possible expansion locations on map
NumberOfOwnExpansions	$[0, 25] \in \mathcal{N}$	Players' current expansions
NumberOfEnemyExpansions	$[0, 25] \in \mathcal{N}$	Opponents' current expansions
CurrentPresenceInShortestPath	$[0, 25] \in \mathcal{N}$	Occurence in shortest path from any enemy base to any owned base
HighestPresenceInShortestPath	$[0, 25] \in \mathcal{N}$	For every available baselocation, the highest occurence in shortest path from any enemy base to any owned base
GameTime	$[0, 2147483647] \in \mathcal{N}$	Logical time in frames since match start
BestMinerals	$[0, 400000] \in \mathcal{N}$	For every available baselocation, the highest amount of minerals
BestGas	$[0, 100000] \in \mathcal{N}$	For every available baselocation, the highest amount of gas
BestResources	$[0, 500000] \in \mathcal{N}$	For every available baselocation, the highest amount of gas + minerals
OpenSides	$[0, 4] \in \mathcal{N}$	Number of sides on location with a set distance from sides of map
OwnFlyingDistanceNormalized	$[0, 1] \in \mathcal{R}$	Lowest flying distance from own base locations, normalised to highest flying distance from own base locations

Name	Domain	Description
OwnGroundDistanceNormalized	$[0,1] \cup [-1] \in \mathcal{R}$	Lowest ground distance from own base locations, normalised to highest ground distance from own base locations
EnemyFlyingDistanceNormalized	$[0,1] \in \mathcal{R}$	Lowest flying distance from enemy base locations, normalised to highest flying distance from enemy base locations
EnemyGroundDistanceNormalized	$[0,1] \cup [-1] \in \mathcal{R}$	Lowest ground distance from enemy base locations, normalised to highest ground distance from enemy base locations
Race	$[0,2] \in \mathcal{N}$	Starting race of the player
MapPosition	$[0,8] \in \mathcal{N}$	Position on map, where the map is split up in 9 quadrants
OpponentAirUnits	$[0,700] \in \mathcal{N}$	Number of enemy air units that can attack the ground, not counting interceptors
OwnAirUnits	$[0,100] \in \mathcal{N}$	Number of own air units that can attack the ground, not counting interceptors
MapSize	$[4096,65536] \in \mathcal{N}$	Size of map, height·width
OpponentUnitScore	$[0,3500000] \in \mathcal{N}$	Score indicating combined strenght of units created by enemies
OpponentBuildingScore	$[0,2450000] \in \mathcal{N}$	Score indicating combined strenght of buildings created by enemies
OpponentRazingScore	$[0,7350000] \in \mathcal{N}$	Score indicating how many and how valuable buildings enemies have destroyed
OwnUnitScore	$[0,500000] \in \mathcal{N}$	Score indicating combined strenght of units created by player
OwnBuildingScore	$[0,400000] \in \mathcal{N}$	Score indicating combined strenght of buildings created by player
OwnRazingScore	$[0,1050000] \in \mathcal{N}$	Score indicating how many and how valuable buildings the player have destroyed

B

TRACE OF SEQUENTIAL FORWARD FLOATING SELECTION

Table B.1: Full trace of the Sequential Forward Floating Selection method using the data set containing 28 features.

Iteration	Feature Count	Feature added	Feature removed	MAE
1	1	OwnGroundDistanceNormalized	N/A	0.1573412
2	2	OwnBuildingScore	N/A	0.1468833
3	2	EnemyGroundDistanceNormalized	OwnBuildingScore	0.1462312
4	2	GameTime	EnemyGroundDistanceNormalized	0.1492445
5	2	EnemyGroundDistanceNormalized	GameTime	0.1480936
6	3	OwnBuildingScore	N/A	0.1381917
7	3	NumberOfGas	OwnBuildingScore	0.1381917
8	3	OwnUnitScore	NumberOfGas	0.1381917
9	4	NumberOfGas	N/A	0.1374030
10	4	NumberOfOwnExpansions	NumberOfGas	0.1374030
11	5	HighestPresenceInShortestPath	N/A	0.1341648
12	6	OwnFlyingDistanceNormalized	N/A	0.1333626
13	6	OpponentRazingScore	OwnUnitScore	0.1333626
14	6	NumberOfGas	OpponentRazingScore	0.1333626
15	7	OpponentBuildingScore	N/A	0.1296521
16	7	OwnAirUnits	HighestPresenceInShortestPath	0.1296521
17	7	NumberOfChokepoints	OwnFlyingDistanceNormalized	0.1296521
18	8	OwnFlyingDistanceNormalized	N/A	0.1306152
19	9	NumberOfPossibleExpansions	N/A	0.1298761

