Adaptive Artificial Intelligence in
Real-Time Strategy Games

Jason Traish
School of Computing and Mathematics
Charles Sturt University

A thesis submitted for the degree of
Doctor of Philosophy

November 7, 2017
Abstract

Highly capable Artificial Intelligences (AI) have been created for board games such as Go and Chess. Players of these games can play against a computerised opponent at the equivalent skill level of grandmaster or better. However, such highly capable AI agents have not yet been developed for Real-Time Strategy (RTS) games.

RTS agents must address several challenging issues to demonstrate real player like skill. The first major issue is that RTS games play out in ‘real-time’. In the RTS game context, ‘real-time’ means that games do not have a rigid turn-based structure but play continuously with players taking actions at any time. The second major issue is that RTS games have a much larger game state-space than Go or Chess. This is because typical RTS games occur on large maps with terrain differentiations, and involve a large number of diverse units. They also involve many more actions such as resource collection, production management and different types of tactical actions. Finally, unlike Go or Chess, players in RTS games possess only incomplete game-state information. RTS AI is one of the next big AI challenges.

This thesis seeks to improve the quality of adaptive tactical RTS agents, by enabling them to respond more effectively to novel player actions. This is intended to improve both the challenge for skilled players and the value of the single-player experience.

The main focus of the thesis is on the issue of real-time decision making and the ability to adapt to changes within a game. The thesis provides a framework that allows an RTS AI to adapt to unknown scenarios without perceptible lag.
Acknowledgements

I would firstly like to thank my Supervisor Dr James Tulip for his invaluable assistance. I still remember my first time meeting Jim. It was my first day at CSU, and he was sternly announcing to my entire class that most of us would fail. His tone and manner were heavy, sincere and a little daunting. He took several minutes to emphasise that if we didn’t want to fail, that we should work hard, really hard! You could immediately tell some of us were intimidated and imagining failure. Instead for me, you set a fire. A burning hunger to conquer and own this challenge of failure. Which led me to become one of only three graduates from a class of forty.

Needless to say, the journey from that first day to this final day of submission has been long and challenging. But you were always there to support me, especially when I was too full of pride or of fear to ask.

I would also like to thank Dr Wayne Moore for incredible insights and directions on building this thesis. Without your countless hours of reviews and feedback, I would surely have been lost. You kept me going when all seemed lost, reminding me that not only could I complete this thesis, but that it would be a good one. Things like this that may appear to be small gestures, but let me emphasise they had a much bigger impact that I think you will ever recognise.

I would also like to thank Dr Michael Antolovich for his technical feedback. With these three wonderful supervisors directing me I was able to reach my goals.

I would also like to thank my parents Christopher Traish and Deborah Traish for their love and support throughout this long journey. And thank you to my whole family, for all of you always believing in me.

I’d also like to thank my close friend Rebecca Kaczmarczyk whose optimism and
sheer love of life showed me a different way to see the world. It is because of people like you that make long journeys like this feel all the shorter and more exciting.
Contents

Abstract iii
Table of Contents ix
List of Figures xiii
List of Tables xvi
Glossary xvii
Publications xxi
Declaration of Authorship xxiii

1 Introduction
  1.1 Modern Real-Time Strategy Game: Starcraft 2 . . . . . . . . . . . . . 3
  1.2 Real-Time Strategy Agent - Research Challenges . . . . . . . . . . . 4
  1.3 Research Proposal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
  1.4 Refined Research Problems . . . . . . . . . . . . . . . . . . . . . . . 8
  1.5 Original Contributions . . . . . . . . . . . . . . . . . . . . . . . . . 10
  1.6 Thesis Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2 Literature Review
  2.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
  2.2 RTS AI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
  2.3 Techniques Relevant to RTS Agents . . . . . . . . . . . . . . . . . . . 15
    2.3.1 Finite State Machine (FSM) . . . . . . . . . . . . . . . . . . . . 16
    2.3.2 Scripted Agents . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
    2.3.3 Reinforcement Learning . . . . . . . . . . . . . . . . . . . . . 17
    2.3.4 Bayesian Approaches . . . . . . . . . . . . . . . . . . . . . . . 19
    2.3.5 Genetic Algorithms (GA) . . . . . . . . . . . . . . . . . . . . 20
    2.3.6 Artificial Neural Networks (ANN) . . . . . . . . . . . . . . . 22
      2.3.6.1 NeuroEvolution of Augmenting Topologies (NEAT) 23
    2.3.7 Game State Search . . . . . . . . . . . . . . . . . . . . . . . . 25
    2.3.8 Supportive algorithms . . . . . . . . . . . . . . . . . . . . . . 27
      2.3.8.1 Clustering . . . . . . . . . . . . . . . . . . . . . . . 27
      2.3.8.2 Nearest Neighbour . . . . . . . . . . . . . . . . . . 27
2.3.8.3 Optimization .................................................. 28
2.3.8.4 Opponent Modelling ................................. 28
2.4 Data Retrieval ...................................................... 29
2.5 Case Based Reasoning (CBR) .............................. 31
2.6 Monte Carlo Tree Search (MCTS) ......................... 34
2.7 Path Finding ......................................................... 42
  2.7.1 A* Search ...................................................... 43
  2.7.2 Compressed Path Databases ............................. 43
  2.7.3 Cooperative Pathfinding .................................. 44
  2.7.4 SubGoal ......................................................... 44
  2.7.5 JPS .............................................................. 45
  2.7.6 JPS+ ............................................................ 47
  2.7.7 Other Pathfinding Related Algorithms ............ 47
    2.7.7.1 Influence Maps ...................................... 48
2.8 RTS Research Platforms ................................. 49
2.9 Summary and Synthesis .................................... 52
3 Data Extraction using Screen Capture .................. 55
  3.1 Introduction ..................................................... 55
  3.2 Overview of Screen Capture Technique ............... 57
  3.3 Avoiding Noisy Game Traces ....................... 61
    3.3.1 Experimental Method .................................. 61
    3.3.2 Extracting Starcraft 2 Build Orders ............. 61
    3.3.3 Results .................................................. 65
  3.4 Retrieving Inaccessible Data ......................... 67
  3.5 Discussion and Further Work ......................... 70
  3.6 Conclusion ................................................... 71
4 Path Finding .......................................................... 73
  4.1 Overview ........................................................ 73
  4.2 Implementations .............................................. 75
    4.2.1 Updating Invalidated Pre-Processed Data ....... 81
  4.3 Method .......................................................... 85
    4.3.1 Algorithm Summaries .................................. 86
    4.3.2 Static Map Experimental Results .................. 88
    4.3.3 Dynamic Map Experimental Results ............... 97
    4.3.4 Single Path Experiments (30 Obstacles 1 Path) .. 97
    4.3.5 Multiple Path Experiments (30 Obstacles 100 Paths) 102
    4.3.6 Memory Usage ........................................... 107
    4.3.7 Pre-Processing Time ................................. 108
  4.4 Discussion ........................................................ 110
  4.5 Conclusion .................................................... 112
5 BLJPS Improvements ........................................... 113
  5.1 Overview ........................................................ 113
    5.1.1 BLJPS2 .................................................... 114
List of Figures

2.1 MCTS Algorithm. Figure from (Browne et al., 2012) . . . . . . . . . 35
2.2 Jump point search natural and forced neighbours. Black cells repre-
sent blocked cells or obstacles. Striped cells represent forced
neighbours. Clear white cells represent open or unblocked cells.
Grey cells are not considered during this directional expansion. . . 46

3.1 Sample screen capture . . . . . . . . . . . . . . . . . . . . . . . . . . 63
3.2 Player 1 - Matching production templates . . . . . . . . . . . . . . . 63
3.3 Player 2 - Matching production templates . . . . . . . . . . . . . . . 63
3.4 Digit with noise . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
3.5 Pre-labeled Digit (Matching Template) . . . . . . . . . . . . . . . . 63
3.6 Progress bar . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
3.7 Sample game time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
3.8 Dota 2 Hero Selection Screen . . . . . . . . . . . . . . . . . . . . . . 68
3.9 Dota 2 Predicted Game Balance . . . . . . . . . . . . . . . . . . . . . 70

4.1 Boundary lookup example on a 13x9 uniform grid. Blacked cells
K6-K9 are obstacle boundaries. P1 is the starting point. P2 and P3
represent jump points. P4 is a forced neighbour of P3. . . . . . . . . 78
4.2 A) Partial Update (PU) displaying invalidated areas after adding
two obstacles. B) New obstacle area. C) Cardinal direction invali-
dated area. D) Diagonal stepping invalidated area. White areas are
unblocked and grey shaded areas are blocked. . . . . . . . . . . . . 84
4.3 Static Map DAO (Fast). Each algorithm is represented as three lines
of the same colour. Average values are indicated by the patterned
middle line. Confidence intervals of +-95% are shown by dotted
lines on either side of the central line. . . . . . . . . . . . . . . . . . 89
4.4 Static Map DAO (Slow). Each algorithm is represented as three
lines of the same colour. Average values are indicated by the pat-
terned middle line. Confidence intervals of +-95% are shown by
dotted lines on either side of the central line. . . . . . . . . . . . . . 90
4.5 Static Map BG (Fast). Each algorithm is represented as three lines
of the same colour. Average values are indicated by the patterned
middle line. Confidence intervals of +-95% are shown by dotted
lines on either side of the central line. . . . . . . . . . . . . . . . . . 91
4.6 Static Map BG (Slow). Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line. ........................................ 92

4.7 Static Map AD (Fast). Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line. ........................................ 93

4.8 Static Map AD (Slow). Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line. ........................................ 94

4.9 Static Map Rooms (Fast). Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line. ........................................ 95

4.10 Static Map Rooms (Slow). Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line. ........................................ 96

4.11 Dynamic Map DAO, 1 Path. Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line. ........................................ 98

4.12 Dynamic Map BG, 1 Path. Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line. ........................................ 99

4.13 Dynamic Map AD, 1 Path. Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line. ........................................ 100

4.14 Dynamic Map Rooms, 1 Path. Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line. ........................................ 101

4.15 Dynamic Map DAO, 100 Paths. Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line. ........................................ 103

4.16 Dynamic Map BG, 100 Paths. Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line. ........................................ 104
4.17 Dynamic Map AD, 100 Paths. Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +/-95% are shown by dotted lines on either side of the central line. 105

4.18 Dynamic Map Rooms, 100 Paths. Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +/-95% are shown by dotted lines on either side of the central line. 106

5.1 DAO Experiment: Average speedup relative to A* against path length. Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +/-95% are shown by dotted lines on either side of the central line. 125

6.1 Search and recall agent process 133
6.2 Experimental Setup 142

7.1 Scenario 1. Red: MCTS, Green: Opponent. Simple Mirror Match. 3Z & 3D: 3 Zealots and 3 Dragoons 168
7.2 Scenario 2. Red: MCTS, Green: Opponent. Simple Mirror Match. 8D: 8 Dragoons vs. 8 Dragoons 168
7.4 Scenario 4. Red: MCTS, Green: Opponent. V & T: 16 Vultures and 4 Tanks vs. 15 Zealots and 9 Dragoons. Opponent 40% bias 169
7.5 Scenario 5. Red: MCTS, Green: Opponent. 1B: 28 Marines vs. 79 Fast Zerglings on Terrain with a single bottleneck. Opponent 41% bias 170
7.6 Scenario 6. Red: MCTS, Green: Opponent. 2B: 48 Zerglings vs. 18 Marines on Terrain with two bottlenecks. Allied 33% bias but enemy has significant positioning advantage. 170
List of Tables

3.1 Data Retrieval - Error Rates ........................................ 66
3.2 Example Game Trace .................................................. 66

4.1 Boundary Lookup Table – Used to determine the nearest obstacle boundary along either the horizontal or vertical axis. ............... 78
4.2 Maps used in experiments. DAO maps vary in size, ranging from the smallest and largest map sizes given. Each map is associated with a varying number of paths, the average for map each set is given. ......................................................... 85
4.3 Summarised results for Figures 4.11-4.14. Speedup rank best to worst 1-6 for map and speedup relative to A*. Each result is given in the form X (Y,Z) where X is the rank, Y is the average and Z is the standard deviation of the speedup. .................. 99
4.4 Summarised results for figures 4.15-4.18. Speedup rank best to worst 1-6 for map and speedup relative to A*. Each result is given in the form X (Y,Z) where X is the rank, Y is the average and Z is the standard deviation of the speedup. .................. 106
4.5 Pre-processing memory overheads for algorithms tested in static maps. Memory overheads are listed in kilobytes. Each result is given in the form X (Y) where X is the average and Y is the standard deviation. .................. 108
4.6 Query memory overheads for algorithms tested in static maps (Inclusive of pre-processing). Each result is given in the form X (Y) where X is the average and Y is the standard deviation. Overheads are listed in average memory use of kilobytes. .................. 108
4.7 Average memory overhead per map cell in bytes as tested in static maps. Each result is given in the form X (Y) where X is the average and Y is the standard deviation. Bytes = Memory Overhead/(Map_Height * Map_Width). .................. 109
4.8 Number of elements within the boundary lookup table after pre-processing. Each element is defined as an unsigned short (2 bytes in size). Each result is given in the form X (Y) where X is the average and Y is the standard deviation. .................. 109
4.9 Average number of elements within the boundary lookup table against map size after pre-processing. .................. 109
4.10 Time to pre-process a map in seconds. Each result is given in the form $X (Y)$ where $X$ is the average and $Y$ is the standard deviation.

5.1 Maps used in experiments. DAO maps vary in size, ranging from the smallest and largest map sizes given. Each map is associated with a varying number of paths, the average for map each set is given.

5.2 Experimental Results: BLJPS5 vs SubGoal. Bold represents the best result. Each result is given in the form $X (Y)$ where $X$ is the average and $Y$ is the standard deviation.

5.3 Experimental Results: BLJPS variations. Bold represents the best result. Each result is given in the form $X (Y)$ where $X$ is the average and $Y$ is the standard deviation.

6.1 Behaviour Descriptions

6.2 Success rates of different agent configurations against ‘Attack Closest’ scripted agent in scenarios A and B and against the default Brood War AI in C and D. BAS: Basic simulator (no collisions or terrain, no kiting), COM: complex simulator (collisions and terrain plus kiting). Search: search only component. S&R: Search and Recall components. (Churchill and Buro, 2012) ABCD search results against a similar scripted ‘Attack Closest’ agent is shown for comparison in Column 2. The success of a standard scripted agent with a primary action of Attack Closest and a secondary action of Attack Wounded is shown (Attack Closest) in Column 7. The success of an agent based on Kiting only is shown in Column 8.

7.1 Grouping Actions

7.2 General Action Descriptions

7.3 Specific Action Description

7.4 Experimental Results. Highest results are highlighted in bold.
## Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A*</td>
<td>Basic path finding algorithm</td>
</tr>
<tr>
<td>ABCD</td>
<td>Alpha-Beta Considering Durations</td>
</tr>
<tr>
<td>AD</td>
<td>Adaptive Depth - Map type from GPPC</td>
</tr>
<tr>
<td>ANN</td>
<td>Artificial Neural Network</td>
</tr>
<tr>
<td>BAS</td>
<td>Basic simulator used in Chapter 6</td>
</tr>
<tr>
<td>Brood War</td>
<td>RTS video game developed by Blizzard</td>
</tr>
<tr>
<td>CBR</td>
<td>Cased Based Reasoning</td>
</tr>
<tr>
<td>BG</td>
<td>Baldur’s Gate - Map type from GPPC</td>
</tr>
<tr>
<td>BLJPS</td>
<td>Boundary Lookup Jump Point Search</td>
</tr>
<tr>
<td>Build Order</td>
<td>The sequence in which units and buildings are created</td>
</tr>
<tr>
<td>BWAPI</td>
<td>Brood War Application Programming Interface</td>
</tr>
<tr>
<td>COM</td>
<td>Complex simulator used in Chapter 6</td>
</tr>
<tr>
<td>CS</td>
<td>Concurrent Search simulator component</td>
</tr>
<tr>
<td>CIGAR</td>
<td>Case Injected Genetic AlgoRithm</td>
</tr>
<tr>
<td>CPD</td>
<td>Compressed Path Databases - Pathfinding algorithm</td>
</tr>
<tr>
<td>DAO</td>
<td>Dragon Age Origins - Map type from GPPC</td>
</tr>
<tr>
<td>DG</td>
<td>Dynamic Granularity 7</td>
</tr>
<tr>
<td>Dire</td>
<td>The team name of 5 heroes in the game of Dota 2</td>
</tr>
<tr>
<td>Dota 2</td>
<td>Defense Of The Ancients 2</td>
</tr>
<tr>
<td>Dragoon</td>
<td>Brood War Protoss ranged unit</td>
</tr>
<tr>
<td>DRL</td>
<td>Deep Reinforcement Learning</td>
</tr>
<tr>
<td>EA</td>
<td>Evolutionary Algorithm</td>
</tr>
<tr>
<td>FFANN</td>
<td>Feed Forward Artificial Neural Network</td>
</tr>
<tr>
<td>FPS</td>
<td>First-&amp;Person &amp; Shooter</td>
</tr>
<tr>
<td>Fog of War</td>
<td>An RTS game feature that allows players to only see areas of the map that they have units within</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>FU</td>
<td>Flush Update - Method for updating dynamic map information</td>
</tr>
<tr>
<td>HCDPS</td>
<td>Hill Climbing Dynamic Programming Search</td>
</tr>
<tr>
<td>HPA*</td>
<td>Hierarchical Path finding A*</td>
</tr>
<tr>
<td>GA</td>
<td>Genetic Algorithm</td>
</tr>
<tr>
<td>GGP</td>
<td>General Game Playing</td>
</tr>
<tr>
<td>GP</td>
<td>Genetic Programming</td>
</tr>
<tr>
<td>GPPC</td>
<td>Grid-Based Path Planning Competition</td>
</tr>
<tr>
<td>GOAP</td>
<td>Goal Oriented Action Planning</td>
</tr>
<tr>
<td>JPS</td>
<td>Jump Point Search: A pathfinding algorithm</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>JPS+</td>
<td>Jump Point Search+: A pathfinding algorithm</td>
</tr>
<tr>
<td>Kite</td>
<td>Is a tactical behaviour where units that are relatively faster with longer range attacks constantly move away from their target and attacking them</td>
</tr>
<tr>
<td>IM</td>
<td>Influence Map</td>
</tr>
<tr>
<td>IPD</td>
<td>Iterative Prisoner Dilemma</td>
</tr>
<tr>
<td>LTD</td>
<td>Life-Time Damage</td>
</tr>
<tr>
<td>Macro</td>
<td>Relates to an RTS player’s strategic decisions</td>
</tr>
<tr>
<td>Marine</td>
<td>Brood War Terran ranged unit</td>
</tr>
<tr>
<td>MCTS</td>
<td>Monte Carlo Tree Search</td>
</tr>
<tr>
<td>Micro</td>
<td>Relates to an RTS player’s tactical decisions</td>
</tr>
<tr>
<td>MM-NEAT</td>
<td>Modular Multi-objective - NeuroEvolution of Augmenting Topologies</td>
</tr>
<tr>
<td>MOBA</td>
<td>Multiplayer Online Battle Arena</td>
</tr>
<tr>
<td>NEAT</td>
<td>NeuroEvolution of Augmenting Topologies</td>
</tr>
<tr>
<td>NovelAStar</td>
<td>NovelAStar - A pathfinding algorithm</td>
</tr>
<tr>
<td>ORTS</td>
<td>Open Real Time Strategy</td>
</tr>
<tr>
<td>PCA</td>
<td>Principle Component Analysis</td>
</tr>
<tr>
<td>Protoss</td>
<td>One of three races in Starcraft Brood War</td>
</tr>
<tr>
<td>PPQ</td>
<td>Pseudo Priority Queues - Pathfinding algorithm</td>
</tr>
<tr>
<td>PRA*</td>
<td>Partial -Refinement A*</td>
</tr>
<tr>
<td>PU</td>
<td>Partial Update - Method for updating dynamic map information</td>
</tr>
<tr>
<td>PvP</td>
<td>Player Versus Player</td>
</tr>
<tr>
<td>Radiant</td>
<td>The team name of 5 heroes in the game of Dota 2</td>
</tr>
<tr>
<td>RGB</td>
<td>Red Green Blue</td>
</tr>
<tr>
<td>RL</td>
<td>Response Library</td>
</tr>
<tr>
<td>Rooms</td>
<td>Rooms - Map type from GPPC</td>
</tr>
<tr>
<td>RPS</td>
<td>Response-Playback Simulator</td>
</tr>
<tr>
<td>RTS</td>
<td>Real-Time Strategy</td>
</tr>
<tr>
<td>S&amp;R</td>
<td>Search &amp; Recall</td>
</tr>
<tr>
<td>Sarsa</td>
<td>State &amp; Action &amp; reward &amp; state &amp; action</td>
</tr>
<tr>
<td>Siege Tank</td>
<td>Brood War Long ranged Terran unit</td>
</tr>
<tr>
<td>SOM</td>
<td>Self-Organizing Maps</td>
</tr>
<tr>
<td>Sparcraft</td>
<td>A simulator for Stacraft Brood War</td>
</tr>
<tr>
<td>Starcraft 2</td>
<td>A commercial RTS game</td>
</tr>
<tr>
<td>Starcraft: Broodwar</td>
<td>A commercial RTS game</td>
</tr>
<tr>
<td></td>
<td>Also known as Starcraft 1 or Brood War.</td>
</tr>
<tr>
<td>SubGoal</td>
<td>A grid-based pathfinding algorithm</td>
</tr>
<tr>
<td>Terran</td>
<td>One of three races in Starcraft Brood War</td>
</tr>
<tr>
<td>TRA*</td>
<td>Triangulation Reduction A*</td>
</tr>
<tr>
<td>UCB</td>
<td>Upper Confidence Bounds</td>
</tr>
<tr>
<td>UCT</td>
<td>Upper Confidence Bound 1 applied to Trees</td>
</tr>
<tr>
<td>Vulture</td>
<td>Brood War Fast ranged Terran unit</td>
</tr>
<tr>
<td>Warcraft 2</td>
<td>A commercial RTS game</td>
</tr>
<tr>
<td><strong>Warcraft 3</strong></td>
<td>A commerical RTS game</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td><strong>Wargus</strong></td>
<td>An open source clone of the commerical RTS game</td>
</tr>
<tr>
<td></td>
<td>Warcraft 2</td>
</tr>
<tr>
<td><strong>Zealot</strong></td>
<td>Brood War Protoss melee unit</td>
</tr>
<tr>
<td><strong>Zerg</strong></td>
<td>One of three races in Starcraft Brood War</td>
</tr>
<tr>
<td><strong>Zergling</strong></td>
<td>Brood War Zerg melee unit</td>
</tr>
</tbody>
</table>
Publications


Declaration of Authorship

I, Jason TRAISH, declare that this thesis titled, “Adaptive Artificial Intelligence in Real-Time Strategy Games” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.

- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.

- Where I have consulted the published work of others, this is always clearly attributed.

- Where I have quoted from the work of others, the source is always given.
  With the exception of such quotations, this thesis is entirely my own work.

- I have acknowledged all main sources of help.

- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date: November 7, 2017
Chapter 1

Introduction

Real-Time Strategy (RTS) games are a sub-genre of strategy games played most commonly on a personal computer. Typically, RTS games involve building and commanding an army of diverse units in an attempt to conquer an enemy. There a diverse number of RTS games with similar game elements. RTS games like Brood War, Starcraft 2 and Command and Conquer are most typically played in matches between two human players in a one on one “Player versus Player” (PvP) match. A player must perform a number of actions in order to defeat their opponent. They need to collect and spend resources, train units and then coordinate them in attacking the enemy, meanwhile constructing buildings allowing access to more advanced units. The maps of RTS games are also typically filled with a "fog of war" which only allows a player to see a small area around the units and buildings they control. This allows players to deceive their opponents which in turn encourages diverse offensive strategies and defensive strategies. In essence, these games require a player to collect resources to train an army to destroy their opponent. A player must also make all of these decisions and execute them in real-time at the same time as their opponent.

There are many types of RTS games that differ in a number of aspects. For example, in Supreme Commander each player is given critical ‘command’ unit and
as in Chess, if that unit is killed, the player loses. The resources within Z: Steel Soldiers are allocated according to the territory of the map the player controls in contrast to requiring the player to assign units to gather them.

In contrast to turn based board games such as Go and Chess, an RTS AI needs to make decisions within a significantly more complex and constrained environment. For example, an RTS AI must manage an economy, as well as create and direct armies of diverse types of units. It has to perform on various maps of differing sizes and terrains with incomplete game state information (Ontanón et al., 2013; Buro and Furtak, 2004; Lara-Cabrera, Cotta, and Fernández-Leiva, 2013). It also has to make fast decisions due to the concurrent way in which players participate in the game.

These are some of the core differences that make RTS AI difficult to develop relative to board game AI for Chess and Go. One of the most challenging issues for RTS AI is working within an exceptionally large state space. The branching factor of Chess was estimated to be around 35 and GO with about 180. In contrast, a typical RTS game such as Starcraft was estimated to have a branching factor of $10^{50}$ or higher (Ontañón, 2016). The number of unique decisions an RTS AI can make grows substantially over the course of a match due to the growing number and diversity of units it controls. Additionally, an RTS AI is expected to respond quickly even though the complexity of the state space is large.

The research proposal presented in this thesis focuses on two areas; creating an RTS AI that can adapt to new situations within a complex game state space, and that make decisions without perceptible lag. It also outlines the challenges involved in integrating the solutions to these two problems.
1.1 Modern Real-Time Strategy Game: Starcraft 2

Currently, one of the most popular commercial RTS games is Starcraft 2 (Klappenbach, 2017). Starcraft 2 (SC2) allows players to play against either other humans or a computer agent. The games involving conflict with the computer agent come in two modes. The first mode involves a virtual campaign against alien races. Each match continues the unfolding story of a campaign. The second mode is similar to Player vs. Player (PvP) mode, but instead, the player is pitted against an AI. This second mode is the focus of this thesis.

Players can play against the AI in matches similarly to how they would play PvP matches. However, matches against these AI suffer from a number of weaknesses that make them uninteresting to play for experienced players. Current commercial RTS AI do not modify their behaviours in response to unpredictable player actions. That is, even though they may exhibit complex state-based behaviours, once an exploit is discovered the challenge of the AI is destroyed.

RTS AI are commonly given an unfair advantage by receiving more map vision and resources than the human player. This bias also changes the game dynamic from what would be expected in a standard match between two players, and leads to uninteresting and unrealistic games. The experiences gained from these biased single-player matches cannot be applied directly to human PvP matches.

Weaknesses within RTS AI may be intentionally implemented as a design decision as indicated by the lead AI developer for Civilization Soren Johnsen (Soren Johnson, 2008). He explains that developers of commercial game AI have a responsibility to create a fun environment for players, and that players that lose to aggressively skilled AI may not enjoy the game. He also explains that many of
Chapter 1. Introduction

the research challenges in AI are outside of the scope of what commercial companies are trying to achieve. allowing game AIs to cheat requires less development resources than creating a non-cheating AI.

Although commercial game AI developers may wish to create a fun user experience, there is definitely room to improve the challenge of AI within the RTS genre and specifically in games like Starcraft 2. More research and understanding into how to create more challenging RTS AI would allow commercial companies to improve their AI implementations without investing greater development resources.

1.2 Real-Time Strategy Agent - Research Challenges

The challenge is to develop a highly capable RTS AI. It must be able to play matches on different maps that give advantages and disadvantages to different strategies. It must play with uncertain information which means that it cannot see the complete game state, which is visible in games such as Go and Chess. Instead, the agent must actively seek out information on their opponent. The agent must be able to manage a number of bases and its economy to build an army of diverse units. It needs to decide the composition and timing of army creation in a way that best suits a strategy to defeat an opponent. It then has to direct its army to defend and attack in order to defeat its opponent. An advanced AI should also be able to model its opponent in order to predict and exploit the opponent’s strategies (Robertson and Watson, 2014).

A complete agent would be expected to perform all of the above actions and more. It also needs to be able to perform all these tasks without perceptible lag.
Each point is a complex problem within itself, and this thesis focuses on two problem areas.

The research in this thesis focuses on giving an RTS AI the ability to make good decisions without perceptible lag. This will give the agent multiple benefits:

1. A human playing against the AI would not see sluggish responses that would tarnish the game experience.

2. Minimising the latency of an agent’s reactions would improve the quality of its responses in very dynamic situations. For example, if the agent detects an army was approaching an ambush, each second of delay in making a retreat decision would lead to greater than necessary losses.

3. Within an ongoing battle an agent could constantly adjust orders to individual units in order to maximise their effectiveness. Given the complexity of the state space, it is unlikely that a preconceived battle-plan will occur as predicted. It is important to adapt as quickly as possible to changing circumstances.

4. If the agent takes longer than a single frame of the game to make a decision then the information it used for that decision is old. For example, if the agent takes 2 seconds to make a decision all the units under its control and the opponent’s units have moved position, taken and given damage. Thus, even if the decision was optimal 2 seconds earlier, it is likely that in the current situation it has become sub-optimal.

The second focus is on dealing with the complex state space of RTS games. At the time of this thesis, a major advance, in Go AI was made with the AI winning against a world champion for the first time (Silver et al., 2016). It took many years of research to develop an AI capable of playing at such a high skill level within the large state space that Go offers. RTS games have an even larger state space
and operate in real-time. Thus, RTS games such as Starcraft offer an important research challenge.

1.3 Research Proposal

The author’s suggestion to create an RTS AI with real-time response capabilities and the capacity to operate in a high dimensional game space is to use a fusion of Cased Based Reasoning (CBR) and Search.

CBR is an approach that generalises past experiences to make decisions in the future. CBR is primarily based on having many high quality recorded experiences to serve as templates for the AI decisions. CBR reduces the decision-making process to a database query for an appropriate response to a given game state. It works as a memory for the AI, decreasing the latency of the decision-making process in most contexts.

The problem with CBR is that as it is currently applied, it requires access to an extensive collection of high quality recorded games previously played by humans, to populate the database with successful strategies. Such collections of recorded games are not available for new games. CBR also flounders when a situation not recorded in the database is encountered. The approach used to enable an AI to deal with novel situations is Search. Search works by trialling various choices and evaluating the consequences of those choices. RTS games have an exceptionally large action space that needs to be explored, making effective solutions hard to find within the very short time frames required for decision making within a game. Currently, these issues are addressed using various sophisticated techniques to make search more effective or to simplify action space, but also by using
extremely simplified simulators to evaluate choices during exploration of action space.

The problem with search as it is currently practiced is that the action space of a typical RTS game is so overwhelmingly large that it is extremely difficult to conduct an effective search, particularly within the time constraints imposed by a game loop.

The novel idea this thesis seeks to explore is whether a fusion of both CBR and Search could resolve both issues. This fusion of techniques is referred to as Search and Recall.

The proposal is that a CBR database could be populated with the results of a Search process. This allows the Search process to be decoupled from the game loop, potentially allowing far more exhaustive searches using far less simplified simulators, without affecting game performance. Conversely, it would allow CBR databases to evolve and grow to include novel situations. The fusion of techniques could allow RTS AI to make real-time decisions as if they were using a CBR database, while the responses recorded in that database could evolve as the result of Search. The fundamental research question is:

"How can adaptive RTS AI be improved while retaining the ability to make decisions without perceptible lag?"

From this fundamental question, a number of ancillary research questions arise. These research questions relate to the following issues involved in the process of developing and an adaptive AI.

- Data acquisition
- Adaptive capability
- Decision making without perceptible lag
Chapter 1. Introduction

- Accuracy of the simulation environment

The following sections expand on the problems associated with each of these issues and provide the research directions of this thesis.

1.4 Refined Research Problems

Based on the constraints of RTS games, the major research problem can be broken down into the following sub-problems:

"How can game state data be collected from RTS games with either noisy or no game log data?"

Data acquisition on the progress of a game is a requirement for AI development. Many games provide game progression logs in the form of game replay files. However, the log data from games such as Starcraft 2 contain noise. Commonly, these game logs are intended to be interpreted through the originating game and not to be used by third parties. Such logs may contain every command issued during the game, irrespective of whether it was ever enacted within the match, resulting in a very ‘noisy’ log. A method of reducing the noise from these game logs is required. On the other hand, some games provide no game logs and game replays are presented purely as a video stream. A method is required to retrieve game state progression from this data. A process of retrieving clean data directly from games is described in Chapter 3.

"How can adaptive tactical RTS AI be improved to more effectively find solutions to complex, previously unseen scenarios?"

A highly capable RTS AI requires that it adapt to unencountered circumstances. Players should not be able to repeatedly reproduce the same matches that lead to victory over the AI, and the AI should be able to cope with novel player strategies.
1.4. Refined Research Problems

Such an adaptive capability would address the issue of AI matches becoming boring and remove the need for them to cheat to appear challenging. This adaptive behaviour could take place within the game or between games.

Complex tactical scenarios involve large armies with different unit types, and also involve collisions between units and interaction with terrain. In order to deal with such scenarios, an important issue is what aspects of the real game need to be included in the simulator in order for the results of the simulation to be useful within the real game. Simulation fidelity with the game is a major issue affecting the quality of responses generated by Search based RTS AI. These issues are explored in Chapters 6 and 7.

"How can an effective adaptive tactical RTS AI respond successfully with little or no perceptual lag?"

RTS AI are required to work within the real-time constraints of the game. It is unacceptable that the AI should reduce the pace of the game. While the agent may take multiple frames to decide on an action this time should be minimised. Reducing the decision time will allow the agent to respond more quickly in highly dynamic situations. This will affect its performance and the perception of the agent from the user’s perspective. An agent that takes too long to make decisions will be exploited by other players. This issue is primarily addressed in Chapter 6. It is indirectly addressed through the work on pathfinding in Chapters 4, 5 and the work on exploring complex state spaces more efficiently in Chapter 7.

"How can a simulator’s pathfinding be improved?"

Pathfinding (that is: dealing with collisions between units, and terrain) is an important aspect of RTS games, commonly neglected in Search simulators due to its computational cost. Pathfinding is a major processing bottleneck in high fidelity simulations. Improving the speed of pathfinding on RTS maps greatly reduces
Chapter 1. Introduction

the time spent in each simulation, thus allowing Search to examine more possible action combinations and sequences. Pathfinding in a simulation environment should execute as quickly as possible in order to improve throughput. Issues associated with this are addressed in Chapters 4 and 5.

1.5 Original Contributions

The key contributions of this thesis can be listed as follows:

- A novel data logging framework for extracting game progression data from applications using Screen Capture has been developed.

- A number of optimisations to pathfinding algorithms that provide faster search times within simulations have been developed.

- A novel approach to combining Case Based Reasoning with the results of Search has been developed. This approach decouples Search from the time constraints of the game loop, allows the game AI to respond without perceptual lag, and allows the CBR database to include responses to novel situation over time. This is referred to as ‘Search and Recall’.

- A novel approach dealing with the issue of search space complexity when applying Monte-Carlo Tree Search to solve complex tactical scenarios has been developed. This approach is referred to as Dynamic Granularity and it optimises the number and makeup of unit groupings used in a simulation, favouring low numbers of groups, and resulting in more effective searches.
1.6 Thesis Outline

Chapter 2 of this thesis contains a literature review beginning with an exploration of General Machine Learning techniques.

Chapter 3 presents a screen capture data extraction framework. This framework was developed to assist in gathering data from applications that either don’t provide a way to access their data (replay files) or in which the replay data is unreliable. The framework demonstrates successful data collection from two different games.

Chapter 4 presents an optimisation of the Jump Point Search pathfinding algorithm Boundary Lookup Jump Point Search (BLJPS). This optimisation further minimises search times in dynamic environments and thus increases search throughput for small time windows. Detailed analysis of the pathfinding algorithm was performed, and Chapter 5 presents extensions to the BLJPS algorithm. Each iteration offers lower search times by increasing pre-processing time and memory usage. The work presented in these chapters supports the high-fidelity simulators used in Chapters 6 and 7.

Chapter 5 presents extensions to the BLJPS algorithm. Each iteration offers lower search times by increasing pre-processing time and memory usage.

Chapter 6 presents the Search and Recall framework. This chapter presents in detail the framework that uses both Search and Case Based Reasoning. It tests the framework against a previously published set of experiments and demonstrates improvement over earlier work. It established the basic principles and architecture for building a system that leverages search and memory for RTS AI.

Chapter 7 presents a new approach to addressing action space complexity in
Monte-Carlo Tree Search based on optimising the effective number of units involved in Search. The technique is referred to as Monte Carlo Tree Search using Dynamic Granularity (MCTS-DG). This chapter also addresses the issues of simulation fidelity encountered in the development of Search and Recall.

Chapter 8 concludes with a re-examination of the research problems presented in this Chapter (Chapter 1) and a plan for future research.
Chapter 2

Literature Review

2.1 Overview

This chapter begins by outlining a number of different RTS AI challenges and general research areas. Next, a review of these areas within the literature is given with specific regards to research work relevant to RTS AI. The review then expands on the techniques that correspond to the main research questions defined in Chapter 1.

Other research works potentially related to addressing these problems are recognised and summarised.

2.2 RTS AI

RTS AI are required to possess many competencies to fully function in an RTS environment. For the following discussion, Starcraft 2 and Starcraft: Brood War will be considered as examples of the RTS genre. The challenges for these two games are broadly categorised below.

1. Reason from partial game state observations (Fog of war).
2. Control large armies containing hundreds of units of diverse compositions in real-time.

3. Plan, execute and adapt strategies based on multiple factors such as map, opponent, race, game state and observations.

4. Reason with non-deterministic gameplay (E.g. Varying chance to hit units on higher terrain). This affects which machine learning models can be used to represent the agent, as actions must be capable of probabilistic outcomes which result in differing transitions between game states.

5. Manage collaboration of multiple systems (Economy, research, production, tactical and micro decisions).

The major challenge for RTS AI is dealing with the large state and actions spaces of the games. The state space of an RTS game refers to the different possible states the game can be in. The game state of RTS games is relatively complex compared to games like chess. The game state of Chess is defined by where each piece on the board is located and which player’s turn it is. The game state of an RTS game such as Brood War is defined by the number, position, health, attack cooldown, energy, type, movement velocity, movement acceleration and facing direction of each unit on the map. The game state would also take into account the status of resources on the map.

Action space refers to the different actions for each unit that can be chosen by a player. Within Brood War there are many actions that can be taken. Actions include the turning and movement of units in multiple directions and attacking other units. It also considers unit specific actions such as workers choosing which resource they gather and where to return their harvest. Workers can also construct structures, which in turn can train more units, research upgrades, or unlock the ability to construct more advanced structures. Furthermore, some units
have special abilities that only they can use on a target unit or area.

From a practical standpoint, there are additional challenges in implementing an AI for RTS games. These include:

1. Accessing data for analysis requires additional work for games like Starcraft 2 where the replay data is noisy.

2. A simulator is required for in-game planning and search in order to deal with previously unencountered scenarios. A simulator also allows an agent to perform look ahead reasoning to test possible effects of alternate decisions.

3. Complex games such as Brood War are difficult to simulate exactly. This is especially in regard to mechanics such as pathfinding, where the exact algorithm is unknown and can only be approximated. Simulated solutions therefore do not directly map to solutions of equal quality within the target game environment. Thus, minimising divergence between predicted outcomes and actual outcomes in the game is important for simulation based solutions.

### 2.3 Techniques Relevant to RTS Agents

This section introduces a diverse set of algorithms utilised for game AI. Algorithms not directly related to the research goals are summarised while those of interest are expanded.
2.3.1 Finite State Machine (FSM)

An FSM is a system defined as a set of states and transitions between those states. An AI implemented with this model can only be in one state at one time and moves to other states by well-defined transitions. The simplicity, robustness and efficiency of this model allowed early video games that had limited computing power to design AI with different behaviours (Fu and Houlette, 2004). The technique has been refined and developed over many years to produce highly sophisticated behaviours. One variation of FSMs is a stack based FSM. This is an FSM that appends the last state it was into a stack. It can resume its previous state from the stack (Tozour, 2004). The advantages of Finite State Machines advantages are that they are simple to implement, easy to debug and easy to tweak their behaviours. They are favoured by developers because they offer very predictable behaviour which makes them easy to test and tune for game balance. Their weaknesses are that they can become unmanageably large and complex for large games, and players can exploit their deterministic behaviours.

2.3.2 Scripted Agents

Scripting refers to the use of an interpreted language to avoid the need for compilation. It was developed for game AI to reduce turnaround time for AI developers needing to change AI behaviours. It allows developers to change AI parameters either while a game is being played or between games without forcing them to rebuild the game to modify the behaviour of the AI (Tozour, 2002). However, in RTS AI terminology, a scripted agent does not need to use an interpreted language. Instead, ‘scripted’ simply refers to the hard-coded, generally deterministic, nature of an agent.
2.3.3 Reinforcement Learning

Reinforcement Learning represents a broad area of research that explores the game space, through play and/or game state evaluation through heuristics and feedback (Sutton and Barto, 1998). The technique allows an agent to adapt to its environment through iterative sampling.

A popular reinforcement learning method is Q-Learning (Watkins, 1989). Q-Learning attempts to maximise the total reward for an agent over multiple actions in different states. This method addresses the challenge of selecting which actions should be performed in which states. The method seeks to explore state/action pair sequences to determine how actions perform in the long term. After sufficient sampling, the optimal action in a state is the action with the highest long-term reward.

The Q value update for state-action pairs is given below in Formula 2.1. The features of the formula are: s is the state, a is the action in state s and r is the reward. $\gamma$ is between $0 \leq \gamma \leq 1$ and describes the discount factor that assigns more importance to earlier rewards over later ones. $\alpha$ is between $0 \leq \gamma \leq 1$ and describes the learning rate.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha (r_t + \gamma \cdot \max Q(s_{t+1}, a) - Q(s_t, a_t))$$ (2.1)

One popular implementation of this method is to create a table containing the Q values for all state-action pairs. However, this quickly becomes unviable in larger state space problems where table sizes become too large to store, even on modern computers. Thus, research has sought to replace the table approach with an approximator function such as a neural network. (Mnih et al., 2015) used a hybrid
technique between deep learning with neural networks and RL called Deep Reinforcement Learning (DRL). They demonstrated that an agent using their technique could use only pixel values from a games display and game scores to learn how to play multiple different games. Furthermore, interest has risen in applying DRL to RTS games, specifically Brood War with TorchCraft (Synnaeve et al., 2016). The TorchCraft platform was designed to aid development of DRL based RTS AIs and will accelerate the rate that new ideas can be implemented and tested.

A similar model to Q-Learning, the Sarsa (State-Action-Reward-State-Action) model was developed by (Rummery and Niranjan, 1994). Sarsa is an on-policy TD learning model in contrast to Q-Learning which is off-policy.

Other approaches, such as Dynamic Scripting, is made adaptive or 'dynamic' through the integration of reinforcement learning with the scripting system (Spronck, Sprinkhuizen-Kuyper, and Postma, 2004). A rule base is constructed in which each rule specifies a predefined action and is associated with a probability of selection. The system selects a number of rules from the rule base, based on the individual rule’s selection weight. The combination of selected rules forms a ‘script’ which defines the behaviours of an AI. The AI is trained over a number of generations and based on the success or failure of each generation, each selected rule is penalised or rewarded. This decreases or increases the rules selection weight and thus the probability of the rule being selected in the future. Over a number of evaluations, successful rule sets become more probable thus creating successful scripts.