Iteration	Feature Count	Feature added	Feature removed	MAE
20	10	OwnUnitScore	N/A	0.1308199
21	10	OwnBuildingScore	OwnAirUnits	0.1308199
22	10	EnemyFlyingDistanceNormalized	OwnFlyingDistanceNormalized	0.1308199
23	10	OwnFlyingDistanceNormalized	OpponentBuildingScore	0.1308199
24	10	OwnRazingScore	NumberOfOwnExpansions	0.1308199
25	10	NumberOfEnemyExpansions	EnemyFlyingDistanceNormalized	0.1308199
26	10	MapPosition	OwnRazingScore	0.1308199
27	11	Race	N/A	0.1307027
28	11	NumberOfOwnExpansions	MapPosition	0.1307027
29	11	BestMinerals	OwnFlyingDistanceNormalized	0.1307027
30	11	OpenSides	OwnBuildingScore	0.1307027
31	12	MapSize	N/A	0.1268168
32	12	EnemyFlyingDistanceNormalized	NumberOfChokepoints	0.1268168
33	12	BestGas	Race	0.1268168
34	12	OwnAirUnits	OwnUnitScore	0.1268168
35	12	NumberOfMinerals	OpenSides	0.1268168
36	12	OwnFlyingDistanceNormalized	EnemyFlyingDistanceNormalized	0.1268168
37	12	OpponentAirUnits	OwnAirUnits	0.1268168
38	12	OwnBuildingScore	BestGas	0.1268168

Iteration	Feature Count	Feature added	Feature removed	MAE
39	12	CurrentPresenceInShortestPath	NumberOfGas	0.1268168
40	13	OwnUnitScore	N/A	0.1277823
41	13	OpponentUnitScore	OwnBuildingScore	0.1277823
42	13	OpponentBuildingScore	OwnUnitScore	0.1277823
43	13	NumberOfGas	CurrentPresenceInShortestPath	0.1277823
44	13	OpenSides	NumberOfMinerals	0.1277823
45	13	CurrentPresenceInShortestPath	BestMinerals	0.1277823
46	13	BestResources	MapSize	0.1277823
47	13	Race	OpponentBuildingScore	0.1277823
48	13	OwnAirUnits	CurrentPresenceInShortestPath	0.1277823
49	13	OwnUnitScore	Race	0.1277823
50	13	HighestPresenceInShortestPath	NumberOfOwnExpansions	0.1277823
51	13	NumberOfOwnExpansions	OpponentAirUnits	0.1277823
52	13	MapSize	OpponentUnitScore	0.1277823
53	13	OpponentRazingScore	OwnUnitScore	0.1277823
54	13	OpponentBuildingScore	MapSize	0.1277823
55	13	CurrentPresenceInShortestPath	OpponentRazingScore	0.1277823
56	13	OwnUnitScore	OwnFlyingDistanceNormalized	0.1277823
57	13	OpponentUnitScore	OpponentBuildingScore	0.1277823