(Efthymiadis and Kudenko, 2013) used plan-based reward shaping to create a build order in Brood War. (Marthi et al., 2005) used Concurrent Hierarchical Reinforcement Learning within Wargus to learn how to manage an economy and strategy. They demonstrated their method outperformed flat concurrent Q-Learning. (Xiaoqin, Qinghua, and Jianjun, 2009) demonstrated that using their
hierarchical RL method converged faster than flat RL within in the First-Person Shooter game Quake 2. (Wender and Watson, 2012) showed that multiple RL variations were capable of winning small scale combat scenarios within Brood War. (Madeira, Corruble, and Ramalho, 2006) applied RL to large scale armies within a turn-based strategy game by combining units into groups for coordination. (Bishop and Miikkulainen, 2013) combined evolutionary algorithms to rank features to be used by a Q-Learning RL agent. (Dey and Child, 2013) used Q-Learning to optimise the performance of behaviour trees. (Glavin and Madden, 2015) improved an agent’s shooting technique within a First-Person Shooter game. (Emigh et al., 2016) utilised nearest neighbour to help RL adapt to large dimensional environments. (Wender and Watson, 2014) combined RL with CBR to control the micro of a small number of units within Brood War. They showed that the hybrid approach converges faster than just RL, but the approach would require more work in regard to working with larger army sizes. (Micić, Arnarson, and Jónsson, 2011) applied RL to a very small number of units within Brood War.

### 2.3.4 Bayesian Approaches

This section gives a general outline of Bayesian approaches. Bayesian methods come in a number of different forms but generally try to model problems in probabilistic terms.

This allows an agent to learn about models and adapt based on different probabilities.

(Synnaeve and Bessiere, 2011) developed a Brood War AI using Bayesian programming for the micro management of the units in the agent’s army. They tested their system on 12-36 units and noted that more collisions took place between
units in larger armies. They concluded that their approach needed more work to deal with melee unit encounters. Their approach utilised a form of flocking based pathfinding and influence maps that take into account terrain from (Preuss et al., 2010). In an extension of this work, (Synnaeve and Bessiere, 2012) predict the opponent’s tactical decisions, classifying them into four broad types ground, air, drop and invisible. Their agent could autonomously adapt its own tactical decision making as a result. (Stanescu et al., 2013) trained a Bayesian network on simulated game data to predict the victor between army conflicts.

### 2.3.5 Genetic Algorithms (GA)

A Genetic Algorithm is a search technique that uses a set of randomly generated phenotypes (populations) to explore a state space and converge on an optimum solution as defined by a fitness function. Each genotype is encoded with genes that can be decoded and expressed as a phenotype (Stanley, 2004). A phenotype represents a potential solution within a state space whose optimality is rated by a fitness function. This technique is powerful in that it can balance the effort spent in exploration of the state space with the exploitation of discovered optima. The weaknesses commonly associated with GAs are that they require a relatively large number of evaluations to converge on a solution which can make them slow. It can also be difficult to develop an encoding scheme which allows the efficient evolution of specific behaviours. Finally, it can be difficult to design a fitness function that results in the desired behaviour.

Recent works using genetic algorithms include the following. (Benbassat and Sipper, 2013) combine genetic programming to improve Monte Carlo Tree Search in the board game of Othello. (Alhejali and Lucas, 2013) also incorporated the use of GAs to enhance Monte Carlo performance in the game Ms. Pac Man. (Young
and Hawes, 2012) used case injected GA to learn goal priority profiles for use in goal management for a Brood War agent. Their system allows the agent to change goals in game based on observations.

(Fernández-Ares et al., 2012) developed an agent for Planet Wars, a 2010 Google AI challenge. They used GA to tune parameters of different bots to specialise on different maps.

(Liu, Louis, and Nicolescu, 2013b; Liu, Louis, and Nicolescu, 2013a) use CIGAR (Case Injected Genetic AlgoRithm) for group control within Brood War. They tested their system against a small number of units (9 units for each player) with no terrain. They showed that their CIGAR system performed as well as standard GA but took less processing time.

(Jia and Ebner, 2015) created a general game playing agent utilising genetic programming (GP) that performed similarly to Monte Carlo Tree Search algorithms.

(Tong et al., 2011) used Feed Forward Artificial Neural Networks (FFANN) tuned by GA to play Warcraft 3 games. Their agent focused only on which units to create.

(Li and Kendall, 2015) used a hyper-heuristic methodology to generate adaptive strategies for games, where a high-level algorithm selected different heuristics to use at different stages of the game. (Nguyen, Nguyen, and Thawonmas, 2014) performed experiments with 24 units in Sparcraft using GA to change the micro management search heuristic function. They showed that as some army compositions are stronger than others, these changes should influence the search heuristic.

(Othman et al., 2012) used GA to create tactical attack paths for groups of units which improved the performance of their competition AI.
(Liu, Louis, and Ballinger, 2014) improved the performance of fast moving units (Vultures) within Starcraft tactical scenarios by tuning AI parameters with GA.

(Takahama and Sakai, 2015) used Differential Evolution to create an collective intelligence player for Othello. They showed that using majority vote from multiple agents from a learned evaluation function performed well.

(Garciá-Sánchez et al., 2015) used Genetic Programming (GP) capable of building a strategy (build order), composition of unit groupings and rules for a bot’s behaviour within Starcraft.

2.3.6 Artificial Neural Networks (ANN)

Artificial Neural Networks are a machine learning technique inspired by the human brain. ANNs consist of nodes connected by weighted connections, modelled after the neurones and the synapses which join them in the human brain. There are a number of ANN architectures of which Feed Forward Artificial Neural Networks (FFANN) are the most commonly discussed. These generally possess an input layer, a number of hidden layers and an output layer. A typical ANN learns through back propagation, which is a gradient descent method that incrementally changes the weights of connections in order to match the input/output relationships demonstrated in a training set (Russell and Norvig, 2002). An ANN using standard back propagated training does not undergo any change in structure when connection weights are modified. ANN based AI has the ability to adapt during gameplay, in that ANNs can make different decisions based on the input from the current game environment. However, video games tend to avoid ANNs using back propagation due to the requirements for:
2.3. Techniques Relevant to RTS Agents

1. Large Training Sets: ANNs with many inputs and a requirement to exhibit complex behaviours require a very large training set. In the context of video games, this implies a large database of recorded games.

2. Long Training Times: A long time is needed to train a complex ANN (the more complex the ANN, the longer it takes to train).

3. Predefined ANN structures: Developers are required to make an educated guess as to initial structure (number and configuration of neurones and connections) of the ANN since this strongly influences the performance and training of the ANN. Even if all of these problems could be addressed, the best outcome that could be hoped for would be that the ANN displays similar behaviours to those shown in the training data set.

ANNs have been used in dynamically adjusting a game’s difficulty (Yin et al., 2015). They have also been used in hybrid approaches such as with MCTS in improving game sampling (see Section 2.6).

ANNs have also been extended to deep learning models which use more complex neural network structures. Deep learning has been particularly focused on image classification. However, deep learning in combination with MCTS has recently seen great success in game AI with the AlphaGO AI beating a world champion (Silver et al., 2016).

2.3.6.1 NeuroEvolution of Augmenting Topologies (NEAT)

NEAT addresses several problems that occur in standard neural networks using back propagation. Typical ANNs rely on the designer of the system to make an educated guess regarding the network structure that is needed. They must balance the need to express the required solution space against the requirement that
the network should not be so large that finding the solution would take unreasonably long or require too large a training set. This is made more difficult because of the lack of certainty in exactly how the structure of the ANN affects the behaviour of the model it expresses. As (Stanley, 2004, p. 32) described; "NEAT starts out with a population of minimal structures and adds structure as necessary in order to minimise the number of parameters being searched. The resulting process of gradually increasing complexity is called complexification". The population trials a range of ANN structures and connection weights. NEAT then uses GA techniques to select the fittest of these ANN networks. Thus NEAT avoids the use of back propagation in modifying the ANN network. NEAT causes neural networks to compete in an arms race to become the fittest with little direct influence of the designer. NEAT allows the ANN structure to change and thus encodes a solution much more quickly than other ANN training techniques. Also, by encoding the solution of the GAs as an ANN one of the weaknesses of GAs is eliminated; that is the difficulty of relating the genetic code selected with the expressed solution behaviour.

NEAT has been applied to various areas, such as RTS agents. (Traish and Tulip, 2012) demonstrated NEAT’s ability to adapt its strategy in a game depending on different opponent strategies. However, training even a small number of strategies took a considerable amount of processing time within the game of Wargus. (Zhen and Watson, 2013) demonstrated NEAT performing micromanagement of units within Brood War with encounters of 12 vs. 12 units. NEAT has also been explored for the difficult problem of general game playing (Hausknecht et al., 2014). NEAT has been further developed by Modular Multi-objective NeuroEvolution of Augmenting Topologies (MM-NEAT) to evolve a modular neural network to play Ms. Pac-Man.
2.3. Techniques Relevant to RTS Agents

2.3.7 Game State Search

Game State Search refers to techniques that actively explore and exploit their game state/action space such as Alpha-Beta, depth-first and best-first search. The Alpha-Beta and minimax methods search the game state tree with the use of an evaluation function. In contrast, the best-fit and depth-first search methods are guided through the use of a heuristic such as Life-Time Damage (LTD) and Life-Time Damage 2 (LTD2) defined by (Kovarsky and Buro, 2005). These heuristics estimate the value of a game state based on the health and damage of units in each players army.

\[ dpf(u) \frac{damage(w(u))}{cooldown(w(u))} \]  

\[ LTD(s) = \sum_{u \in U_1} hp(u) \cdot dpf(u) - \sum_{u \in U_2} hp(u) \cdot dpf(u) \]  

\[ LTD2(s) = \sum_{u \in U_1} \sqrt{hp(u)} \cdot dpf(u) - \sum_{u \in U_2} \sqrt{hp(u)} \cdot dpf(u) \]  

This review will focus on simulation-based search, where a technique evaluates decisions using a simulation. Simulations allow agents to reason about their environment in an accelerated manner, independent of the originating environment. Using simulations, an agent can repeatedly search for favourable decisions from any game state, adapting to situations as they arise. However, this ability comes with the requirement that the simulation test bed mimics the original environment as closely as possible. This is important because differences between simulation and the original environment can result in simulation decisions not translating to the intended platform. The issue is illustrated by (Churchill and Buro, 2012) where their results showed a difference between simulated results generated using Alpha-Beta search and actual results in Brood War. (Barriga, Stanescu,
Chapter 2. Literature Review

and Buro, 2015) used a modified Alpha-Beta method search referred to as ‘Puppet Search’ on the SparCraft engine. (Stanescu, Barriga, and Buro, 2014) built a Sparcraft agent using an abstracted Alpha-Beta search mechanism that used a hierarchical goal based tree of three layers: strategy, action sequences and raw per unit execution order. They showed their work outperformed standard Alpha-Beta, UCT, and Portfolio Search in large combat scenarios featuring multiple bases and up to 72 units per player under real-time constraints of 40 ms per search episode. (Ontanón and Buro, 2015) demonstrated progress in dealing with large branching factors by combining minimax search with hierarchical task decomposition in MicroRTS.

(Churchill and Buro, 2011) used heuristics of the depth-first branch and bound search in optimising build orders for Brood War. (Churchill, Saffidine, and Buro, 2012) performed search to improve micro management of units within Brood War. They evaluated game states using playouts, in a similar manner to MCTS (Section 2.6). They used Alpha-Beta Considering Durations (ABCD) in 8-unit vs. 8-unit battles with no pathfinding or collisions.

(Oliveira and Madeira, 2015) used a modified best-first search algorithm to search potential fields in order to generate walls in Brood War.

(Ontanón and Buro, 2015) applied an Adversarial Hierarchical Task Network approach in their work with microRTS. They address the use of terrain and pathfinding and showed that including pathfinding with a particular search strategy (AHTN-LL vs AHTN-LLPF) gave three times better results, although it did slow the search considerably. However, their work was conducted on very small maps (a maximum of 16 x 16 tiles).

While these techniques allow agents to adapt to their environment, they require some form of heuristic or game state evaluator to guide search. However, some
environments are very complex, which makes it difficult to model an appropriate heuristic. Monte Carlo Tree Search (MCTS) doesn’t require an evaluation function or heuristic to guide search, instead using a system of ‘playouts’ to determine favourable decisions. However, an evaluation function can still be used with MCTS in order to avoid simulating through to a game’s terminal state. This is done when it is impractical to simulate to a terminal state within a short time window as illustrated in the work of (Uriarte and Ontañón, 2014).

2.3.8 Supportive algorithms

The following lists and describes algorithms that support and enhance the performance of other RTS AI techniques.

2.3.8.1 Clustering

Clustering is an unsupervised machine learning approach that seeks to group similar data points. This technique was sometimes used in conjunction with CBR to identify groups of strategies. There are many types of clustering which group data in different ways. (Bauckhage, Drachen, and Sifa, 2015) presented a breakdown of several clustering techniques and discussed their strengths and weaknesses.

2.3.8.2 Nearest Neighbour

This technique is relatively simple and is used for finding similar data points in a data space. Nearest neighbour simply calculates the nearest data points using some type of distance metric. Nearest neighbour is useful in techniques such as Case Based Reasoning (CBR), where a given game state may not exactly match
a recorded case. CBR searches through the data space for the closest recorded
game state as indicated by the distance function.

2.3.8.3 Optimization

(Babadi, Omoomi, and Kendall, 2015) presented an Enforced Hill Climbing Based
System for General Game Playing.

(Recio et al., 2012) used ant colony optimisation to play Ms. Pac-Man. They noted
that their approach was competitive with MCTS and GA algorithms.

(Richoux, Uriarte, and Ontanón, 2014) used a constraint optimisation algorithm
to determine the best way to wall off choke points (as defined by Perkins, 2010)
in Brood War.

2.3.8.4 Opponent Modelling

Opponent Modelling is a technique that can be used to aid other systems of an
RTS agent. At a strategic level, an agent may reason about past games against
the same opponent and infer likely enemy strategies. For example, it could deter-
mine whether the opponent was more likely to perform a defensive, aggressive,
balanced or research focused strategy, and the determination is used to influence
the response of the agent. Alternatively, the agent could predict the opponent’s
strategy from observations during a game and thus modify its own response.

This same approach can also be used at the tactical decision and micro-management
levels. Predicting an opponent’s behaviours allows the relevant component of an
RTS agent to focus on a smaller subset of opponent actions. This reduced state
space allows the agent to make more effective use of its processing time, and in
some cases lead to the agent exploiting weaknesses in their opponents.
Due to the predictive nature of opponent modelling, it also acts as a step towards dealing with the incomplete information environment of Brood War.

(Synnaeve and Bessière, 2011; Dereszynski et al., 2011) constructed a model that predicted an opponent’s build order (strategy) even with missing observations. (Weber, Mateas, and Jhala, 2011) used a particle filter to predict enemy unit movements when obscured by the Fog of War (unobserved). (Avontuur, Spronck, and Van Zaanen, 2013) predicted a player’s skill level. (Valls-Vargas, Ontanón, and Zhu, 2015) predicted one of the several models that fit a player’s behaviour. (Yang and Roberts, 2013) predicted outcomes in the team based games Dota, Warcraft 3 and Starcraft 2. (Park and Kim, 2013) used incremental active learning to infer the strategy of Iterative Prisoner Dilemma (IPD) games. (Cho, Kim, and Cho, 2013) built a model that trained on Starcraft replays to predict opponent build orders. (Sarratt, Pynadath, and Jhala, 2014) described a method of updating belief distributions through leveraging information sampled during MCTS on iterative prisoner’s dilemma. (Leece and Jhala, 2014) constructed Markov Random Fields in Brood War from a number of professional replays. They predicted the number of units as well as the existence of buildings/units. (Stanescu and Čertický, 2016) predicted the number and type of units a Starcraft or Warcraft III player trained in a given amount of time.

2.4 Data Retrieval

Research projects generally use game data about the progress of games in one form or another in order to test theories. Hence, obtaining data is the first step towards building an agent capable of making decisions in the RTS domain.
These records of game progression are called game traces. Game traces record the change of game state over the duration of a game.

Game traces for board games such as GO (Bossomaier et al., 2012) and Chess (Lane and Gobet, 2012) are obtained from a sequential list of user interactions with the game. In GO and Chess, the range of user interactions with the game is very limited, and the effect of any player action in the game is deterministic. For example, in GO it is known that when a player places a stone such that an opponent’s stones are surrounded, then the opponent’s stones will be eliminated. However, in commercial RTS games such as Starcraft 2 (*Starcraft*), the set of user interactions is often far larger than in a classic board game, and the effect of player actions on game state is uncertain. A user can move a camera, move units, construct buildings, train units, buy upgrades and much more. Some of these commands (e.g. camera movement) have no effect on game state, and for others, (e.g. unit movement) the effect is indirect. Furthermore, in general, large commercial games such as Starcraft 2, Overwatch, Dota 2, Warcraft 3 and other titles do not permit, and provide no legitimate tools to directly access the internal game state. Instead, the only readily available game data is provided as a list of player interactions with the game.

There are some open source RTS games such as MicroRTS, Wargus, and ORTS, where internal game state may be accessed. However, these were not considered suitable as the basis of this research because they are either oversimplified, unbalanced, or unreliable.

The exception to these generalizations is Starcraft Brood War. Brood War is the highly successful predecessor to Starcraft 2. It is a complex, and finely balanced RTS game. It also has a third party API known as the Brood War API (BWAPI) that does provide access to the internal game state, and supports an extensive research community. Because of its complexity, balance, reliability, and access to
internal game state, Brood War was chosen as the basis for most of the research conducted in this thesis.

Access to game data may be classified into 4 categories:

1. Cases where the internal game state may be accessed directly, such as through the BWAPI interface for Brood War, or in open source games.

2. Cases where user interactions can be used to directly and unambiguously rebuild the game state, such as Chess, or Go.

3. Cases where user interactions give only an indication of the game state such as Starcraft 2. These cases may have issues caused by a lack of correspondence between user interaction and final outcomes in terms of game state.

4. Cases where there may be no record of either user interaction or resulting game state, such as when users interact with configuration options within a menu system.

### 2.5 Cased Based Reasoning (CBR)

Case-based reasoning (CBR) approaches store game traces from previously played games in a case library, allowing the AI to use past experience to make decisions about new situations (Muñoz-Avila and Cox, 2008).

The technique relies on establishing a large library of previous cases so that when new situations occur in a particular game, actions can be effectively derived from past successful behaviours.

This approach shows promise for generating in-game adaptive agents, although (Weber and Mateas, 2009b) point out that "the drawbacks of this approach are that it requires a comprehensive example set in order to achieve good results and
is sensitive to noise”. Case libraries (Weber and Mateas, 2009c) can be built using online replay repositories generated by players recording annotated replays (Sugandh, Ontanón, and Ram, 2008; Ontañón et al., 2007), or can be generated by playing different hand-coded AI against each other (Weber and Mateas, 2009a). The way in which cases in the library are encoded, compared and recalled to generate an effective agent is the focus of much research (Aha, Molineaux, and Ponsen, 2005; Molineaux, Aha, and Moore, 2008; Ng, Shiu, and Wang, 2009).

CBR methods have been used successfully to create adaptive RTS agents. In general, such methods store plans with an associated game state and use this data to reason about future encounters. (Aha, Molineaux, and Ponsen, 2005) demonstrated a CBR agent capable of identifying and adapting to a randomly selected opponent which demonstrated good results. Their agent relied on the availability of a set of pre-generated responses, each capable of winning against an opponent from a given position. (Weber and Mateas, 2009b) improved CBR approaches by changing the case retrieval approach, leading to significantly better results. Their agent demonstrated a high win rate in experiments with imperfect information. Other CBR methods have focused on the use of recorded human player interactions to make decisions (Sugandh, Ontanón, and Ram, 2008; Mehta and Ram, 2009). However, while CBR has been successful in creating adaptive RTS agents, they face a number of challenges. Responses derived from human players can be of inconsistent quality due to the diversity of human players’ skills and the nature of human play. Standard CBR approaches are also ill-equipped to make decisions if there is no similar recorded context. More recently (Cadena and Garrido, 2011) used CBR for strategy decisions and combined CBR with fuzzy logic for making tactical decisions within Brood War. (Oh and Kim, 2015) implemented an imitation style agent for deciding when to attack and retreat within Brood War.
(Gemine et al., 2012) created a Starcraft 2 agent that successfully learned to imitate the game’s inbuilt Terran very hard agent and as a result beat the hard ranked agent.

(Weber and Ontánón, 2010) present a technique for annotating traces with goals from expert demonstrations of Starcraft gameplay. (Gong et al., 2012) utilise conditional random fields to segment and label in game actions from Starcraft 2. (Cho and Kim, 2013) analysed the difference between human and AI Starcraft competition results. They found that bots play styles were simpler and more successful if they could identify and adapt to their opponent’s strategy correctly. (Ballinger and Louis, 2013) utilise case-injection with a coevolutionary algorithm to defeat difficult opponents. (Wehr and Denzinger, 2015) utilise CBR at both a central AI and individual unit level. They identify sequences of actions related to a directive and cluster similar action sequences. (Wirth and Fürnkranz, 2015) used expert game annotations to automatically create an evaluation function for Chess.

In contrast to CBR, Goal Oriented Action Planning (GOAP) build plans relative to a given goal and execute them.

(Hoang, Lee-Urban, and Muñoz-Avila, 2005) used Hierarchical Task Networks for a goal-driven model to react to unforeseen situations. (Molineaux, Klenk, and Aha, 2010) demonstrated a conceptual goal based model that was later used by (Weber, Mateas, and Jhala, 2010). (Weber, Mateas, and Jhala, 2010) further developed a goal-driven approach to work with multiple Brood War systems which was capable of automatically detecting and correcting issues in game.

The work by (Muñoz-Avila and Cox, 2008) demonstrated a hybrid approach they named case based planning. Where traces were recorded with the goals they were trying to achieve. This allowed the reuse of existing plans to solve new problems.
However, determining what goals to follow in goal based planners, how to execute them, and when and how to replan become important questions that need to be explored. CBR in contrast does not require these planning aspects as it has the inherent ability to choose a different response as game progresses.

## 2.6 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) and its use in GO AI has been the subject of much research in recent years. MCTS exhibits many strengths that make it an excellent algorithm for game agents. MCTS is an any time algorithm that improves the accuracy of decisions as its given more processing time. MCTS has proven to converge to minimax solutions over enough time (Browne et al., 2012). Any time algorithms can retrieve decisions at any time. This is especially important for real-time games or games with enforced time limits on making decisions.

Another strength of the MCTS algorithm is that it doesn’t require a heuristic to guide its search like other algorithms (Chaslot et al., 2008). Instead, the MCTS performs many samplings of a decision to determine the effectiveness of that decision.

MCTS is a tree based algorithm where each node in the tree relates to a decision the agent can make. This is illustrated in Figure 2.1 from (Browne et al., 2012). Node selection is based on a ‘policy’. One common policy is based on the Upper Confidence Bounds (UCB1) formula 2.5 presented by (Auer, Cesa-Bianchi, and Fischer, 2002).

UCB1 Formula:

\[
UCB1 = v_i + C \sqrt{\frac{\ln N}{n_i}}
\]  

(2.5)
2.6. Monte Carlo Tree Search (MCTS)

Where $v_i$ is the score of node $i$, $C$ is a tunable parameter, $N$ is the number of parent’s visits, and $n_i$ is the number of visits to node $i$. UCB1 was extended into MCTS by (Kocsis and Szepesvári, 2006) in an algorithm known as UCT. This algorithm balances exploration of sub-optimal parts of the tree with exploitation of favourable parts by selecting a node to expand based on its "expected payoff" as calculated by the UCB1 formula.

The UCT algorithm describes how to explore and expand the decision tree. Generally, each other layer of the tree corresponds to the opponent’s decisions.

The MCTS Basic Algorithm as shown in Figure 2.1 can be broken into four activities.

1. Selection: Starting at the root node, the tree policy is applied recursively to select the child nodes with the highest expected reward until a non-expanded node is reached.

2. Expansion: If the selected node does not result in a terminal state then the tree is expanded on the selected node. It is expanded by adding a child node for each available decision to the selected node and then choosing one of them.
3. Simulation: The new child node is evaluated by running a simulation until a terminal state is reached using a “default policy”. This policy determines any other decisions that need to be made. This process is also known as a playout.

4. Back Propagation: The result of the simulation is back-propagated up the selected nodes updating each node’s score and the number of times they have been visited.

Playouts involve completing a game using a “default policy” to determine actions taken after those selected within the tree. This could be making random decisions until an end state is detected, usually, playout policies are improved beyond purely random moves to improve agent performance. In the RTS context, the “default policy” may involve the use of a scripted agent.

When a game model can be modelled exactly the MCTS tree doesn’t have to be thrown away completely after a turn. Instead, it can simply reset the root node to the decision made by the agent or its opponent. This works particularly well for board game AI where the number of rules and interactions is relatively easy to simulate. At the end of a turn the agent will keep the sub tree of its decision, thus increasing the number of samples that each sub tree receives over multiple turns.

In contrast, a game of Starcraft is too complex to be exactly replicated in a simulator. The results of a decision made by an agent in the simulator may not exactly represent the results gained by an agent in Starcraft making the same decision (given differences between a simulator and Starcraft). This means that Starcraft simulation based MCTS agents are not able to accumulate tree sampling like board game based MCTS. Furthermore, complex game models require substantially more processing time per MCTS sample than traditional board games. In
the case of Starcraft, pathfinding in particular is an expensive system that slows the number of samples per second that can be taken.

Many research papers applying MCTS to games and especially RTS games have severely limited the complexity of the agent. Such limitations include restricting opposing armies to only a few units each, restricting the range of unit types involved, and neglecting pathfinding and unit collisions. Such limitations present issues when applying the research to real games.

(Ontanón, 2013; Chung, Buro, and Schaeffer, 2005; Zhe et al., 2012; Balla and Fern, 2009) presented research using MCTS with small numbers of units. (Ontanón, 2013) applied a Combinatorial Multi-Armed Bandit formulation of the Monte Carlo algorithm and gained better performance in games with large branching factors. His NaiveMCTS algorithm performed better in the microRTS game compared to other sampling strategies. (Chung, Buro, and Schaeffer, 2005) presented an investigation into using Monte Carlo planning for tactical situations in ORTS. (Zhe et al., 2012) designed their own simulator and also applied Monte Carlo planning to tactical situations. (Balla and Fern, 2009) applied UCT within the Wargus platform. Their approach was capable of beating human players in simplified scenarios.

Other authors have tried to scale up the number of units by grouping units together to act as one. (Uriarte and Ontañón, 2014) defined the number of groups as a parameter. Other authors have applied clustering. (Justesen et al., 2014) used clustering based on unit type and spatial proximity, but again the clustering parameters were fixed. (Justesen et al., 2014) identified that when experimenting with less than 32 units per side, clustering for grouping was less effective than non-clustering. These approaches, while reducing the dimensionality of the search space, may not generalise well, or work well in more dynamic game scenarios.
(Balla and Fern, 2009) allowed merging of units into groups as an action in their application of UCT in Wargus. (Bowen, Todd, and Sukthankar, 2013) grouped units based on spatial reasoning and used MCTS to search for group merging or attack commands between groups within Starcraft.

(Churchill and Buro, 2015) performed grouping in a card game. These approaches are dynamic, but start grouping from the most finely divided state, that of individual units.

Other research has worked on improving the MCTS selection phase. (Baier and Winands, 2012) presented Beam Monte-Carlo Tree Search. When a predetermined number of simulations has traversed the nodes of a given tree depth, these nodes are sorted by their estimated value, and only a fixed number of them (or beam) is selected for further exploration. While the parameter of beam width (i.e. number of nodes selected for further exploration) was domain specific, Beam MCTS offered better performance than standard MCTS (Cazenave, 2012). Selection improvement was also affected by defining macro decisions (Powley, Whitehouse, and Cowling, 2012). Macro actions reduced the number of decisions the agent could take, thus decreasing the search space and improving search speed at the cost of decision fidelity. (Powley, Whitehouse, and Cowling, 2012) described a state space reduction from $10^{1556}$ states to $10^{103}$ states. This reduction in fidelity was noted to sometimes reduce performance (Frydenberg et al., 2015). This work was further extended to the multi-objective travelling salesman problem (Powley, Whitehouse, and Cowling, 2013). (Benbassat and Sipper, 2013; Alhejali and Lucas, 2013) used GA to evaluate game states to help guide MCTS selection. (Benbassat and Sipper, 2013) combine genetic programming to improve Monte Carlo Tree Search in the board game of Othello. (Alhejali and Lucas, 2013) also incorporated the use of GAs to enhance Monte Carlo performance in the game Ms. Pac Man.
Depth limiting search within MCTS was also found to be effective (Fujiki, Ikeda, and Viennot, 2015). Further research into selection improvement was also conducted by (Perick et al., 2012; Park and Kim, 2015).

Other research has focused on improving MCTS through improving the playout policy. Work by (Marcolino and Matsubara, 2011) had MCTS play against multiple opponents in the playout phase, which increased the MCTS agent’s performance. They interestingly noted that simply adding agents to the pool was not always beneficial and that opponents were best chosen when they improved the MCTS agent’s performance. (Zhuang et al., 2015) trained an ANN to be occasionally used within the playout phase to improve performance. They noted that overusing the ANN could degrade performance as its much slower than using random playout, so they balanced its use.

Other research also focused on improving playouts. (Kao et al., 2013) improve the MCTS model by learning the default policy online. (Basaldúa et al., 2014) also used a learning policy in place of the default policy by learning from win and loss rates in Go. (Swiechowski, Mandziuk, and Ong, 2016) created six enhancements for guiding exploration, optimised the UCT formula and implemented a custom method of using state transpositions for a General Game Playing (GGP) agent. (Churchill and Buro, 2013) modified the UCT algorithm to account for simultaneous and durative actions. They also created a method called Portfolio Greedy Search to efficiently search large combat scenarios within SparCraft, although this method performed hill-climbing rather than using MCTS.

Other works have developed Monte Carlo’s ability to work within partially observable environments. (Furtak and Buro, 2013) created an imperfect information Monte Carlo search for card games. (Wang et al., 2015) developed a belief-state MCTS algorithm capable of working with imperfect information which was evaluated against Phantom Tic-Tac-Toe and Phantom Go. (Mizukami and Tsuruoka,
Chapter 2. Literature Review

2015) developed a method of analysing and predicting an opponent’s moves for the imperfect information game Mahjong. (Ivanovo et al., 2015) model an opponent with the Apprentice Learning method and apply it to MCTS within an imperfect information game of Capture the Flag. (Cowling, Powley, and Whitehouse, 2012) developed three information set MCTS algorithms which were evaluated in multiple imperfect game environments. (Cowling, Ward, and Powley, 2012) showed a binary tree based MCTS performed well within the imperfect information game "Magic the Gathering". (Cowling, Whitehouse, and Powley, 2015) developed a MCTS model capable of bluffing within the imperfect information game of card game "The Resistance". (Nijssen and Winands, 2012) developed a technique location categorization to work with MCTS to play the imperfect hide-and-seek game "Scotland Yard".

Other work has focused on using MCTS within dynamic and partially observable environments (Naveed et al., 2012).

(Pepels, Winands, and Lanctot, 2014) and (Pepels and Winands, 2012) examined ways to improve search by maintaining information between decisions. One of the changes they made was to add a decay function on the tree values "between" searches, so some information is kept between searches.

Research has also explored using MCTS within multi-objective environments. (Perez, Samothrakis, and Lucas, 2013) demonstrated a method of using offline and online processing to build a multi-objective MCTS model. (Perez et al., 2015) developed a multi-objective MCTS model that outperformed two other methods in multiple real-time domains.

(Baier and Winands, 2013; Baier and Winands, 2015) showed that MCTS could
be improved by integration with Minimax search. They used minimax to perform shallow depth searches within MCTS which increased performance without using evaluation functions. (Lanctot et al., 2014) had MCTS store two types of information, the estimated win rate and a heuristic evaluation from minimax search.

Given the high computational requirements of simulations in MCTS, some research has examined parallelising search where possible. (Barriga, Stanescu, and Buro, 2014) implemented MCTS on a GPU. (Schaefers and Platzner, 2015) presented a cluster based approach to MCTS on GO where they utilised 2048 CPU cores.

Other research has applied MCTS to General Game Playing (GGP) (Benbassat and Sipper, 2014). One such application improved its performance by integrating pathfinding (Chu et al., 2015).

(Browne, 2013) outlined potential issues with using MCTS for game playing. They outlined a specific scenario where MCTS performed very badly and addressed the issue by introducing domain knowledge that significantly improved the results. (Maes, St-Pierre, and Ernst, 2013) tried to automate the process of tuning a MCTS model which is generally achieved through the application of domain knowledge. The survey paper by (Ontanón et al., 2013) also pointed out that automatically mining patterns that are given as domain knowledge is an area in need of further research.

(Nguyen and Thawonmas, 2013) tried a multi-agent approach to controlling ghosts in Ms. Pac-Man. They showed that having some rule based agents was better than having all MCTS ghost agents. They indicated that one of the issues was that using more MCTS agents resulted in less sampling per agent, as the agents had to share computational resources.
Many of the above RTS related above works speedup playouts by using highly simplified RTS simulators that neglect collisions, pathfinding, and terrain effects. The problem with this approach is that the solutions found may not translate very well to actual RTS games. If playouts to evaluate the success of the MCTS sample are performed within the simulator, then the success of the sample does not always extend to the actual game. (Churchill and Buro, 2012) noted this issue when they observed that their simulation results were not replicated in the actual game of Brood War.

To date the only way to guarantee that a search method generates useful responses within a game has been to carry out the search within the actual game. For the games currently used as research platforms this has involved limitations related to screen handling that severely restrict simulation speeds. While useful for research purposes, this method is not a practical approach to support a search based agent in an actual game.

What seems to be required to successfully apply search based agents in full scale RTS games, is a way to conduct a large number of simulations that very accurately model the target environment (including pathfinding, collisions, and terrain) as fast as possible.

### 2.7 Path Finding

This review of pathfinding techniques is focused on the area of uniform grid-cost pathfinding. Uniform grid-cost pathfinding is useful and relevant for RTS AI because RTS AI research platforms such as Brood War use such grids for their maps. The models explored offer a reduction in the amount of processing required for
pathfinding relative to the classic A* algorithm and can also handle constant changes to the environment without incurring too much processing overhead.

The techniques are examined for their potential contributions to a real-time MCTS approach for tactical planning. The best techniques should:

1. Perform path search as quickly as possible
2. Find optimal paths
3. Work within a dynamic map and update changes to the map quickly

Optimal paths are considered important since timing can be important in the success or failure of an action, and the use of non-optimal paths introduces an unwelcome source of uncertainty in the evaluation of playouts.

### 2.7.1 A* Search

This classical pathfinding algorithm relies heavily on an open set, closed set and a heuristic to generate optimal paths in an environment. The algorithm is commonly used as a baseline to analyse the performance enhancements of other algorithms.

### 2.7.2 Compressed Path Databases

In a compressed path database, optimal paths between all pairs of locations are pre-computed. Since straightforward ways to store pre-computed paths are prohibitively expensive, even for maps of moderate size, the pre-computed path data is compressed, reducing the memory requirements dramatically. At runtime, pathfinding is very fast, as it requires only visiting the locations on an optimal path. In each location, a quick computation provides the next move along the
optimal path. The compression factor reaches two orders of magnitude, bringing the memory requirements down to reasonable values. Compared to A* search, the runtime speedup reached and even exceeded two orders of magnitude. When averaged over paths of similar cost, the speedup reached a value of 700 in experiments by (Botea, 2011; Botea, 2012). However, this approach does not work well in dynamic environments, since the speedup is due entirely to the pre-processing step.

2.7.3 Cooperative Pathfinding

Cooperative pathfinding algorithms are specifically designed for multiple agents moving in an environment. In the work by (Silver, 2005) all agents were aware of all other agents routes. This work searched paths in time-space space using a 3D grid. Where 2 dimensions represent the map and the other dimension stored the times units occupied cells.

While this method would lead to better overall pathfinding for armies controlled by an agent, it would come at the cost of more processing time and memory. This is because the large map data is represented in 3 dimensions and it takes into account all units routes.

2.7.4 SubGoal

The SubGoal algorithm (Uras, Koenig, and Hernández, 2013) is one of the strongest performing algorithms in respect to its ratio of search speed to allocated memory overhead and pre-processing time. It is one of the fastest algorithms that was submitted to the Grid-Based Path Planning Competition (GPPC) (Sturtevant, 2013).
However, given the current structure of its pre-processed data, it cannot be applied efficiently in dynamic environments. It would require changes to the algorithm’s pre-processed data storage design in order to work effectively in dynamic environments.

### 2.7.5 JPS

Jump Point Search is a pathfinding algorithm that exploits ‘path symmetry’ in uniform-cost grid maps in order to prune search paths at runtime. It returns optimal paths significantly faster than A* search. Symmetrical paths are any paths between two points which share a start and end point, are the same length, and differ only in the sequence of moves taken. Figure 2.2 illustrates the operation of JPS. JPS starts by considering all neighbouring nodes from the starting point. If JPS identifies a neighbour in a particular direction as the start of a possible path, it continues to extend that path in that direction until it is blocked by an obstacle or identifies a ‘jump point’. If a path is blocked, then all nodes along that expansion direction are pruned from further consideration. While JPS is extending a path in a particular direction, it identifies a set of ‘natural’ neighbours for a node under evaluation. A natural neighbour is defined by the direction of expansion. Expansions in a cardinal direction define their natural neighbour as the next node in the same direction (the Eastern natural neighbour is to the East). When expansion is diagonal, the set of natural neighbours includes only 3 nodes: the next node along the diagonal of the expansion, and the next vertical and horizontal nodes, also in the direction of expansion. Nodes are expanded in diagonal directions after considering the vertical and horizontal expansions until they are either blocked or a forced neighbour is found. A forced neighbour (striped cell) is identified when a neighbouring cell is blocked (black cell), which allows for a change of expansion direction that is not within the set of natural neighbours (clear white cells) (See
Figure 2.2: Jump point search natural and forced neighbours. Black cells represent blocked cells or obstacles. Striped cells represent forced neighbours. Clear white cells represent open or unblocked cells. Grey cells are not considered during this directional expansion.

Examples of Figure 2.2. If a forced neighbour is found then the node is identified as a ‘jump point’. This point represents the first node in a particular search direction from the original node that must consider expanding in other directions (Harabor and Grastien, 2012).

Sub figures within Figure 2.2 show how natural and forced neighbours are evaluated given the direction of travel (diagram modified from (Harabor and Grastien, 2011)). Forced neighbours are striped cells and natural neighbours are clear white cells. All grey cells represent nodes culled from search. Figures 2.2.a shows that with no blocked cells then only the natural neighbour is expanded (East to 5). Figure 2.2.b shows that a blocked cell to the North creates a forced neighbour at (North-East 3) in addition to the natural neighbour. Figure 2.2.c demonstrates that diagonal directions have 3 natural neighbours, 1 diagonal and 2 respective cardinals (NE 3, North 2 and East 5). Figure 2.2.d demonstrates that a forced node exists at (NW 1) in addition to its natural neighbours. These cases are symmetrical and can be rotated/flipped to generalise to all local scenarios.
2.7.6 JPS+

JPS+ (Harabor and Grastien, 2012) builds on JPS by pre-processing a grid into jump points. This significantly speeds up run time search since searches find the next jump point in a path using a lookup. The lookup eliminates the most significant processing overhead JPS incurs, which is the iterative evaluation of nodes along a particular search direction until the next jump point is encountered. The implementation used in this work stores preprocessed jump points in a table of size map width * map height * 8. This is because each cell of the map can move in 8 directions (4 cardinal and 4 diagonal). When pre-processing the map each cell is then associated with a corresponding jump point or boundary in each of the given directions. JPS+ then uses the values stored within the table to retrieve the next location to jump to instead of searching for it as in JPS. This leads to a significant speed up for static maps but presents issues with updating this data in dynamic scenarios.

2.7.7 Other Pathfinding Related Algorithms

(Naveed et al., 2012) used Monte Carlo for pathfinding in a dynamic and partially observable environment. (Lawrence and Bulitko, 2013) created Hill-Climbing and Dynamic Programming Search (HCDPS) algorithm to perform pathfinding on video game maps. However, their approach returns sub-optimal paths and they also mention that changing the precomputed data for dynamic maps may be slow. (Perkins, 2010) performed work in identifying terrain bottlenecks in Brood War maps. (Kring, Champandard, and Samarin, 2010) presented hierarchical pathfinding algorithms for finding slightly sub-optimal paths within dynamic environments. The ‘swamps’ algorithm identifies areas that paths are unlikely to travel through, and speeds up run time searches by ignoring these areas (Pochter,
Zohar, and Rosenschein, 2009; Pochter et al., 2010). The Hierarchical Pathfinding A* (HPA*) algorithm presented by (Botea, Müller, and Schaeffer, 2004) trades off path optimality in return for speed by pre-processing the environment into hierarchical regions and interconnecting paths. The Partial-Refinement A* (PRA*) algorithm presented by (Sturtevant and Buro, 2005) trades off path optimality also by performing search on an abstracted graph and they note it performed similarly to HPA*. The Triangulation Reduction A* (TRA*) algorithm presented by (Demyen and Buro, 2006) trades off path optimality by performing search on an abstracted graph based on decomposing the environment into triangles. They noted that in almost all cases TRA* outperformed PRA*. (Koenig and Likhachev, 2002) improved the A* algorithm to navigate environments of unknown terrain. (Reynolds, 1999) coordinated navigation with flocking. (Treuille, Cooper, and Popović, 2006) performed efficient crowd based navigation using a dynamic potential field with moving obstacles. Their technique has been incorporated into several commercial games like Starcraft 2 and Supreme Commander 2 (Ontanón et al., 2013). (Preuss et al., 2010) demonstrated moving teams of units in a natural way using Self-Organizing Maps (SOM), evolutionary algorithms (EA), flocking and influence maps.

2.7.7.1 Influence Maps

The disposition of resources, structures and friendly or enemy forces is very important to strategic and tactical decision making in RTS games. Influence maps provide an RTS AI with a strategic perspective of the game state by representing game features as a set of height maps known as layers (Sweetser, 2004). Each layer represents the ‘influence’ of a distinct game feature such as military forces, resources or buildings. Each individual game asset within a layer adds its ‘influence’ to the height of its location on the influence map with a surrounding drop
off with a defined radius. An AI can analyse a series of layers to make informed high-level decisions in relation to the game geometry without having to query individual game features. Information from different layers can be combined using techniques such as weighted sum desirability values which are calculated as a weighted sum of the corresponding grid values for the respective layers (Miles and Louis, 2006). Another technique is to combine values from different layers into a tree structure (Miles and Louis, 2006). (Sweetser, 2004) suggested that a neural network could take each of the layers as input, and learn how to combine them in order to make appropriate decisions. (Uriarte and Ontanón, 2012) explored unit kiting with influence maps. (Hagelbäck, 2012) combined A* and potential fields to improve navigation of armies relative to threats, by surrounding their targets. (Nguyen, Wang, and Thawonmas, 2013) used potential fields to make more effective scouting behaviour in Brood War. (Nguyen, Nguyen, and Thawonmas, 2013) used potential fields for unit positioning. (Ng, Li, and Shiu, 2011) applied potential fields, fuzzy measures and integral to perform micro management in Warcraft 3. They showed their agent was capable of dividing its army into subgroups and performing unit formation planning. (Park and Kim, 2015) combined influence maps with MCTS to help guide selection and tested in the general game playing framework. (Hagelback, 2016) further explored hybrid A* and potential field based navigation.

## 2.8 RTS Research Platforms

There are several RTS platforms used in RTS AI research. The following platforms were evaluated for their suitability to support the current research.

1. Wargus: A clone of the RTS game Warcraft 2 produced by Blizzard. This platform resembles an older style game with an extensive API in which
many characteristics of a game can be changed (Wargus/wargus: Importer and scripts for Warcraft II: Tides of Darkness, the expansion Beyond the Dark Portal, and Aleonas Tales).

2. Open-RTS (ORTS): An open source RTS game designed as a research platform. ORTS uses a client/server model to allow multiple agents from different authors to compete against each other (ORTS Homepage).


4. MicroRTS: A simple java based RTS platform to test AI techniques (Home · santiontanon/microrts Wiki).

5. Spring: An open source multi-platform RTS engine (Spring RTS Engine).

6. Starcraft Brood War/ BWAPI (Brood War Application Programming Interface): Blizzard’s Starcraft: Brood War (Brood War) is one of the most popular RTS games of all time (Starcraft). Brood War allows limited access to game state data through a third party API known as the Brood War API (bwapi/bwapi: Brood War API). Although relatively old, Brood War has a proven depth of strategy, and a large player base who still play the game. This provides a huge resource for capturing high quality game traces for CBR. Each player can control up to 200 or more mobile units on maps as large as 256x256 tiles. This results in a state space larger than $10^{1685}$ considering only positioning of units. Further, Brood War’s branching factor is estimated to be between $10^{50}$ and $10^{200}$ (Ontanón et al., 2013).