Iteration	Feature Count	Feature added	Feature removed	MAE
58	14	OwnFlyingDistanceNormalized	N/A	0.1279682
59	14	MapPosition	OpponentUnitScore	0.1279682
60	14	BestGas	CurrentPresenceInShortestPath	0.1279682
61	14	NumberOfMinerals	NumberOfPossibleExpansions	0.1279682
62	14	OwnRazingScore	NumberOfEnemyExpansions	0.1279682
63	14	NumberOfChokepoints	MapPosition	0.1279682
64	14	NumberOfEnemyExpansions	BestResources	0.1279682
65	14	OpponentAirUnits	NumberOfChokepoints	0.1279682
66	14	GameTime	OwnUnitScore	0.1279682
67	14	MapSize	OwnRazingScore	0.1279682
68	14	NumberOfPossibleExpansions	MapSize	0.1279682
69	14	OwnRazingScore	GameTime	0.1279682
70	14	OwnBuildingScore	NumberOfMinerals	0.1279682
71	14	Race	OwnAirUnits	0.1279682
72	14	OpponentBuildingScore	OwnRazingScore	0.1279682
73	14	OpponentUnitScore	OpenSides	0.1279682
74	14	CurrentPresenceInShortestPath	Race	0.1279682
75	14	OwnAirUnits	CurrentPresenceInShortestPath	0.1279682

Iteration	Feature Count	Feature added	Feature removed	MAE
76	14	CurrentPresenceInShortestPath	OwnBuildingScore	0.1279682
77	14	OpenSides	CurrentPresenceInShortestPath	0.1279682
78	14	OwnUnitScore	OwnAirUnits	0.1279682
79	14	NumberOfMinerals	OpponentUnitScore	0.1279682
80	15	BestMinerals	N/A	0.1279682