7. Starcraft 2: The most current RTS game produced by Blizzard. The author has a great deal of domain knowledge in regard to the strategies and mechanics of this game (Starcraft 2). However, currently there is no API to allow external agents to interface with the game.
8. TorchCraft: A framework for developing deep learning methods which utilises the BWAPI platform (Synnaeve et al., 2016).

Each platform was evaluated as a potential research platform. MicroRTS, Spring and ORTS are research RTS platforms that lack a rich player base from which to extract high quality game traces to support CBR methods. Spring is an open source RTS engine and the games it supports are relatively small titles compared to the commercial Blizzard RTS games such as Warcraft 2, Brood War and Starcraft 2. It is far more likely to find game replays suitable for CBR and a well balanced strategy design within the Blizzard based games. Wargus is a completely open source clone of Warcraft that could be useful as a research platform given the access it provides to game state, and the collection of replays it has access to from Warcraft.

However, currently one of the most popular RTS AI research platforms is Brood War. The game is complex and finely balanced, allowing sophisticated strategies to be applied, and the BWAPI interface allows external agents to be developed and tested. Many RTS AI publications use the platform and competitions existed to test AI from different researchers against each other. This provides a large body of other research to compare with work conducted using the platform.

Initially, Starcraft 2 was seen as a desirable research platform, based on the author’s experience and the level of community interest in the game at the time. However, due to the challenges obtaining useful game traces from Starcraft 2, and issues involved in developing an agent for that platform, the decision was made to use Brood War as the main research platform. Brood War offered much better access to internal game state, and facilitated agent development through the BWAPI interface. However, the initial interest in Starcraft 2 led to the early work on data extraction detailed in Chapter 3.
On the other hand, the SparCraft simulator was not used in work conducted for this thesis, as it does not implement path finding, collisions, or interaction with terrain. These features are required to support the high fidelity simulations identified as necessary early in the thesis. The TorchCraft platform was not used as it is a specialised platform designed to explore the use of deep learning algorithms.

2.9 Summary and Synthesis

This chapter reviewed a number of research areas relevant to the development of RTS AI. Of the adaptive techniques reviewed, techniques based on Monte-Carlo Tree Search (MCTS) have been selected as the basis of the current work due to their recent success in other areas (eg Go AI), ability to find solutions to novel problems, and the amount of other relevant research the area. However, MCTS based techniques face two critical challenges: how to effectively discover useful solutions in extremely large state and action spaces involved with RTS games within a very limited amount of time, and how to ensure that the results of sample evaluation (i.e. results from playouts) are applicable to actual games.

MCTS techniques require the use of simulators to evaluate playouts in order to determine the best action sequences separately from the actual game. However, fidelity between the simulator and the game is crucial to the utility of results generated by the simulator in the game. Simulators currently used in many works applying MCTS to RTS games neglect issues such collisions, pathfinding and interaction with terrain in order to speed up playouts. However, these aspects of RTS games can be critical in choosing successful strategies. Therefore, it is necessary to develop a high fidelity simulator.
Regardless of the quality of simulation, simulation speed is critical. Simulation speed determines how many MCTS samples can be evaluated, and in the context of the large search spaces involved, a high throughput of simulations is necessary to adequately explore the solution space. Furthermore, RTS games require agents to make decisions within small time windows such as 50 milliseconds. There are two responses to this issue: first to ensure simulations run as fast as possible, and second to adopt an effective tree policy to increase the probability of good solutions being found using a relatively small number of samples.

Pathfinding has been found to be the critical processing bottleneck for high fidelity simulations. Hence, fast pathfinding algorithms are crucial to increase the speed of high fidelity simulators. RTS environments are highly dynamic with mobile units moving around, and production and military building being created and destroyed. Brood War, the RTS platform selected as the basis of the current research uses a fixed uniform-cost grid as the basis of its maps. For these reasons, the JPS algorithm was selected as the pathfinding algorithm due to its speed, its ability to find optimal paths, and its ability to handle dynamic environments. Other algorithms such as SubGoal and ‘Compressed Path Databases’ perform well in static environments, but are difficult to adapt to dynamic environments. Near sub-optimal pathfinding algorithms such as HPA*, PRA* and TRA* were not investigated since in many experimental situations involving collisions and terrain, an agent choosing a sub-optimal path could lose where if an optimal path had been chosen it would have won. It would have required further research outside the scope of the thesis to investigate the impact of using non-optimal paths for agent responses. Work to improve the performance of the basic JPS algorithm is presented in Chapter 4 and even faster variants are presented in Chapter 5.

The pathfinding research focus in this thesis was on finding optimal paths on uniform grids. This is because Starcraft Brood War (the experimental platform
used in this thesis) utilises such uniform grids.

Effective sampling strategies are the other major issue faced by MCTS techniques. The underlying reason for the high dimensionality of RTS solution spaces is the high number of units and their actions. Current approaches to reducing this number by clustering units seem inflexible in the way groupings are determined. This thesis investigates an alternative approach: discovering effective unit arrangements by splitting and joining unit groups. The approach seeks to find the optimum number and composition of groups of units dynamically and is referred to as Dynamic Granularity. Work investigating the effectiveness of this mechanism is presented in Chapter 7.

Finally, the review noted that CBR techniques possess significantly fewer performance issues than those encountered by MCTS. Instead, CBR faces issues with responding appropriately to novel situations; precisely those situations that search based techniques are designed to handle. This thesis investigates whether combining CBR with Search can address both issues by populating a CBR database with the results obtained by Search. The use of a database to store results would allow simulations to be decoupled from the tight time constraints of the game loop, while allowing game agents to access the latest results of search with little computational cost. Conversely, the inclusion of search results in the CBR database would allow a CBR agent to adapt to novel situations. The technique is referred to as Search and Recall. Work demonstrating the feasibility and utility of this technique is presented in Chapter 6.
Chapter 3

Data Extraction using Screen Capture

3.1 Introduction

Detailed records of the progression of game states throughout a game are essential for research and to construct CBR databases using records of previously played games. This chapter seeks addresses the general problem of retrieving game state data from games with identifiable 2D icons for analysis. While good game traces exist for some games, they may not be available for others such as Starcraft 2. They may not even be recorded for actions of interest such as in the picking phase of DOTA 2. Thus, an important open research question is how can game traces be generated for arbitrary features or actions of interest in the absence of game logs recording such information. The retrieval of good game traces is essential for the operation of Case-Based Reasoning (CBR) systems as they rely on the description and characterization of game state progression in existing games.

This chapter demonstrates a framework for retrieving game state data from Starcraft 2 games. This work does not seek to build an agent capable of directly utilising input from the game in the form of pixels. For example, (Mnih et al., 2015) was able to process the pixel level input and create agents for 2600 different Atari games. (Lample and Chaplot, 2017) demonstrated an agent capable of playing
the 3D First Person Shooter game of ViZDoom again using just pixel input data. In contrast to those approaches, this chapter focuses on retrieving clean data sets for use by other algorithms such as CBR methods.

The framework utilises screen capture and image analytical techniques to retrieve game traces in respect to build orders from replays. This framework addresses data retrieval Category 2 (Section 2.4) games which provide only noisy game trace data. The aim is to retrieve the most accurate game trace information possible for later analysis.

The image capture technique is a flexible approach capable of working with many games and applications not designed to allow data access through APIs or game logs. The approach further addresses the problem of accessing otherwise inaccessible data within a game (Category 3 games, Section 2.4). It also allows retrieval of higher quality data in cases where the application provides noisy replay data.

Results are contrasted with game state traces generated by file based replay parsers. Replay parsers work on game log files which record all the layers interactions of a player with the game. Because they are file based and don’t involve execution of the actual game, replay parsers can operate extremely fast. However, the files are designed to be processed by the game itself as a type of ‘replay script’, with the recorded commands executed again by the actual game. Given a deterministic relationship between commands and the actions of the game, the recorded game can be played back.

The issue is that many player actions may have no effect on the game. For example, the player may issue several build commands, when there are only resources available to complete one. The later commands are issued but have no effect on the game. While the game would process these commands correctly (i.e. ignore them), a replay parser encountering them in a game log may not be sufficiently
sophisticated to recognise that they cannot be completed, and therefore records several units instead of one.

Image capture has an advantage over replay log parsers in this regard, because it is possible to observe what actually happens within the game. However, this capability comes at a cost relative to file based replay parsers. It requires significantly more CPU resources to identify images from the screen relative to reading data from a file. Additionally, it takes longer to retrieve data using image capture as the data must be played back on the screen and the processing rate is thus limited to the refresh rate of the screen. This comes as a once off cost as screen capture results can be stored in files for future quick access. Screen capture also has potential inaccuracies which are evaluated later in the chapter.

Screen capture was used to retrieve game state data from Starcraft 2. This was used during preliminary investigations of the Case Based Reasoning (CBR) approach discussed in Chapter 6 to provide data for the memory component of the Search and Recall technique.

The work in this chapter work was published as the following conference paper:

### 3.2 Overview of Screen Capture Technique

The screen capture system models a human observer tracking and recording changes in a game. It identifies areas of the screen which display information relevant to game state and then monitors changes in those areas, interpreting them in terms of the game state.
Initially, the area of the game window in which the relevant information will appear is specified. Then, patterns showing all the information to be recognised in that area of interest are recorded as a set of templates. All templates have the same dimensions to simplify and speed up matching. After all templates have been loaded into the system, Principal Component Analysis (PCA) (Jolliffe, 1986) is used to compress each set of templates down to 30 dimensions per template. Each dimension represents a pixel colour channel value bound between 0 and 255. For example, a single production icon is made up of 37*25 pixels. Each pixel has a RGB (Red, Green, Blue) value, giving a total of 37*25*3=2775 dimensions.

Using PCA these 2775 dimensions are compressed to 30. The method below describes how PCA based templates for production icons related to the Terran race in Starcraft 2 are constructed offline.

**Algorithm 1** Screen Capture Template Matching

1. PCA_DIMENSIONS = 30
2. procedure PRE-PROCESS CLASS TEMPLATES(Class_Icons)
3.   \[ Class\_PCA\_Matrix \leftarrow \text{calculate}\_PCA(Class\_Icons,\text{PCA\_DIMENSIONS}) \]
4.   for all 2D\_Icon in Class\_Icons do
5.     Compressed\_2D\_Icon = transform(2D\_Icon,Class\_PCA\_Matrix)
6.     Class\_Compressed\_Icons.append(Compressed\_2D\_Icon)
7.   CLASS\_THRESHOLD = Max\_Distance\_Between\_All\_Icons(Class\_Compressed\_Icons)*1.1
8. procedure CLASSIFY SCREEN CAPTURED 2D ICON(new\_Icon, Class\_Compressed\_Icons)
9.   compressed\_New\_Icon = transform(new\_Icon,Class\_PCA\_Matrix)
10.  minimum\_Distance = CLASS\_THRESHOLD
11.  matching\_Template = Null
12.  for all Class\_Pre\_Icon in Class\_Compressed\_Icons do
13.     dist = distance(compressed\_New\_Icon,Class\_Pre\_Icon)
14.     if dist<minimum\_Distance then
15.        minimum\_Distance = dist
16.        matching\_Template = Class\_Pre\_Icon
17.     if minimum\_Distance >= CLASS\_THRESHOLD then
18.         return No\_Match
19.     else
20.         return matching\_Template
3.2. Overview of Screen Capture Technique

1. Load the 71 Terran race production icons as X.

2. Calculate the covariance matrix over all icons using Equation 3.1.

3. Calculate the eigen vectors and values for the covariance matrix.

4. Select the largest 30 eigen values and associated eigen vectors.

5. Form the transform matrix T from the 30 selected eigen vectors.

6. Transform each of the 71 Terran templates from a 2775 dimensional vector to a 30 dimensional vector to be used as a matching template, using the transformation matrix T in Equation 3.2.

$$Covariance_{ij} = \frac{\sum_{k=1}^{n} (x_{ik} - \bar{x}_i)(x_{jk} - \bar{x}_j)}{n - 1}$$ \hspace{1cm} (3.1)

$$X_i^{30} = T \cdot X_i^{2775}$$ \hspace{1cm} (3.2)

The process below illustrates how a captured icon is then matched against a Terran race production icon template.

1. An icon is captured from the game.

2. The icon is reduced from 2775 dimensions to 30 using the same formula and transform matrix as above.

3. The reduced icon is then compared with the 30 values of each of the compressed production icon templates. The matching template is the template with the smallest Euclidean distance between the reduced icon and production icon template.

4. If the distance is less than or equal to a threshold value, the captured icon is matched.
5. If the distance is greater than the threshold then no template match is recognised.

Reducing the dimensionality of the icon descriptors from 2775 to 30 greatly speeds up comparisons between templates and captured icons. This reduction in dimensionality of the template description reduces the computational cost of comparisons for each template and facilitates real-time analysis of captured images. When the application is run as a replay for game trace retrieval, the games window contents are captured via the Windows API and stored as OpenCV images as often as the refresh rate allows. The images are then decomposed into the identified areas of interest such as the game timer, player’s production icons, progress bars, and resource supply. Game specific heuristics are then used to extract information from the screen using template matching. The extracted information is then analysed to monitor game state events, and the processed game state information is then stored as a game trace.

The screen capture system can be summarised as taking the following steps:

1. Load pre-labelled templates.

2. Decompose templates into basic descriptors.

3. Open the game to be analysed associated replay file.

4. Capture the game window using the Windows API.

5. Store and decompose the window’s contents into areas of interest.

6. Match templates against areas of interest using a multi-threaded framework.

7. Process the results and store the resulting game trace.

8. Repeat from step 3 to analyse further games.
3.3 Avoiding Noisy Game Traces

3.3.1 Experimental Method

An experiment was conducted to evaluate how the screen capture system performs in capturing a build order. The results were compared with those produced by the established build order analysis tool Sc2Gears (Belicza, 2012). Sc2Gears applies the user interaction approach to analyse build orders. This approach extracts game state information based on user commands recorded in a game log. Both systems were tested using a set of 100 public Starcraft single player versus single player ladder games. Comparisons were made only on the first 10 minutes of gameplay so that replays of diverse lengths would not affect the results significantly. Each of the 100 games was also analysed by a human to generate a ground truth set of build orders. The accuracy for the automated systems was calculated as the percentage of matching build order steps compared with the human verified sequence. Accuracy is defined as the ratio of the number of matching steps to the total number of steps recorded by the data extraction method as shown in formula 3.3. For example, in Table 3.2, SC2Gears records 8 steps, but only 4 of them match the actual steps observed by a human, giving 50% accuracy. On the other hand, the screen capture approach records 4 steps all of which match the observed steps giving 100% accuracy.

\[
\text{Accuracy} = \frac{\text{MatchingSteps}}{\text{TotalRecordedSteps}}
\]  

(3.3)

3.3.2 Extracting Starcraft 2 Build Orders

Figure 3.1 displays the replay interface in Starcraft 2 that was used to retrieve game traces. Before starting the process, the system must be aware of where to
look for which templates. The templates are stored in sets, one set for each of the production icons of each player selectable race, and an extra one for other GUI elements. Separate sets of icons for each race reduces the number of comparisons necessary as a player can only produce items for their chosen race.

To retrieve a build order, the HUD icons representing a player’s production queue are identified using the library of PCA refined descriptors. The top left-hand corner of Figure 3.1 shows seven units/buildings in production. These production icons are displayed continuously as a HUD in the same location of the screen no matter where on the game map the camera is located. Each item of production shows an identifying image, a number showing how many units are being produced simultaneously, and a green progress bar reflecting the completion percentage of that item. Each different production icon indicates that a build queue is active within that area of interest. In this case, it can be seen that 4 build queues are active for player 1, and 3 are active for player 2. The icon positions are then posted to different worker threads which compare the captured image with an assigned template set. Figures 3.2 and 3.3 show the matching templates used to identify production icons collected from the scene shown in Figure 3.1. Each template is labelled with the name of the production icon.

After identifying the production icon, the game trace heuristic then finds the number on each template as shown in Figure 3.4. Numbers are identified using a relatively naive yet accurate method. Because numerals are imprinted against a production icon’s image, they contain a small amount of noise. The noise is reduced by only accounting for pixels that are very close to white. Then the filtered image is compared against a set of number templates and the most similar template is selected as the matching number. Following this process identifies the digit shown Figure 3.4 as matching the template shown in Figure 3.5.
3.3. Avoiding Noisy Game Traces

**Figure 3.1:** Sample screen capture

**Figure 3.2:** Player 1 - Matching production templates

**Figure 3.3:** Player 2 - Matching production templates

**Figure 3.4:** Digit with noise

**Figure 3.5:** Pre-labeled Digit (Matching Template)
The completion percentage shown in the production icon is then determined (Figure 3.6). For this, a threshold check is performed for predefined pixel values along the length of the progress bar, returning when an empty pixel is found.

To receive the game time of the game environment each symbol is shown in Figure 3.7 is matched, thus returning a time of "3:45 / 23:08". The returned time is used to identify the end of the replay and readies the next replay to be opened for processing.

Since each template comparison is independent, all template comparisons can be run in separate threads. Once all threads have finished analysing each real-time acquired production icon image, the information is used to update each player’s build order along with the game state, and the game time at which the image was retrieved.

As each player’s build order is updated, it is possible that a previously recorded production item is cancelled. If a production item is not listed but was less than 97% complete when last identified then that item is assumed to have been cancelled and is removed from the recorded build order. Within a game of Starcraft 2, this can occur at any point in time when a user selects a production item and cancels it. A cancellation is also noted if the number of items listed within the production icon drops while the current completion percentage is under 97%. This leads to a potential flaw in the current game trace heuristic. If a production item
is almost complete, then any number of production items of the same type can be
cancelled, and they will be falsely recorded as completed. In practice, this rarely
occurs.

When a new production item type appears or the production count increases then
the game trace heuristic appends that item to the build order. If the number of
items in production recorded by a production icon number remains the same for
longer than the time to create that item, then another production item of that
type is appended to the build order. This deals with the case of when a series
of probes/workers are queued. Since they are created one at a time, a constant
production count of 1 appears over an extended period. Thus, by keeping track
of how long it has been from when a production icon first appears, it can be de-
termined when an item repeats production. The exception is when production is
halted or paused. However, halting or pausing can be detected when the progress
bar is halted.

After a game has completed, each players’ build order is recorded to file, and the
next game is opened and the process repeated. The replay interface is controlled
by sending Windows API keyboard messages to Starcraft 2 to display the produc-
tion icons and accelerate the playback. The replay playback can be accelerated to
the maximum of eight times the normal playback rate allowing rapid analysis of
multiple recorded games.

### 3.3.3 Results

Table 3.1 shows that the screen capture technique was able to significantly re-
duce the number of errors in calculated build orders compared with an analysis
based on raw user interactions. The screen capture system still generated a small
Table 3.1: Data Retrieval - Error Rates

<table>
<thead>
<tr>
<th>Error</th>
<th>Screen Capture</th>
<th>Sc2Gears</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>0.39%</td>
<td>30.71%</td>
</tr>
<tr>
<td>StdDev</td>
<td>0.96%</td>
<td>27.75%</td>
</tr>
</tbody>
</table>

Table 3.2: Example Game Trace

<table>
<thead>
<tr>
<th>Sc2Gears</th>
<th>Screen Capture</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Probe</td>
<td>1. Probe</td>
</tr>
<tr>
<td>2. Probe</td>
<td>2. Probe</td>
</tr>
<tr>
<td>3. Probe</td>
<td>3. Probe</td>
</tr>
<tr>
<td>4. Probe</td>
<td>4. Pylon</td>
</tr>
<tr>
<td>5. Probe</td>
<td></td>
</tr>
<tr>
<td>6. Probe</td>
<td></td>
</tr>
<tr>
<td>7. Pylon</td>
<td></td>
</tr>
<tr>
<td>8. Pylon</td>
<td></td>
</tr>
</tbody>
</table>

number of errors in cases where actions were cancelled on the last frame (thus incorrectly appearing to have been completed).

Table 3.2 shows an example of an opening build order extracted using screen capture compared with one using Sc2Gears from the same game. The extracted information is significantly different. Sc2Gears incorrectly identifies the creation of three additional probes and an additional pylon. In this case, the player requests production of additional Probes without the necessary resources, a situation that can only be determined by running the game replay. The extra pylon identified by Sc2Gears was the result of the player ordering construction of a pylon and then moments later changing the location of its construction. These errors highlight the issues encountered when using user interaction methods to extract game traces.
3.4 Retrieving Inaccessible Data

The screen capture technique was also applied to another game; Defence of the Ancients 2 (Dota 2), to test the real-time capabilities of the screen capture framework, and its capacity to generalise beyond Starcraft 2. This section re-enforces the general utility of the screen capture framework in retrieving data from a 2D display interface. The data from the Dota 2 interface was retrieved without error, and thus no comparison against other methods is given. Instead, the experiment with Dota 2 shows the flexibility and versatility of the screen capture approach.

Dota 2 is a multiplayer online battle (MOBA) game that involves two teams of 5 players. Each player must pick a hero, and after a hero is selected and locked in, it can not be picked by any other player. Players can select the hero they intend to pick before locking it in, and this is referred to as shadow picking. A shadow pick will only display to the allied team and is important in influencing the heroes other members of the team select.

Heroes fall into general categories based on their abilities and how they interact with other heroes within the game. The picking process leads to a diverse set of combinations that can be formed between the two teams. However, some of these combinations are weaker than others due to the interaction of hero’s strengths and weaknesses. Each hero can have synergies with certain allied heroes, exploit weaknesses in particular enemy heroes, or do both. Thus, it is an interesting problem to see how players adapt their choice of hero during the 1-minute picking phase. It is also interesting to see whether these picks can be used to predict the winning team and what rate of success they might have.