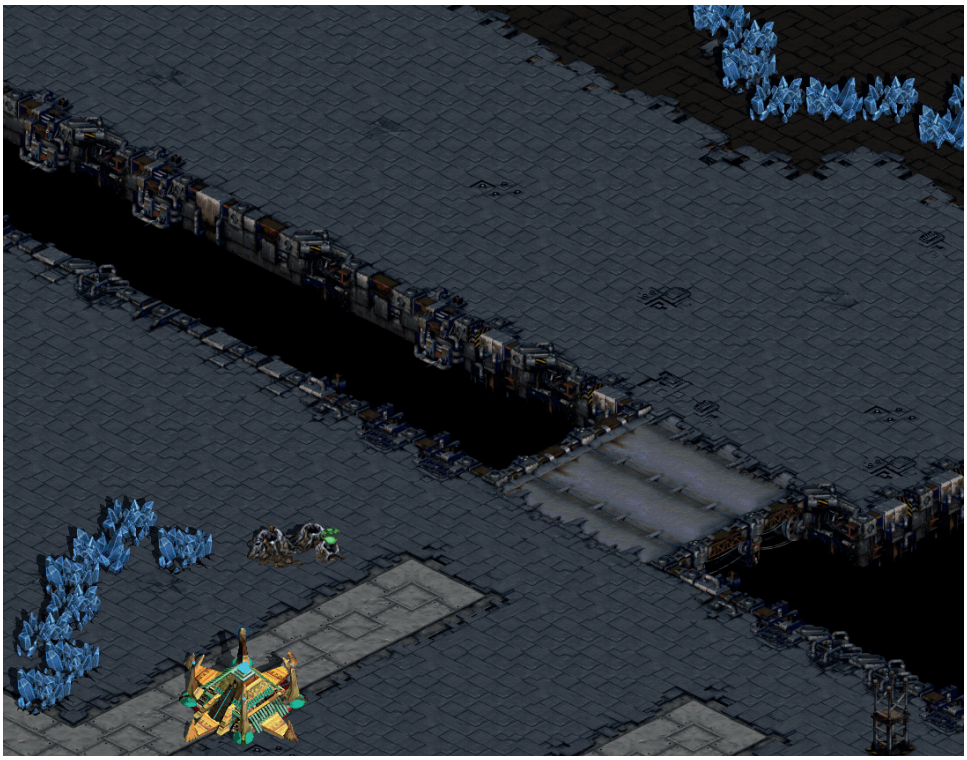
C

HIGH RESOLUTION COLOUR FIGURES

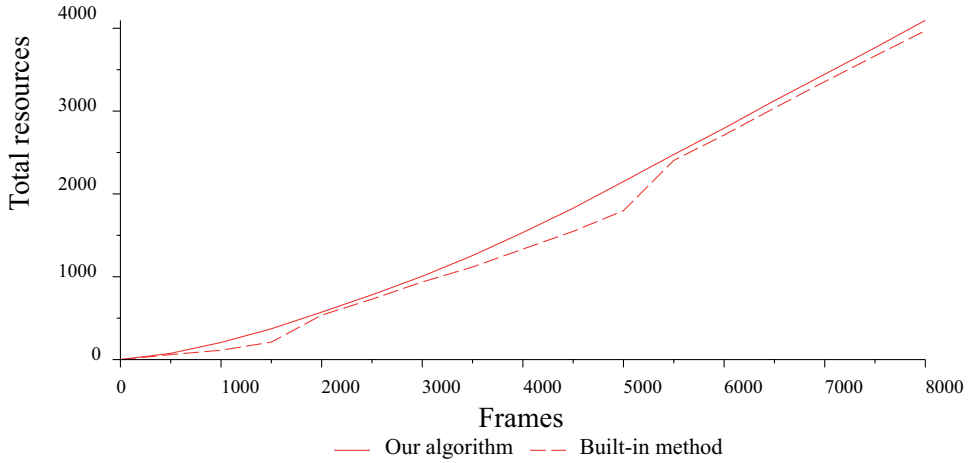
High resolution colour figure of Figure 1.1.

High resolution colour figure of Figure 1.2.

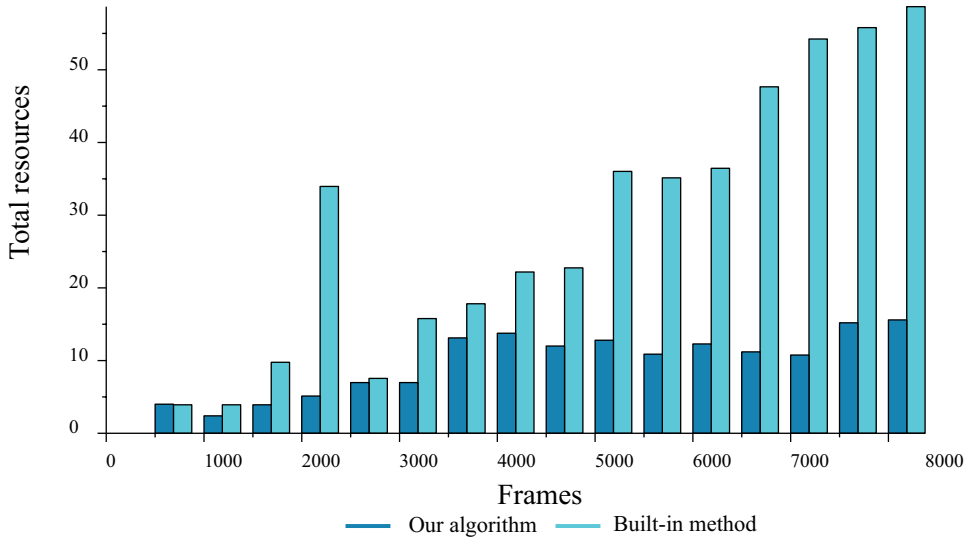
High resolution colour figure of Figure 1.3.