In Dota 2, there is much interest in the real-time capture of game actions since such a capability offers the potential to support real-time guidance on hero selection. It also provides information useful for calculating the likelihood of final
outcomes. Screen capture can potentially achieve this while user interaction logs are available only after a game has ended.

Figure 3.8 shows a standard Dota 2 ‘all pick’ mode selection screen. All players have locked in their hero choices except for the player shown on the upper left. This player’s portrait is rendered in grey scale to show that the depicted hero has only been shadow picked. During the picking phase, the screen capture framework is used to identify which heroes have been locked in or shadow picked. This data is then analysed and a guiding heuristic referred to as the ‘Team Win Rate Difference’ (TWRD) is calculated based on statistics gathered from hundreds of thousands of Dota 2 games. The TWRD is calculated as follows:

1. First, the five allied heroes, five enemy heroes, winning side, and game time is extracted from every game in the database of recorded Dota 2 games.

2. Games are then filtered to group games that complete within successive time windows of interest.

3. Then, the average win rate for every possible pair of heroes is calculated based on data for games within each time window. This is referred to as the
3.4. Retrieving Inaccessible Data

HeroPairWinRate for a particular time window, and is the probability that a team will win within that time window given that the given pair of heroes exist on that team.

4. Then, for a specific combination of five heroes, all possible pairs of heroes are extracted. This gives a total of 25 possible pairs of heroes for each 5 hero team.

5. The HeroPairWinRates for each possible pair of heroes within a team is then summed and normalised to give the overall Team Win Rate heuristic, as shown in Equation 3.4.

6. The final Team Win Rate Difference is then calculated as the difference between the TWR for the allied (i.e. the player’s) team and the TWR for the opposing team, as shown in Equation 3.5. Results will vary between -1 and 1 with larger positive results indicating more probable wins for the allied team, and negative results indicating a probable loss.

\[
Team_{Allied\_WinRate} = \frac{\sum_{\text{Hero}_1}^{\text{Hero}_5} \sum_{\text{Hero}_1}^{\text{Hero}_5} \text{HeroPairWinRate}(\text{hero}_x, \text{hero}_y)}{25}
\]  

(3.4)

\[
Team_1\_WinRate = Team_{Allied\_WinRate}^{Team\_1} - Team_{Allied\_WinRate}^{Team\_2}
\]  

(3.5)

The program then displays the win rate for any point in time during a game involving the selected teams as shown in Figure 3.9. This graph can be used loosely to identify when one team is stronger than another and can be utilized as an indicator for players to become more aggressive within the favoured time.
zones. It can also be used by lower skilled players to help better identify hero picks that complement their team and to see what effect their pick would have on the progress of the game. Figure 3.9 shows that the Enemy Team has a small advantage that decreases over time until around the 60-minute mark, at which point My Team increases substantially in strength.

![Figure 3.9: Dota 2 Predicted Game Balance](image)

### 3.5 Discussion and Further Work

The Starcraft 2 experiment shows that the screen capture approach can help generate more accurate build order analyses than conventional systems based on logs of player actions. Its application to analysing hero selection in Dota 2 shows that the principles can be applied generally to any game with identifiable 2D icons, and for any analytical purpose, using different sets of image templates and different analytical heuristics. The technique can be applied to almost any application where a streaming 2D display record is available. Furthermore, no access to game code or proprietary APIs is required. This opens up opportunities for data collection and analysis of previously inaccessible games and other applications. The high performance provided by the simplified PCA based image descriptors
and parallel template matching allows the development of real-time in-game decision support systems, once again without access to game code or proprietary APIs.

The screen capture system takes advantage of using the game display to retrieve actual game events while user interaction logging methods can result in noisy data that can detrimentally affect further analysis.

While analytic techniques relying on replays to retrieve game data have to wait until a game has been played and recorded before analysis can be applied, a screen capture system can be used to analyse live games, allowing interested parties to use the data in prediction systems or other applications.

However, currently screen capture has only been applied to applications where the state is represented with scale and rotation invariant 2D images. There would be considerable challenges in applying the technique to applications that display their state in 3D.

### 3.6 Conclusion

Screen capture for building accurate game traces data offers great advantages to researchers and applications looking to gather data from complex environments with 2D displays. The framework demonstrated significant improvements in Category 2 (Section 2.4) data retrieval problems concerned with avoiding the use of provided noisy game trace data. It also addressed Category 3 (Section 2.4) data retrieval problems concerned with accessing otherwise unavailable data within a game. The system is flexible and more accurate than user interaction logs for such applications.
The data retrieved through these experiments was used in preliminary experiments related to building a CBR database for Starcraft 2. However, due to the issues involved in creating an agent for Starcraft 2, a decision was made to use Brood War as the research platform for the rest of the thesis, and the image capture technique was not further developed.
Chapter 4

Path Finding

4.1 Overview

Later search related work in this thesis (Chapter 6 and Chapter 7) depend on high fidelity simulators used to model Brood War. Pathfinding is a critical processing bottleneck for such simulators and improving the speed of pathfinding indirectly improves the quality of search based solutions by allowing more samples to be evaluated during restricted time windows.

This chapter investigates fast algorithms for fixed grid path finding in dynamic environments. This is because the Brood War (the chosen RTS AI research platform) uses such a fixed grid, and pathfinding represents the largest processing bottleneck for accurate simulations involving collisions and terrain. The need for such accurate simulations is established in Sections 2.6 and 2.9, and the algorithms developed in this Chapter and Chapter 5 are applied in both Chapters 6 and 7. The work in this chapter provides the basis of fast pathfinding involving collisions and terrain on which the AI framework developed in Chapters 6 and 7 depend.
Pathfinding is important for RTS AIs because many strategic and tactical decisions depend on the fast and accurate evaluation of alternate paths. Path optimality and collision detection particularly in relation to dynamic obstacles can determine the difference between a successful and a failing strategy. RTS agents require pathfinding algorithms that work as fast as possible within a dynamic environment (e.g. obstacles can be removed or added, units can be created or die, bridges over rivers can be destroyed or built).

The work in this thesis on simulation-based RTS agent AI requires extremely fast pathfinding within dynamic environments to be performed multiple times to evaluate different possible scenarios. There are many pathfinding algorithms as shown in Section 2.7, but most of these are unsuitable for the needs of RTS agents. Cache based pathfinding algorithms can be extremely fast, but lose much of their advantage in dynamic environments (see discussion in Section 2.7). Algorithms such as Compressed Path Database (CPD) store all possible paths in a database (Botea, 2011; Botea, 2012), while SubGoal Fast (Uras, Koenig, and Hernández, 2013) performs layers of pre-processing based on decomposition of a map into different areas. Dynamic environments invalidate pre-processed data constantly by removing, adding or modifying the environment which then requires rebuilding of pre-processed data. Rebuilding the invalidated data can outweigh the search speedup provided by pre-processing. Therefore, a fast, online search algorithm with very little pre-processing is required.

This chapter tests the algorithm developed on a variety of open to dense maps to give a wider application and performance analysis. The map data set used also provides a direct comparison with the Jump Point Search (JPS) algorithm which this work extends (Harabor and Grastien, 2011). JPS is a fast, online algorithm which works by eliminating most map nodes from evaluation during path expansion (see Section 2.7.5).
This chapter presents an improvement to the basic JPS algorithm which significantly improves the speed of search. The work presented in this chapter has been published in the IEEE journal Transactions on Computational Intelligence & AI in Games.


Further work making additional optimizations is presented in Chapter 5, and was presented at the 2016 conference on Artificial Intelligence and the Simulation of Behaviour.


The two sets of algorithms were evaluated respectively in the 2013 and 2014 Grid based Path Planning Competition.


### 4.2 Implementations

The basic mechanics of the JPS algorithm are discussed above in Section 2.7.5. In essence, JPS prunes nodes from possible search paths by identifying ‘Jump Points’ where possible direction changes in an optimal path can occur. In dynamic environments, the position of jump points change and this requires recalculation of jump points in the areas of a grid affected by change. If changes are continuous, any overhead in the process of re-evaluating optimal paths becomes
a performance issue for real-time path planning. This is because the basic JPS algorithm discovers jump points by iteratively extending paths in the vertical and horizontal directions, and this iterative method of discovering jump points becomes a processing bottleneck.

However, boundaries that constrain paths form a relatively small subset of the nodes on a grid. Boundaries that dynamically change position form an even smaller subset, and it is a relatively cheap operation to update a table recording the positions of obstacle boundaries within a grid.

The proposed new algorithm, Boundary Lookup JPS (BLJPS) exploits the observation that jump points for horizontal and vertical path extensions occur only due to the effects of obstacles or the position of the final target node. Obstacles are defined by the position of their boundaries, and a list of the boundary nodes within a grid forms a relatively small table. By pre-processing the grid and recording the positions of these boundaries in a table, the search speed for a jump point in a particular direction can be greatly increased by simply looking up the next boundary occurring in that direction within the table. BLJPS implements this approach and is more efficient in searching for a path than standard JPS.

BLJPS uses the same directional expansion method as JPS. However, BLJPS does not iterate over each adjacent cell in a vertical or horizontal direction. Instead, it looks up whether there is a boundary or obstacle in those directions and immediately evaluates whether the current node is a jump point or not. Vertical (North and South) directions work on the same principle as horizontal (East and West) directions. Thus, East and West directions use the horizontal boundary lookup table, and the North and South directions use the vertical boundary lookup table.

BLJPS identifies jump points in horizontal or vertical directions when it finds a boundary that is further in the given direction than the reopen value (where
4.2. Implementations

the map changes from closed to open) for either of the neighbouring rows or columns. To identify jump points that are not accessible via the cardinal directions the search must expand out diagonally. For each step taken diagonally, the relevant cardinal directions are checked for a jump point. Each diagonal step then checks if it has intersected the goal or has any forced neighbours that would terminate its expansion.

Table 4.1 shows the lookup table corresponding to Figure 4.1. The values in Table 4.1 record where cells toggle between open and closed states, starting from the Western (or Northern) boundaries. Rows such as 1 through 5 are completely open (i.e. have no obstructions), resulting in the full width of the map (N) being recorded as the only horizontal boundary for those rows. Rows 6 through 9 have a boundary (i.e. change from open to closed) at K, and this is recorded as the first entry for those rows. Each row then reopens (i.e. changes from closed to open) on cell L, and this change is also recorded in the table. The width of the map (N) is again recorded as the final boundary from open to closed. Thus, every odd entry within the table is a location of a boundary, and every even entry is the location of a reopening.

A simple example of using the lookup table would be to test whether (A8) can move directly east to (H8). The horizontal boundary lookup table entry corresponding to row 8 (Cells 6 to 9) is searched to find an entry that is greater than or equal to A. This returns K. K is greater than (further East than) H and therefore (A8) is passable to the East to (H8). However, if the reachability of M8 from A8 were tested, the value K would be less than M, and so the test would determine that (M8) is not reachable Eastwards from (A8). Both determinations take only one lookup, whereas the basic JPS algorithm would have taken ten iterations to achieve the same result in each case. This demonstrates the power and efficiency of the boundary lookup approach.
Chapter 4. Path Finding

<table>
<thead>
<tr>
<th>Horizontal Boundary Lookup</th>
<th>Vertical Boundary Lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cells 1 to 5: (N)</td>
<td>Cells A to J: (10)</td>
</tr>
<tr>
<td>Cells 6 to 9: (K,L,N)</td>
<td>Cell K: (6)</td>
</tr>
<tr>
<td></td>
<td>Cell L, M, N: (10)</td>
</tr>
</tbody>
</table>

Table 4.1: Boundary Lookup Table – Used to determine the nearest obstacle boundary along either the horizontal or vertical axis.

Figure 4.1: Boundary lookup example on a 13x9 uniform grid. Blacked cells K6-K9 are obstacle boundaries. P1 is the starting point. P2 and P3 represent jump points. P4 is a forced neighbour of P3.
In terms of identifying jump points, take the example of a node expansion starting at (A5). (A5) on row 5 has a boundary to the East of N, which is greater than the reopen value of row 6 below it of L. This indicates that a jump point has been found at K (L-1=K) on row 5 or P3. If the node expansion is started from M5 in the Western direction, then the first boundary encountered is the Eastern boundary of the grid at A-1. The row below (row 6) encounters K as a western boundary. Row 6 reopens at J which is not as far in the western direction as the A-1 boundary.

BLJPS is even more efficient in terms of diagonal expansions since the basic JPS algorithm requires iterative expansions in both cardinal directions for each diagonal step. The following paragraph demonstrates how JPS finds P2 as a jump point from P1 in the North-East direction.

In basic JPS, each diagonal move requires an iterative search in the horizontal and vertical directions which in this case incurs 9+8+7+6+6 horizontal checks and 8+7+6+5+4 vertical checks. This generates a total of 66 iterated checks to move four diagonal steps. BLJPS performs only one check per axis to determine whether a node is a jump point although that check does involve three lookups. For a horizontal axis check, BLJPS looks up the distance of boundaries on the row above, the same row, and the row beneath the queried node. In this case, this incurs 3+3+3+3+3 horizontal checks and 3+3+3+3+3 vertical checks, resulting in a total of only 30 checks. The advantage is more pronounced over longer distances but can become a disadvantage on very constricted pathways.

The following example demonstrates how BLJPS moves from P1 to P4 on Figure 4.1 using diagonal expansion. Each cardinal direction check is displayed as a dotted line. The search starts with P1 checking for Jump Points in all eight directions. In this case, only the East, North and North-East directions are accessible. Both North and East directions fail to return a jump point, encountering either the edge of the map or an internal boundary. BLJPS then attempts to expand to
the North-East moving to location (B8). It then repeats the cardinal jump point checks relative to a North-East expansion (North and East). Again, these checks fail to find jump points along either axis. This process is repeated until the diagonal step reaches a boundary or a location from which a jump point can be found. In this case, the North-East expansion is repeated until location P2 is reached. At this point, the Eastward cardinal direction check returns a jump point at P3, because row 5 reopens at a smaller distance than the first boundary discovered in row 5. P2 is then added to the open list (for node expansion) with the direction North-East, and the node expansion from P1 is terminated.

The top node within the open list is then popped, returning P2. P2 is expanded in the NE natural neighbour directions (N, NE, E) in addition to any forced neighbour directions (in this case none). The North and North-East searches find no further jump points. However, the Eastward direction check returns P3 as a jump point. P3 is then expanded in the Eastern direction encountering the eastern map boundary. In addition, P3 is expanded to the South-East as a result of the forced neighbour in the South-East direction. Stepping in the South-East direction identifies the goal node P4 has been reached. Thus, the path (P1, P2, P3, P4) is finally returned.

BLJPS pseudo code for finding a jump point in a given direction is shown in Algorithm 2 below. The pseudo code is designed for giving an understanding of the main algorithm, and many details of low-level functions are omitted. For a more detailed and specific implementation see the source code in C++ and data used in this work provided at (Traish, 2017a) (https://github.com/narsue/Dynamic_BLJPS).
4.2. Implementations

Algorithm 2 BLJPS

1: procedure BLJPS_JUMP(cp: current position, d: direction, g: goal)
2:   np ← step(cp, d, 1)
3:   if not is_passable(np) then return Null
4:   if np == g then return g
5:   if has_forced_neighbour(np, d) then return np
6:   if d is diagonal then
7:     for all cardinal direction c in d do
8:       if cardinal_jump(np, c, g) ≠ Null then return np
9:     return BLJPS_Jump(np, d, g)
10:   else
11:     return cardinal_jump(np, d, g)
12: procedure CARDINAL_JUMP(cp: current position, cd: cardinal direction, g: goal)
13:   np ← step(cp, cd, 1)
14:   nl ← step(np, left(cd), 1)
15:   nr ← step(np, right(cd), 1)
16:   if not is_passable(np) then return Null
17:   if np == g then return g
18:   cc ← next_Closed_Boundary(np, cd)
19:   if direction(cp, g) == cd and distance(cp, g) ≤ distance(cp, cc) then
20:     return g
21:   jumpPoint ← null
22:   lo ← next_Open_Boundary(nl, cd)
23:   if lo ≠ null and distance(np, cc) ≥ distance(nl, lo)-1 then
24:     jumpPoint ← step(np, cd, distance(nl, lo) − 1)
25:   ro ← next_Open_Boundary(nr, cd)
26:   if ro ≠ null and distance(np, cc) ≥ distance(nr, ro)-1 and (jumpPoint == null or distance(nr, ro) < distance(nl, lo)) then
27:     jumpPoint ← step(np, cd, distance(nr, ro) − 1)
28: return jumpPoint

4.2.1 Updating Invalidated Pre-Processed Data

JPS+ (Harabor and Grastien, 2012) addresses the iterative search processing bottleneck in JPS by pre-processing the grid to identify all jump points to generate a jump-point lookup table, resulting in the fast determination of whether a jump
point exists in a particular direction. However, any changes in boundary positions invalidate parts of the jump-point lookup table. In a dynamic environment where obstacles are being removed, inserted or otherwise moved or modified, pre-processed data for areas affected by any change require updating. This can greatly reduce the advantage of pre-processing methods since paths through affected areas need to be recalculated. The recalculation is strongly affected by the underlying search algorithm.

The main advantage of BLJPS is realised in dynamic environments where the positions of obstacles change. The cheap update of an obstacle boundary table combined with the elimination of the iterative search procedure makes BLJPS more efficient than attempting to update jump-point lookup tables for invalidated areas of the grid as necessary in JPS+.

In the following section, the effects of dynamic environments on the search cost of JPS+ and BLJPS variants are considered. The effects of randomly adding and removing varying numbers and sizes of obstacles to several different map sets are evaluated. All the algorithms tested use one or both of the following two pre-processed data sets.

1. Jump Point lookup pre-processed data is used by algorithms with the postfix +. This data allows the algorithms to go to the next jump point as given in the lookup table rather than searching the map for it.

2. Boundary lookup pre-processed data is used by BLJPS algorithms. These algorithms store a table of horizontal and vertical boundaries to be used in search methods.

Two approaches to updating pre-processed Jump Point lookup data affected by dynamic obstacles are examined.
4.2. Implementations

1. Flush Update (FU) – In this approach, all pre-processed data (jump point information) for the entire grid is erased, and all paths are recalculated as needed.

2. Partial Update (PU) – In this approach, only pre-processed data for areas affected by the appearance or disappearance of an obstacle are cleared and recalculated.

In the Partial Update (PU) method, grid cells that potentially hold invalid pre-processed jump point data after a changed obstacle has affected the environment are identified. The partial clear is a fill operation that works in reverse to the JPS direction of search. The obstacle is first dilated by 1 in each cardinal direction to ensure that the following path expansion captures all relevant jump points. PU expands out from the dilated obstacle in the vertical and horizontal directions. For each cell expanded it then clears along the diagonals (E.g. for a Node expanded to the East, all cells along that Eastward path are then expanded along the North East and South East directions). This process is repeated until the expansion terminates at a boundary. Each clear operation is flagged in lookup so that the same paths are not needlessly cleared several times.

Figure 4.2 shows the extent of invalidated areas due to the introduction of two 3x3 obstacles. Blue represents the dilated obstacle, green shows the expansion in the cardinal directions, and red shows the invalidated area due to secondary expansion along diagonal directions. All jump points within the invalidated area need to be discarded and new jump points calculated.

Modifications to boundary lookup tables for BLJPS variants are far less extensive. Only the rows and columns corresponding to where the obstacles are added or removed need to be updated. For each row and column that the obstacle covers,
Chapter 4. Path Finding

Figure 4.2: A) Partial Update (PU) displaying invalidated areas after adding two obstacles. B) New obstacle area. C) Cardinal direction invalidated area. D) Diagonal stepping invalidated area. White areas are unblocked and grey shaded areas are blocked.

...the respective table lists are cleared. When all obstacles have been added or removed the lists that were cleared are recreated using the updated grid data. For example, inserting a 3x2 obstacle into the map will clear three vertical lookup lists and two horizontal lists. Partial Update then rebuilds the cleared lists using the same method used in the pre-processing stage to identify boundary positions.
4.3 Method

The presented experiments evaluate the strengths and weaknesses of BLJPS and compare its performance against JPS and JPS+ in static and dynamic environments. The experiments of (Harabor and Grastien, 2011) are replicated, using the datasets freely available from (Sturtevant, 2013). Each map includes a set of predefined problem paths that are used to benchmark the algorithms. Table 4.2 lists the maps used in the experiments.

Two main sets of experiments were conducted. First, the performance of JPS, JPS+ /BLJPS+, and BLJPS were evaluated in a static environment. In static environments, JPS+ and BLJPS+ are effectively identical as they use the same search algorithm and data. On each map, all paths provided in the dataset are executed giving paths of varying lengths. Optimal paths between start and end points are then generated repeatedly until each has been calculated at least 100 times. This method provides an accurate evaluation of the time taken to find a path. The time taken to find a path is then calculated as the average search time over the number of iterations.

Next, the performance of each algorithm was evaluated in a dynamic environment. The experiments were run on the same maps as for the static environment.

<table>
<thead>
<tr>
<th>Map</th>
<th>Number of Maps</th>
<th>Map Size</th>
<th>Average number of paths per map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baldur’s Gate (BG)</td>
<td>75</td>
<td>512x512</td>
<td>1242</td>
</tr>
<tr>
<td>Dragon Age Origins (DAO)</td>
<td>156</td>
<td>Smallest: 28x22 Largest: 104x1260</td>
<td>1022</td>
</tr>
<tr>
<td>Rooms (Rooms)</td>
<td>10</td>
<td>512x512</td>
<td>1928</td>
</tr>
<tr>
<td>Adaptive Depth (AD)</td>
<td>12</td>
<td>100x100</td>
<td>1000</td>
</tr>
</tbody>
</table>

Table 4.2: Maps used in experiments. DAO maps vary in size, ranging from the smallest and largest map sizes given. Each map is associated with a varying number of paths, the average for map each set is given.
experiments. However, for the dynamic experiments several randomly placed obstacles are added to the map before each run. The randomly placed obstacles cause optimal paths to vary between each run and simulate a dynamic environment. Once again, each path search is run 100 times and the average duration of a path search is recorded.

The Jump Point Search (JPS) algorithm used is an optimised version of the code provided from (Astar-jps) and the results obtained are comparable with those obtained in (Harabor and Grastien, 2011). The implementation of JPS+ is based on the description given in (Harabor and Grastien, 2012).