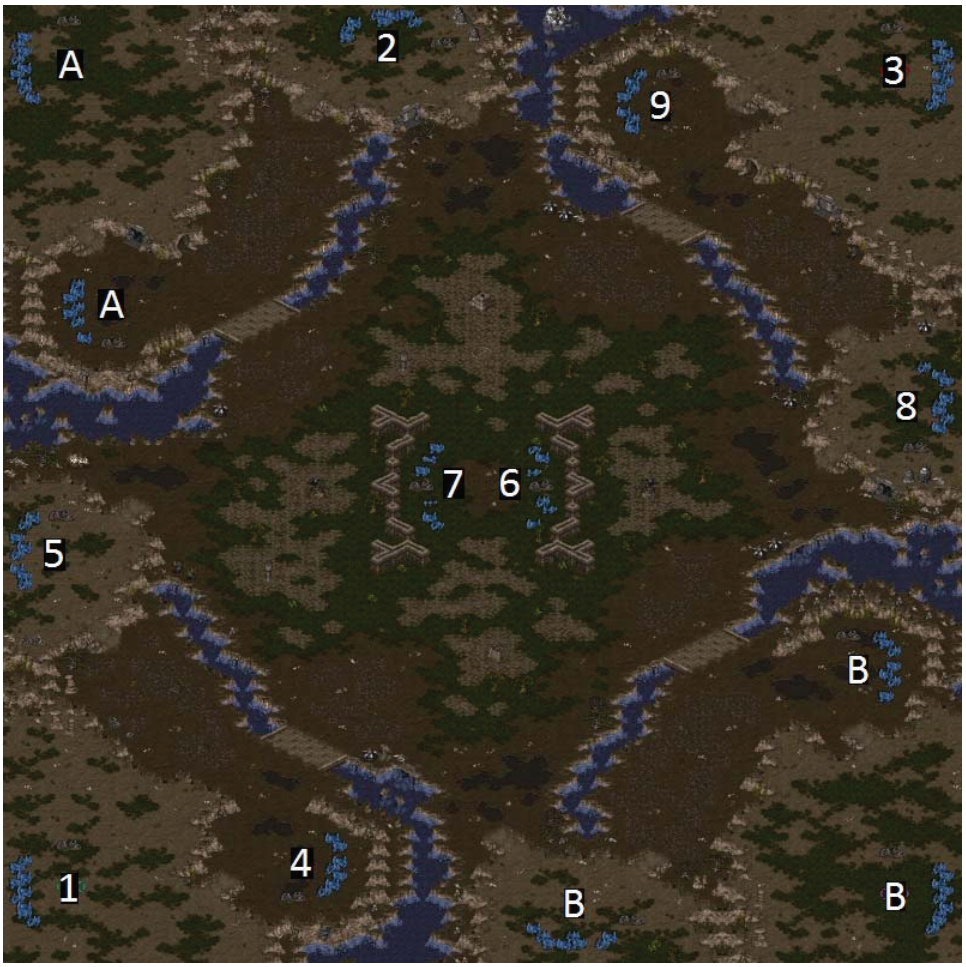
High resolution colour figure of Figure 3.1.



High resolution colour figure of Figure 6.1.



High resolution colour figure of Figure 6.2.



High resolution colour figure of Figure 13.1.

BIBLIOGRAPHY

[1] Michael Buro. Call for ai research in rts games. In *In Proceedings of the AAAI Workshop on AI in Games*, pages 139–141. AAAI Press, 2004.

[2] Michael Buro and Timothy Furtak. On the development of a free rts game engine. In *GameOn'NA Conference*, pages 23–27, 2005.

[3] Hei Chan, Alan Fern, Soumya Ray, Nick Wilson, and Chris Ventura. Online planning for resource production in real-time strategy games. 2007.

[4] VG Chartz. Software totals. http://www.vgchartz.com/worldtotals.php?genre=Strategy&sort=Total. [Online; accessed March 10th 2011].

[5] Dion Christensen, Henrik Ossipoff Hansen, Jorge Pablo Cordero Hernandez, Lasse Juul-Jensen, Kasper Kastaniegaard, and Yifeng Zeng. A data-driven approach for resource gathering in real-time strategy games. Accepted by ADMI-11, May 2011.

[6] Dion Bak Christensen, Henrik Ossipoff Hansen, Lasse Juul-Jensen, and Kasper Kastaniegaard. Efficient resource management in starcraft: Brood war. Technical report, December 2010. Technical report made as part of the DAT5 semester.

[7] Laurent Hyafil and Ronald L Rivest. Constructing optimal binary decision trees is np-complete. *Information Processing Letters*, 5(1), May 1976.