A number of parameters affect the performance of the algorithms within the dynamic maps. These parameters include:

1. The number of obstacles.
2. The size of the obstacles.
3. The number of paths searched before varying the obstacle configuration.

Varying the number of paths searched before modifying the environment simulates varying the number of agents that may be moving on a map. The effects of using full flush and partial update approaches to modifying the lookup tables utilised by JPS+ and BLJPS are also evaluated. The only setting that changes between the dynamic map experiments is the number of paths run before modifying the map. This will weigh each algorithm’s ability to utilise pre-processed data and their cost in rebuilding it.

### 4.3.1 Algorithm Summaries

Experiment 1 tested a range of algorithms on static maps to evaluate the strength of the BLJPS algorithm against other established algorithms. Three algorithms
were selected from the Grid-Based Path Planning Competition (GPPC) (Sturtevant, 2013). These were selected on the basis that each had an online search component, they returned optimal paths and that they worked on the majority of maps.

1. Pseudo Priority Queues (PPQ) (Guivant, Seton, and Whitty, 2012): An optimised A-Star algorithm that provides a faster way for quickly accessing best node on the OPEN list.


3. Subgoal Graph Optimal (SubGoal) (Uras, Koenig, and Hernández, 2013): This algorithm uses a subgraph of regions with perfect heuristics. This variant was chosen for its desirable ratio of search speed to allocated memory overhead.

4. JPS (Harabor and Grastien, 2011): JPS exploits path symmetry in uniform-cost grid maps to prune paths at runtime.

5. JPS+ (Harabor and Grastien, 2012): This algorithm uses JPS to pre-process the map and stores all jump points in a lookup table for usage during search.

6. BLJPS: Substitutes the iterative evaluation of neighbouring nodes during search with a direct lookup within a Boundary Lookup (BL) table.

7. BLJPS+: Uses BLJPS to pre-process the map storing jump points for lookup during search.

BLJPS+ and JPS+ don’t differ in online search time for a static map and as such are evaluated as a single method in those tests.

The experiments 2 and 3 on dynamic maps focused on JPS and BLJPS variants. These experiments break JPS+ and BLJPS+ into Partial Update (PU) and Flush
Update (FU) to evaluate how they utilise and rebuild pre-processed data. Unfortunately, SubGoal doesn’t support dynamic maps. The static map results for PPQ and NovelAStar indicated that further comparisons would not be of value.

Results for varying path lengths showing the speedup achieved by the various algorithms relative to the processing time taken by the $A^*$ algorithm are shown below in Figures 4.3 through 4.10.

### 4.3.2 Static Map Experimental Results

Figures 4.3 through 4.10 show the different algorithms performance in static environments. Due to the speed differences of the algorithms, the Figures on the left represent the four faster algorithms while the figures on the right represent the slower three. JPS is included in both to establish a point of reference. The figures show that JPS+/BLJPS+ is superior to JPS/BLJPS on all maps. BLJPS out-performs basic JPS on BG, DAO, and AD maps while basic JPS out-performs BLJPS on the RM maps. The SubGoal algorithm is only outperformed in BG maps and is the clearly superior algorithm for the majority of these tests.
**Figure 4.3**: Static Map DAO (Fast). Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of ±95% are shown by dotted lines on either side of the central line.
Figure 4.4: Static Map DAO (Slow). Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of ±95% are shown by dotted lines on either side of the central line.
4.3. Method

**FIGURE 4.5:** Static Map BG (Fast). Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of ±95% are shown by dotted lines on either side of the central line.
Chapter 4. Path Finding

Figure 4.6: Static Map BG (Slow). Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line.
4.3. Method

**Figure 4.7:** Static Map AD (Fast). Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line.
Figure 4.8: Static Map AD (Slow). Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +−95% are shown by dotted lines on either side of the central line.
4.3. Method

FIGURE 4.9: Static Map Rooms (Fast). Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of $\pm 95\%$ are shown by dotted lines on either side of the central line.
Figure 4.10: Static Map Rooms (Slow). Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line.
4.3. Method

4.3.3 Dynamic Map Experimental Results

These results focus on the performance differences between the JPS and BLJPS variants on dynamic environments. Two sets of results are explored. One set explores high frequency map changes (Section 4.3.4 30 Obstacles 1 Path). While the other explores low frequency map changes (Section 4.3.5 30 Obstacles 100 Paths).

4.3.4 Single Path Experiments (30 Obstacles 1 Path)

Figures 4.11 through 4.14 show results for dynamic environments in which 30 obstacles of between 1 and 3 grid squares in linear dimension are added randomly to the maps before each path search is made. This forces a new search to be made for every path.
The results show that JPS+ no longer displays the clear superiority it demonstrated in static environments. Instead, BLJPS shows a clear advantage for longer paths on the open maps (BG, DAO, AD), and basic JPS performs relatively well in all circumstances. Also, the partial update (PU) approach performs poorly compared with the full update (FU) approach in the open maps but out performs other methods in the constricted Rooms maps. Table 4.3 summarises the average speedup obtained by the different algorithms on the different maps relative to A*.
4.3. Method

**Dynamic Map BG - Paths: 1**

![Graph showing speedup against distance for different algorithms.]

**Figure 4.12:** Dynamic Map BG, 1 Path. Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line.

<table>
<thead>
<tr>
<th></th>
<th>DAO</th>
<th>BG</th>
<th>AD</th>
<th>Rooms</th>
<th>Average Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPS</td>
<td>3 (13.2,11.0)</td>
<td>3 (19.2,21.9)</td>
<td>3 (5.3,7.3)</td>
<td>3 (37.2,24.0)</td>
<td>3.0</td>
</tr>
<tr>
<td>BLJPS</td>
<td>1 (28.9,33.1)</td>
<td>1 (90.9,172.6)</td>
<td>1 (9.2,11.9)</td>
<td>4 (27.9,20.6)</td>
<td>1.75</td>
</tr>
<tr>
<td>JPS+ PU</td>
<td>6 (3.2,3.5)</td>
<td>6 (1.6,2.2)</td>
<td>6 (0.3,0.5)</td>
<td>1 (76.0,64.7)</td>
<td>4.75</td>
</tr>
<tr>
<td>BLJPS+PU</td>
<td>5 (4.3,5.3)</td>
<td>5 (2.0,2.9)</td>
<td>5 (0.4,0.6)</td>
<td>2 (61.7,55.3)</td>
<td>4.25</td>
</tr>
<tr>
<td>JPS+ FU</td>
<td>4 (7.7,6.5)</td>
<td>4 (11.9,14.6)</td>
<td>4 (4.1,5.1)</td>
<td>5 (26.8,20.1)</td>
<td>4.25</td>
</tr>
<tr>
<td>BLJPS+FU</td>
<td>2 (17.0,20.5)</td>
<td>2 (43.3,75.7)</td>
<td>2 (7.5,9.2)</td>
<td>6 (18.2,13.8)</td>
<td>3.0</td>
</tr>
</tbody>
</table>

**Table 4.3:** Summarised results for Figures 4.11-4.14. Speedup rank best to worst 1-6 for map and speedup relative to A*. Each result is given in the form X (Y,Z) where X is the rank, Y is the average and Z is the standard deviation of the speedup.
Figure 4.13: Dynamic Map AD, 1 Path. Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line.
4.3. Method

Figure 4.14: Dynamic Map Rooms, 1 Path. Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of ±95% are shown by dotted lines on either side of the central line.
4.3.5 Multiple Path Experiments (30 Obstacles 100 Paths)

Figures 4.15 through 4.18 show results for dynamic environments in which 30 obstacles of between 1 and 3 grid squares in linear dimension are added randomly to the maps before a set of 100 path searches is made. Relative to the previous experiment this configuration queries more paths before the map is modified. The configuration allows the JPS+ algorithm to incrementally rebuild the jump point cache and illustrates the trade-off between the cost of rebuilding the jump-point cache and the efficiency of utilising pre-processed data. Thus, the results show how the cost of rebuilding the jump-point cache is ‘amortised’ over the number of paths that need to be built on top of the invalidated areas. Table 4.4 summarises the average speedup obtained by the different algorithms on the different maps relative to A*.
Figure 4.15: Dynamic Map DAO, 100 Paths. Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of ±95% are shown by dotted lines on either side of the central line.
Figure 4.16: Dynamic Map BG, 100 Paths. Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line.
4.3. Method

FIGURE 4.17: Dynamic Map AD, 100 Paths. Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line.
Table 4.4: Summarised results for figures 4.15-4.18. Speedup rank best to worst 1-6 for map and speedup relative to A*. Each result is given in the form X (Y,Z) where X is the rank, Y is the average and Z is the standard deviation of the speedup.

<table>
<thead>
<tr>
<th></th>
<th>DAO</th>
<th>BG</th>
<th>AD</th>
<th>Rooms</th>
<th>Average Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPS</td>
<td>6 (12,6)</td>
<td>6 (21,7)</td>
<td>6 (5,3)</td>
<td>5 (36,20)</td>
<td>5.75</td>
</tr>
<tr>
<td>BLJPS</td>
<td>5 (63,70)</td>
<td>2 (402,429)</td>
<td>2 (22,14)</td>
<td>6 (31,17)</td>
<td>3.75</td>
</tr>
<tr>
<td>JPS+ PU</td>
<td>4 (64,51)</td>
<td>5 (39,26)</td>
<td>5 (8,8)</td>
<td>1 (118,66)</td>
<td>3.75</td>
</tr>
<tr>
<td>BLJPS+FU</td>
<td>2 (99,86)</td>
<td>3 (125,86)</td>
<td>3 (17,15)</td>
<td>2 (116,67)</td>
<td>2.5</td>
</tr>
<tr>
<td>JPS+ FU</td>
<td>3 (70,52)</td>
<td>4 (46,30)</td>
<td>4 (11,9)</td>
<td>3 (81,60)</td>
<td>3.5</td>
</tr>
<tr>
<td>BLJPS+FU</td>
<td>1 (148,150)</td>
<td>1 (904,1547)</td>
<td>1 (36,25)</td>
<td>4 (71,56)</td>
<td>1.75</td>
</tr>
</tbody>
</table>

Table 4.4 summarizes the average speedup obtained by the different algorithms on the different maps relative to A*.

Figure 4.18: Dynamic Map Rooms, 100 Paths. Each algorithm is represented as three lines of the same colour. Average values are indicated by the patterned middle line. Confidence intervals of +95% are shown by dotted lines on either side of the central line.

Table 4.4 summarizes the average speedup obtained by the different algorithms on the different maps relative to A*.
The results indicate that once again algorithms based on BLJPS outperform those based on standard JPS. However, the effects of caching are marked, leading to BLJPS+FU outperforming standard BLJPS.

### 4.3.6 Memory Usage

A summary of the average memory used by the algorithms is shown below in Table 4.5. The memory allocations show that recording the boundaries of given maps is inexpensive relative to the cost of storing the jump point look up table that JPS+ requires. The Rooms map which represents a worst-case scenario of many boundaries and very short path segments results in an overhead memory increase of 4%. However, for this memory cost, BLJPS+ is in most cases significantly faster at pre-processing the maps than JPS+ (Table 4.10: JPS+ vs. BLJPS+). The adaptive depths map represents a best-case scenario for BLJPS, featuring few boundaries and long path segments. BLJPS shows an increase in the memory allocation of less than 13KB, giving a 38% memory overhead increase in return for approximately three times the execution speed (Table 4.4: AD, BLJPS+FU vs. JPS+FU). The relatively small memory footprint required to store boundaries is justified by the frequent search and pre-processing speedups boundary lookup yields.

BLJPS variants are less efficient in highly complex environments such as the Room maps. Table 4.8 shows the relative number of boundaries stored for each map set. Rooms demands significantly more boundaries to be stored relative to the other maps. Table 4.9 measures the density of the changes within a map, relating how often the boundaries change within a map relative to its size. It can be seen that although BG maps stored on average the second highest number of boundaries per map (Table 4.8) they were the lowest changing environments relative to their
Chapter 4. Path Finding

4.3.7 Pre-Processing Time

Table 4.10 lists the pre-processing time in seconds for the tested algorithms. BLJPS+ is significantly faster than JPS+ with the exception of the Rooms map (P<0.0001). BLJPS+ is up to 32 times faster in pre-processing BG maps, but 1.5 times slower in pre-processing Rooms relative to JPS+. 

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>DAO</th>
<th>BG</th>
<th>AD</th>
<th>Rooms</th>
</tr>
</thead>
<tbody>
<tr>
<td>NovelAStar</td>
<td>29239 (31492)</td>
<td>24826 (123)</td>
<td>1000 (27)</td>
<td>24912 (78)</td>
</tr>
<tr>
<td>PPQ</td>
<td>27 (36)</td>
<td>19 (16)</td>
<td>0 (0)</td>
<td>17 (16)</td>
</tr>
<tr>
<td>SubGoal</td>
<td>1729 (1578)</td>
<td>1341 (167)</td>
<td>160 (9)</td>
<td>2603 (82)</td>
</tr>
<tr>
<td>JPS</td>
<td>11 (4)</td>
<td>11 (2)</td>
<td>10 (2)</td>
<td>12 (3)</td>
</tr>
<tr>
<td>BLJPS</td>
<td>49 (31)</td>
<td>53 (21)</td>
<td>16 (3)</td>
<td>359 (20)</td>
</tr>
<tr>
<td>JPS+</td>
<td>9744 (10511)</td>
<td>8220 (33)</td>
<td>328 (1)</td>
<td>8229 (15)</td>
</tr>
<tr>
<td>BL-JPS+</td>
<td>9784 (10528)</td>
<td>8275 (29)</td>
<td>339 (3)</td>
<td>8579 (29)</td>
</tr>
</tbody>
</table>

Table 4.5: Pre-processing memory overheads for algorithms tested in static maps. Memory overheads are listed in kilobytes. Each result is given in the form X (Y) where X is the average and Y is the standard deviation.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>DAO</th>
<th>BG</th>
<th>AD</th>
<th>Rooms</th>
</tr>
</thead>
<tbody>
<tr>
<td>NovelAStar</td>
<td>34623 (37131)</td>
<td>48614 (83704)</td>
<td>1275 (66)</td>
<td>29227 (153)</td>
</tr>
<tr>
<td>PPQ</td>
<td>1339 (1372)</td>
<td>1124 (50)</td>
<td>85 (37)</td>
<td>1150 (77)</td>
</tr>
<tr>
<td>SubGoal</td>
<td>1854 (1614)</td>
<td>1422 (221)</td>
<td>190 (13)</td>
<td>2890 (30)</td>
</tr>
<tr>
<td>JPS</td>
<td>72 (60)</td>
<td>58 (18)</td>
<td>34 (3)</td>
<td>171 (111)</td>
</tr>
<tr>
<td>BLJPS</td>
<td>118 (82)</td>
<td>102 (30)</td>
<td>47 (5)</td>
<td>531 (119)</td>
</tr>
<tr>
<td>JPS+</td>
<td>9789 (10519)</td>
<td>8260 (19)</td>
<td>358 (2)</td>
<td>8375 (112)</td>
</tr>
<tr>
<td>BL-JPS+</td>
<td>9832 (10534)</td>
<td>8306 (31)</td>
<td>369 (4)</td>
<td>8733 (119)</td>
</tr>
</tbody>
</table>

Table 4.6: Query memory overheads for algorithms tested in static maps (Inclusive of pre-processing). Each result is given in the form X (Y) where X is the average and Y is the standard deviation. Overheads are listed in average memory use of kilobytes.

size (Table 4.9). This could be an indication as to the environments that BLJPS is best suited to, as it performs significantly better on BG maps relative to the other algorithms, outperforming even the SubGoal algorithm.
4.3. Method

<table>
<thead>
<tr>
<th></th>
<th>DAO</th>
<th>BG</th>
<th>AD</th>
<th>Rooms</th>
</tr>
</thead>
<tbody>
<tr>
<td>NovelAStar</td>
<td>96.6 (116.3)</td>
<td>96.6 (189.2)</td>
<td>100.3 (127.9)</td>
<td>96.9 (113.7)</td>
</tr>
<tr>
<td>PPQ</td>
<td>0.1 (5.4)</td>
<td>0.1 (4.4)</td>
<td>0.0 (8.5)</td>
<td>0.1 (4.5)</td>
</tr>
<tr>
<td>SubGoal</td>
<td>8.1 (9.5)</td>
<td>5.2 (5.5)</td>
<td>16.0 (19.1)</td>
<td>10.1 (11.2)</td>
</tr>
<tr>
<td>JPS</td>
<td>0.3 (1.2)</td>
<td>0.0 (0.2)</td>
<td>1.0 (3.4)</td>
<td>0.0 (0.7)</td>
</tr>
<tr>
<td>BLJPS+</td>
<td>0.6 (1.7)</td>
<td>0.2 (0.4)</td>
<td>1.6 (4.7)</td>
<td>1.4 (2.1)</td>
</tr>
<tr>
<td>JPS+</td>
<td>32.2 (33.2)</td>
<td>32.0 (32.1)</td>
<td>33.0 (35.9)</td>
<td>32.0 (32.6)</td>
</tr>
<tr>
<td>BLJPS+</td>
<td>32.7 (33.7)</td>
<td>32.2 (32.3)</td>
<td>34.0 (37.0)</td>
<td>33.4 (34.0)</td>
</tr>
</tbody>
</table>

**TABLE 4.7:** Average memory overhead per map cell in bytes as tested in static maps. Each result is given in the form X (Y) where X is the average and Y is the standard deviation. Bytes = Memory Overhead/(Map_Height * Map_Width).

<table>
<thead>
<tr>
<th></th>
<th>DAO</th>
<th>BG</th>
<th>AD</th>
<th>Rooms</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLJPS+</td>
<td>4187 (4311)</td>
<td>6653 (3109)</td>
<td>1892 (118)</td>
<td>114746 (90)</td>
</tr>
</tbody>
</table>

**TABLE 4.8:** Number of elements within the boundary lookup table after pre-processing. Each element is defined as an unsigned short (2 bytes in size). Each result is given in the form X (Y) where X is the average and Y is the standard deviation.

<table>
<thead>
<tr>
<th></th>
<th>DAO</th>
<th>BG</th>
<th>AD</th>
<th>Rooms</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLJPS+</td>
<td>0.09</td>
<td>0.03</td>
<td>0.19</td>
<td>0.44</td>
</tr>
</tbody>
</table>

**TABLE 4.9:** Average number of elements within the boundary lookup table against map size after pre-processing.
Chapter 4. Path Finding

<table>
<thead>
<tr>
<th></th>
<th>DAO</th>
<th>BG</th>
<th>AD</th>
<th>Rooms</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPS</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>PPQ</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>NovelAstar</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>BLJPS</td>
<td>0.0011s (0.00016)</td>
<td>0.002s (0.0005)</td>
<td>0.0002s (0.0001)</td>
<td>0.0032s (0.0008)</td>
</tr>
<tr>
<td>JPS+</td>
<td>0.8168s (1.751)</td>
<td>26.3316s (57.3343)</td>
<td>0.1362s (0.0233)</td>
<td>0.3597s (0.0195)</td>
</tr>
<tr>
<td>BL-JPS+</td>
<td>0.0613s (0.087)</td>
<td>0.6712s (0.5886)</td>
<td>0.0153s (0.0029)</td>
<td>0.6752s (0.0293)</td>
</tr>
<tr>
<td>SubGoal</td>
<td>0.1084s (0.2375)</td>
<td>0.1265s (0.1363)</td>
<td>0.0105s (0.0016)</td>
<td>0.039s (0.0052)</td>
</tr>
</tbody>
</table>

**Table 4.10:** Time to pre-process a map in seconds. Each result is given in the form X (Y) where X is the average and Y is the standard deviation.

Overall the table indicates that SubGoal is extremely fast at pre-processing. It requires only a portion of the time that BLJPS+ and JPS+ take to pre-process the map while giving relatively good search times as shown in Figs 4.3-4.10.

### 4.4 Discussion

Results in the static maps are exactly as expected: JPS+ is massively faster than either of the basic online search techniques (JPS and BLJPS) due to its direct lookup capability, while BLJPS is faster than standard JPS due to the elimination of iterative searches (P<0.0001). The advantage of BLJPS over standard JPS is more pronounced on relatively open and larger maps where the amount of iteration eliminated is greater. While the search performance for BLJPS+ and JPS+ is the same on static maps, BLJPS+ offers significantly faster pre-processing times (AD, BG and DAO P<0.0001) for a relatively small memory increase.

The results of the dynamic experiments are more complex. When only a single path is searched, all advantages of caching are lost, and rebuilding the lookup
4.4. Discussion

tables becomes a pure cost. The speed of the basic search methods dominates, resulting in the basic BLJPS algorithm dominating on all maps except the constricted Rooms map, where the even more basic JPS algorithm is the most effective. This is due to the relatively small number of points JPS iterates over relative to the cost of finding the associated neighbouring boundary for a given cell in BLJPS. This is reflected in Table 4.9 where the size of the boundary lookup tables relative to the map size is significantly higher than the other maps.

Furthermore, the differing costs of the cache clearing mechanisms are highly evident. A search based cache clearing approach is clearly slower than a simple full clearing, except once again on highly constricted maps. Observation of paths rebuilding invalidated cached data showed that only very few points needed recalculation. This is because only the points along the queried path made up of jump points and their intermediate steps require recalculation. As such the majority of the invalidated data doesn’t require recalculation.

When multiple paths are searched, the advantages of caching are gradually re-established. The hybrid of JPS+ utilising a BLJPS online search mechanism combined with a full cache clearing approach represented by BLJPS+FU shows the best overall performance, and this advantage becomes more pronounced for longer paths. Once again, the costs of search based cache clearing outweigh the benefits of a smaller area to rebuild on all but the most constricted maps. This is because when multiple paths are searched, common jump points are quickly re-cached by earlier searches resulting in faster later searches.
4.5 Conclusion

As long suspected, pathfinding algorithms based on pre-processed data lose their advantage in dynamic environments, where the raw speed of the basic search algorithm dominates. However, the relationship is not binary; when many paths need to be found at each time step, caches can be quickly rebuilt returning the advantage to algorithms utilising pre-processed data. Furthermore, the approach taken to clearing the cache has a large impact. The costs of searching which areas of a grid are impacted by dynamic objects can outweigh the costs of rebuilding a cache for an entire map. The relative costs also depend on the nature of the map. In the end, a hybrid approach which clears cached data quickly, exploits pre-processed data where possible, and maximises the efficiency of the online search had the best performance. BLJPS offers a useful speedup to the basic JPS algorithm and can be directly utilised in JPS+. This combination provides a very fast online pathfinding mechanism over a range of map types and numbers of paths that need to be found. Further work will need to be done on improving the invalidation and recreation of jump point look up tables to improve performance in dynamic maps.