[8] A. Jain and D. Zongker. Feature selection: evaluation, application, and small sample performance. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 19(2):153–158, February 1997.

[9] Kelly Olsen. South korean gamers get a sneak peek at 'starcraft ii'. USA Today Online, http://www.usatoday.com/tech/gaming/2007-05-21-starcraft2-peek_N.htm, May 2007. [Online; accessed March 10th 2011].

[10] P. Pudil, P. Novovicova, and J. Kittler. Floating search methods in feature selection. *Pattern Recognition Letters*, 15(11):1119 – 1125, 1994.

[11] Steve Rabin. *Introduction to Game Development*. CHARLES RIVER MEDIA, INC., 10 Dower Avenue, Hingham, Massachusets, 1st edition, 2005. ISBN 1584503777.

[12] Thomas Randall, Peter Cowling, Roderick Baker, and Ping Jiang. Using neural networks for strategy selection in real-time strategy games.

[13] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009. ISBN 0136042597.

[14] Ben G. Weber and Michael Mateas. Case-based reasoning for build order in real-time strategy games. In *Preceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference*, 10/2009 2009.

[15] Sam Wintermute, Xu Joseph, and John E. Laird. Sorts: A human-level approach to real-time strategy ai. In *Proceedings of the Third Artificial Intelligence and Interactive Digital Entertainment Conference*. The AAAI Press, 2007.

[16] Georgios N. Yannakakis. Learning from preferences and selected multimodal features of players. In *Proceedings of the 2009 international conference on Multimodal interfaces*, ICMI-MLMI '09, pages 115–118, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-772-1.

[17] Georgios N. Yannakakis and John Hallam. Entertainment modeling through physiology in physical play. *Int. J. Hum.-Comput. Stud.*, 66:741–755, October 2008.

[18] Georgios N. Yannakakis, Manolis Maragoudakis, and John Hallam. Preference learning for cognitive modeling: a case study on entertainment preferences. *Trans. Sys. Man Cyber. Part A*, 39: 1165–1175, November 2009.

DATA-DRIVEN RESOURCE MANAGEMENT IN REAL-TIME STRATEGY GAMES

RESUMÉ

DION CHRISTENSEN, HENRIK OSSIPOFF HANSEN,
LASSE JUUL-JENSEN, KASPER KASTANIEGAARD

The thesis *Data-driven Resource Management in Real-time Strategy Games* introduces ways to streamline resource management in real-time strategy games. This is done using a data-driven approach, which has become possible due to the increased availability of replay data for this gaming genre. The thesis is split into two parts, each dealing with a distinct aspect of resource management: exploitative resource management and explorative resource management. The real-time strategy game *StarCraft: Brood War* by Blizzard Entertainment, is utilised as test bed for the work done in the thesis.

EXPLOITATIVE RESOURCE MANAGEMENT

Deals with the accumulation of resources and optimisation of the gathering process for available resources. Gathering efficiency is increased by adopting an algorithm for controlling agent behaviour. By utilising a simple queueing system, predictability and income is increased in comparison with the heuristic approach used by the test bed. The work on exploitative resource management is a summary of previous work by the authors.

EXPLORATIVE RESOURCE MANAGEMENT

Deals with the discovery or creation of new resource gathering opportunities. In traditional real-time strategy games, resource gathering opportunities are created by the construction of expansions. In order to efficiently expand to new locations in the game environment, some decision model is needed. The aim of the models proposed in this thesis, is to mimic expansion behaviour of human players, based on the decisions observed in replays. 28 potentially influential features are identified using expert knowledge. Sample data is extracted from replays, for use in training the decision models. Feature selection is utilised in order to identify the most significant features. A total of six different sets are tested using both artificial neural networks and decision trees. Subsets show performance similar to the full feature set, indicating low noise of the data. The decision models using the feature sets are able to predict base expansions in replay data with a hit rate of up to 64.43%. Based on the test results, it seems feasible that the approach may be utilised in a game scenario, for making sound choices of base expansion sites.