The SubGoal algorithm had significantly lower search (three of four map types) and pre-processing overheads relative to BLJPS+ (P<0.0001). It could be possible that BLJPS+ be modified to include the advantages of the SubGoal algorithm. Alternatively, further research into using SubGoal in dynamic maps could yield interesting comparisons with this work.

The BLJPS algorithm was used to speed up pathfinding for Search and Recall, discussed in Chapter 6. An extension of this work (BLJPS2, Chapter 5) was used to support high fidelity simulations of complex tactical scenarios for the work on Dynamic Granularity presented in Chapter 7.
Chapter 5

BLJPS Improvements

5.1 Overview

In a further effort to improve pathfinding speed to support high fidelity simulations, a number of further optimisations to the BLJPS algorithm were explored. These optimisations are presented in this chapter. The optimisations were designed to increase the pathfinding search speeds in potentially dynamic domains. The work in this Chapter, specifically BLJPS2, is used as the path finding algorithm for the work on Dynamic Granularity in Chapter 7.

Many pathfinding algorithms currently exist that are capable of searching grid space much more rapidly than the basic JPS or BLJPS algorithms. However, this speedup of search time generally comes at the cost of pre-processing time, memory and storage space. Techniques such as Compressed Path Databases (CPD) (Botea, 2011; Botea, 2012) takes hours to pre-process maps and gigabytes of storage to hold data, but can perform extremely fast queries. However, because these algorithms require a large amount of memory and storage they are not well suited for mobile devices and certain applications. The memory and storage requirements increase substantially when maps contain a high level of detail, multiple maps are required, or both.
Other algorithms such as BLJPS (Traish, Tulip, and Moore, 2016) and SubGoal (Uras, Koenig, and Hernández, 2013) do very well within a smaller memory, storage and pre-processing footprint. SubGoal has been shown to be very efficient at pruning a map’s nodes to improve search speed.

With these techniques in mind, an arc pruning algorithm is presented which is designed to exploit the direction expanding nature of BLJPS. It is shown that such an approach can achieve much higher search speeds than BLJPS with a small trade-off in memory and pre-process time.

Sections 5.1.1 to 5.1.4 give a brief description of several variants of the BLJPS algorithms, which incrementally improve search speeds. Each version is built on the previous version, so a steady increase in search speed is found with an increasing cost of memory and pre-processing time. Note that experiments were conducted only on non-dynamic maps in order to facilitate comparison to results in other published work (Uras, Koenig, and Hernández, 2013).

The work presented in this chapter has been published in the following conference papers:


### 5.1.1 BLJPS2

BLJPS2 improves search speeds by storing the jump point locations along the cardinal directions. This modification reduces the number of vertical and horizontal
axis lookups to 1 each per diagonal step compared to standard BLJPS.

Algorithm 3 BLJPS2

1: Goal_Y_Boundary_Space[Min,Max]  ▷ This is calculated once at the start of
   the search against the boundary list
2: procedure BLJPS2_CARDINAL_JUMP(cp :current position, cd : cardinal di-
   rection, g : goal)
3:  np ← step(cp, cd, 1)
4:  if not is_passable(np) then return Null
5:  if cd == NORTH then
6:    if (np.x == g.x) and (np.y <= g.y) and (np.y >=
      Goal_Y_Boundary_Space.Min) and (np.y <= Goal_Y_Boundary_Space.Max)
    then return g
7:    next_Y ← binary_Search(nothernJumpPoints[np.x], np.y)
8:    if next_Y == Null then
9:      return Null
10:   else
11:     return jumpPoint(np.x, next_Y)
     ▷ Repeat check against South, East and West

5.1.2 BLJPS3

BLJPS3 stores the locations of jump points along the diagonal directions. Thus it
requires only a single lookup for a diagonal expansion, once again moving away
from iterative stepping.
Algorithm 4 BLJPS3

1: procedure BLJPS3_JUMP(cp: current position, d: direction, g: goal)
2:     np ← step(cp, d, 1)
3:     if not is_passable(np) then return Null
4:     if np == g then return g
5:     if has_forced_neighbour(np, d) then return n
6:     if d is diagonal then
7:         return BLJPS3_Diag_Jump(np, d, g)
8:     else
9:         return BLJPS2_Cardinal_Jump(np, d, g)

10: procedure BLJPS3_Diag_Jump(cp: current position, d: direction, g: goal)
11:     if d is North-East or South-West then
12:         cp_Diagonal ← cp.x + cp.y
13:         goal_Diagonal ← g.x + g.y
14:     else
15:         cp_Diagonal ← cp.x - cp.y + (GRID_HEIGHT - 1)
16:         goal_Diagonal ← g.x - g.y + (GRID_HEIGHT - 1)
17:     diagonalJumpPointList ← getDiagJumpPointList(Pre-ProcessedDiagJP[d][cp_Diagonal], cp)
18:     if diagonalJumpPointList == Null then return Null
19:     if cp_Diagonal == goal_Diagonal then △ The cell and goal lie on the same diagonal line
20:         if diagonalJumpPointList.SpaceStartX <= g.x and diagonalJumpPointList.SpaceEndX >= g.x then
21:             return g
22:     else
23:         diagMovements ← min(abs(cp.x - g.x), abs(cp.y - g.y))
24:         if openSpace(cp+diagMovements*d, g) then
25:             return jumpPoint(cp+diagMovements*d)
26:     for all diagJumpPoint in diagonalJumpPointList do
27:         if d is North-East or South-East then
28:             if diagJumpPoint.x >= cp.x then return diagJumpPoint
29:         else
30:             if diagJumpPoint.x <= cp.x then return diagJumpPoint
31: return Null
5.1.3 BLJPS4

BLJPS4 creates a node corresponding to each forced neighbour on the map. Each of these nodes stores a list of connected nodes in each possible direction of travel. These are referred to as arcs. A search starts by adding all the relevant forced neighbours reachable from the start point to an open list. As each node is popped from the list, it performs a quick check to determine if that node can move directly to the destination position. If the destination node can be reached, then the solution has been found. Otherwise, the node’s arcs are expanded in the direction of travel and the process iterates until the open list is empty or the solution is found. The closed check list size is reduced significantly in size with this approach. Previous to BLJPS4 the closed list had to account for any possible grid position, but in BLJPS4 it only has to account for forced neighbour positions. On one map this reduces the number of possible states from 10,000 (100 by 100 units along the x and y-axis) to 500 states.
Algorithm 5 BLJPS4

1: procedure BLJPS4_PRE-PROCESSING
2:     pre-processBoundaryLists_BLJPS()
3:     pre-processCardinalJumpPoints_BLJPS2() ▷ Adds all cardinal jump points to allGraphNodes
4:     pre-processDiagonalJumpPoints_BLJPS3() ▷ Adds all diagonal jump points to allGraphNodes
5:     pre-processJumpPointNodeLinks_BLJPS4() ▷ Links all jump points into a graph
6:     pre-ProcessLimits() ▷ For each jump point find how much open space exists in each direction
7:     closedGrid_Size ← allGraphNodes.size()
8: procedure PRE-PROCESS_JUMPPOINTNODELINKS_BLJPS4
9:     for all JumpPoint in allGraphNodes do
10:        for all d in allDirections do
11:           recursive_BLJPS_Jump(JumpPoint.position, d, JumpPoint.linkedNodes[d])
12: procedure RECURSIVE_BLJPS_JUMP(cp: current position, d: direction, jpList: Jump Point List)
13:     np ← step(cp, d, 1)
14:     if not is_passable(np) then return
15:     if has_forced_neighbour(np,d) then
16:        jpList.append(getJumpPointAtLocation(np))
17:     return
18:     if d is diagonal then
19:        for all cardinal direction c in d do
20:           cardinalJumpPoint = BLJPS2_cardinal_Jump(np, c, Null)
21:           if cardinalJumpPoint ≠ Null then
22:              jpList.append(cardinalJumpPoint)
23:        else
24:           cardinalJumpPoint = BLJPS2_cardinal_Jump(np, c, Null)
25:           if cardinalJumpPoint ≠ Null then
26:              jpList.append(cardinalJumpPoint)
27:     return
28: recursive_BLJPS_Jump(np, d, jpList)
5.1. Overview

5.1.4 BLJPS5

This algorithm creates a backwards lookup table for incoming arcs from jump points. It also prunes arcs between nodes in a process that will be described below. When a search is initiated all the start nodes are added, as in BLJPS4, but in addition, all the identified end nodes are flagged (i.e., nodes that can directly reach the end goal). Now, as the search takes place a trivial check against a node’s flagged state will determine if an end node has been reached. Thus, this approach only incurs a one-off cost in determining the end nodes at the start of the search, compared to BLJPS4 which incurs this cost at every node expansion. This causes BLJPS4 to run slower on longer more complex paths but faster on simpler paths.

BLJPS5 prunes arcs between nodes, which works in contrast to SubGoal’s approach of pruning entire nodes from the graph. BLJPS5 uses two copies of the arcs for this pruning method. One acts as a read-only list of all the original arcs while the second is modified during the following process. For each arc in the read-only list, a series of possible following arcs are identified. Where an arc comes to a node, it is expanded by its natural and forced neighbour directions. If the originating node of the tested arc can connect to all the identified outgoing arcs, the arc is pruned, and all the identified outgoing arcs take its place on the originating node. In essence, this skips the tested node and moves directly onto the next nodes during a search.

All nodes that have an arc pruned have the arc added to an incoming pruned arc list. This is used when flagging end nodes at the start of a search. Due to the pruning process, an end node may have been pruned from the graph in a given direction. Therefore, all directly connected end nodes are expanded in forced neighbour directions adding nodes from the incoming pruned arc list. This allows the search to find an end-node whether it was pruned or not. This pruning
process allows many unnecessary nodes to be skipped during search and simplifies determining if an end node has been reached to a flag check.

Algorithm 6 BLJPS5

1: procedure BLJPS5_PRE-PROCESSING
2: pre-processBoundaryLists_BLJPS()
3: pre-processCardinalJumpPoints_BLJPS2() ▷ Adds all cardinal jump points to allGraphNodes
4: pre-processDiagonalJumpPoints_BLJPS3() ▷ Adds all diagonal jump points to allGraphNodes
5: pre-processJumpPointNodeLinks_BLJPS4() ▷ Links all jump points into a graph
6: pruneDirectionLinks_BLJPS5() ▷ Builds a high level graph
7: pre-ProcessBackwardsList_BLJPS5() ▷ Links all grid positions directly onto the graph
8: closedGrid_Size ← allGraphNodes.size()
9: procedure PRE-PROCESSBACKWARDSLIST_BLJPS5
10: for all JumpPoint in allGraphNodes do
11: for all diagonalDirection in forcedDirections(JumpPoint) do
12: \( np \leftarrow \text{step}(\text{JumpPoint}.\text{position}, \text{diagonalDirection}, 1) \)
13: while is_passable(np) do
14: for all cardinal direction c in diagonalDirection do
15: if isDirectionHorizontal(c) then
16: axisValue ← np.y
17: else
18: axisValue ← np.x
19: backwardLookupTable[c][axisValue].append(np)
20: \( np \leftarrow \text{step}(\text{JumpPoint}.\text{position}, \text{diagonalDirection}, 1) \)

5.2 Experimental Approach

The experiments evaluate the improvements of the BLJPS algorithms and secondarily compares the performance of BLJPS5 arc pruning algorithm against SubGoal. SubGoal (Uras, Koenig, and Hernández, 2013) was chosen because of its quick search times and relatively small memory and pre-processing implements a node pruning approach. SubGoal’s source was taken from (Sturtevant, 2013) (2013 entry: SubGoal Optimal). The JPS experiments of (Harabor and Grastien,
5.2. Experimental Approach

<table>
<thead>
<tr>
<th>Map</th>
<th>Number of Maps</th>
<th>Map Size</th>
<th>Average number of paths per map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptive Depth (AD)</td>
<td>12</td>
<td>100x100</td>
<td>1000</td>
</tr>
<tr>
<td>Baldur’s Gate (BG)</td>
<td>75</td>
<td>512x512</td>
<td>1242</td>
</tr>
<tr>
<td>Dragon Age Origins (DAO)</td>
<td>156</td>
<td>Smallest: 28x22 Largest: 104x1260</td>
<td>1022</td>
</tr>
<tr>
<td>Rooms (Rooms)</td>
<td>10</td>
<td>512x512</td>
<td>1928</td>
</tr>
</tbody>
</table>

Table 5.1: Maps used in experiments. DAO maps vary in size, ranging from the smallest and largest map sizes given. Each map is associated with a varying number of paths, the average for map each set is given.

2011) were replicated, using the freely distributed datasets from the (Sturtevant, 2013) (GPPC). The source code for these experiments is publicly available on GitHub (Traish, 2017b). The map used in each experiment includes a set of predefined problem paths that are used to benchmark the algorithms.

The experiments were conducted using diagonal unblocked mode allowing paths to 'cut' corners. This mode was selected so that SubGoal and BLJPS variants would have comparable results. This mode differs from the earlier implementation of JPS where both cardinal directions associated with a diagonal step must be obstacle free to be a legal step. This unblocked mode is used in the GPPC (Sturtevant, 2013) but differs from the original implementation of JPS (Harabor and Grastien, 2011).

Table 5.1 lists the maps used in the experiments. The maps are consistent in size, except for the Dragon Age Origins (DAO) maps which varied, ranging between the specified smallest and largest sizes. Each map was associated with a varying number of paths, each specified by a start and end point. The average number of paths for each map set is listed Table 5.1.

1. Adaptive Depth (AD): Maps are relatively small and have few obstacles.
2. Baldur’s Gate (BG): Maps are large size and generally have very simple layouts with large open areas.

3. Dragon Age Origins (DAO): Maps are diverse in size and obstacle density.

4. Rooms maps are large (Rooms): Maps are large and contain an extremely high density of obstacles.

The performance of BLJPS, BLJPS2, BLJPS3, BLJPS4, BLJPS5 and SubGoal were evaluated. On each map, all paths specified in the data set were executed giving paths of varying lengths. Optimal paths between start and end points were then generated repeatedly until each had been calculated at least 100 times. This approach allowed for accurate evaluation of the time taken to find a path. The time taken to find a path was then calculated as the average search time over the number of iterations.

The time spent pre-processing the maps was measured, the memory usage after completing pre-processing and the maximum memory used after completing all searches was recorded. These results are summarized in Tables 5.2 and 5.3.

5.3 Results and Discussion

A brief summary of the Tables 5.2 and 5.3 are as follows:

1. BLJPS5 has a faster search time than SubGoal in 3 out of 4 cases (AD: P<0.042, BG: P<0.0001 and DAO: P<0.0001).

2. BLJPS5 is faster than BLJPS (For all map types P<0.0001).

3. BLJPS5 occasionally pre-processes maps faster than SubGoal (BG: P<0.0001 and DAO: P<0.0013).

4. BLJPS5 is more memory intensive than SubGoal (All map types: P<0.0001).
5.3. Results and Discussion

<table>
<thead>
<tr>
<th>Map</th>
<th>Algorithm</th>
<th>Total Search Time</th>
<th>Average Pre-Process Time</th>
<th>Average Starting Memory</th>
<th>Average Max Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD</td>
<td>BLJPS5</td>
<td>59 (8)</td>
<td>12s (2)</td>
<td>570 (32)</td>
<td>604 (30)</td>
</tr>
<tr>
<td></td>
<td>SubGoal</td>
<td>61 (8)</td>
<td>11s (2)</td>
<td>160 (10)</td>
<td>190 (13)</td>
</tr>
<tr>
<td>BG</td>
<td>BLJPS5</td>
<td>352 (66)</td>
<td>70s (27)</td>
<td>2619 (889)</td>
<td>2669 (932)</td>
</tr>
<tr>
<td></td>
<td>SubGoal</td>
<td>1192 (107)</td>
<td>127s (136)</td>
<td>1341 (167)</td>
<td>1422 (221)</td>
</tr>
<tr>
<td>DAO</td>
<td>BLJPS5</td>
<td>4141 (247)</td>
<td>35s (47)</td>
<td>3451 (2747)</td>
<td>3605 (2859)</td>
</tr>
<tr>
<td></td>
<td>SubGoal</td>
<td>6278 (234)</td>
<td>108s (237)</td>
<td>1729 (1578)</td>
<td>1855 (1614)</td>
</tr>
<tr>
<td>Rooms</td>
<td>BLJPS5</td>
<td>6334 (218)</td>
<td>305s (15)</td>
<td>16302 (198)</td>
<td>18047 (204)</td>
</tr>
<tr>
<td></td>
<td>SubGoal</td>
<td>3839 (64)</td>
<td>39s (5)</td>
<td>2604 (82)</td>
<td>2890 (30)</td>
</tr>
</tbody>
</table>

Table 5.2: Experimental Results: BLJPS5 vs SubGoal. Bold represents the best result. Each result is given in the form X (Y) where X is the average and Y is the standard deviation.

The results from Table 5.2 show BLJPS5 has the overall fastest search speed in the maps AD, BG and DAO. However, it does not perform as well as SubGoal on the obstacle dense map set of Rooms.

Table 5.3 shows BLJPS5 was substantially superior in search speed relative to BLJPS. It is worth remarking that BLJPS4 outperformed BLJPS5 on the maps AD (P<0.0001) and BG (P<0.0001) due to the relatively simple paths on these maps that didn’t require the computational overhead of BLJPS5 (see section 5.1.4). On more complex maps, DAO and Rooms, the performance improvements of BLJPS5 is clearly seen.

Figure 5.1 shows a detailed graph of the performance of each algorithm relative to A* with respect to path lengths. The large spikes in performance for BLJPS3, 4 and 5 at shorter distances represent paths that pass around only a single corner. A function finds and returns these paths before a search takes place. The check is inexpensive and substantially speeds up searches that only have a single corner between start and end points. This performance gain was found throughout the BG testing maps, which resulted in the relatively quick search times.

BLJPS5 incurs a larger memory footprint relative to SubGoal but remains within...
<table>
<thead>
<tr>
<th>Map</th>
<th>Algorithm</th>
<th>Total Search Time (ms)</th>
<th>Average Pre-Process Time (ms)</th>
<th>Average Starting Memory (Kb)</th>
<th>Average Max Memory (Kb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD</td>
<td>BLJPS</td>
<td>338 (8)</td>
<td>0.2 (0.1)</td>
<td>12 (3)</td>
<td>43 (5)</td>
</tr>
<tr>
<td></td>
<td>BLJPS2</td>
<td>174 (4)</td>
<td>0.6 (0.2)</td>
<td>27 (5)</td>
<td>67 (6)</td>
</tr>
<tr>
<td></td>
<td>BLJPS3</td>
<td>123 (5)</td>
<td>5.2 (1)</td>
<td>183 (11)</td>
<td>217 (12)</td>
</tr>
<tr>
<td></td>
<td>BLJPS4</td>
<td>36 (5)</td>
<td>8.8 (1.78)</td>
<td>299 (14)</td>
<td>336 (14)</td>
</tr>
<tr>
<td></td>
<td>BLJPS5</td>
<td>59 (8)</td>
<td>12.1 (2.4)</td>
<td>570 (32)</td>
<td>604 (30)</td>
</tr>
<tr>
<td>BG</td>
<td>BLJPS</td>
<td>5523 (81)</td>
<td>2.2 (0.7)</td>
<td>49 (20)</td>
<td>98 (30)</td>
</tr>
<tr>
<td></td>
<td>BLJPS2</td>
<td>2387 (52)</td>
<td>4.1 (1.5)</td>
<td>109 (32.1)</td>
<td>157 (45)</td>
</tr>
<tr>
<td></td>
<td>BLJPS3</td>
<td>991 (61)</td>
<td>36.0 (12.8)</td>
<td>544 (222)</td>
<td>594 (229)</td>
</tr>
<tr>
<td></td>
<td>BLJPS4</td>
<td>250 (42)</td>
<td>59.8 (21.9)</td>
<td>1740 (365)</td>
<td>1777 (367)</td>
</tr>
<tr>
<td></td>
<td>BLJPS5</td>
<td>352 (66)</td>
<td>70.2 (27.3)</td>
<td>2619 (889)</td>
<td>2669 (932)</td>
</tr>
<tr>
<td>DAO</td>
<td>BLJPS</td>
<td>54080 (114)</td>
<td>1.2 (1.8)</td>
<td>45 (31)</td>
<td>114 (82)</td>
</tr>
<tr>
<td></td>
<td>BLJPS2</td>
<td>30686 (121)</td>
<td>2.5 (3.4)</td>
<td>112 (63)</td>
<td>185 (110)</td>
</tr>
<tr>
<td></td>
<td>BLJPS3</td>
<td>24050 (207)</td>
<td>14.6 (18.4)</td>
<td>659 (421)</td>
<td>733 (467)</td>
</tr>
<tr>
<td></td>
<td>BLJPS4</td>
<td>5264 (211)</td>
<td>24.3 (30.7)</td>
<td>2276 (1906)</td>
<td>2317 (1910)</td>
</tr>
<tr>
<td></td>
<td>BLJPS5</td>
<td>4141 (247)</td>
<td>35.2 (46.8)</td>
<td>3451 (2747)</td>
<td>3605 (2859)</td>
</tr>
<tr>
<td>Rooms</td>
<td>BLJPS</td>
<td>70853 (77)</td>
<td>3.7 (1)</td>
<td>354 (19)</td>
<td>557 (148)</td>
</tr>
<tr>
<td></td>
<td>BLJPS2</td>
<td>37242 (65)</td>
<td>31.4 (3.4)</td>
<td>608 (17)</td>
<td>813 (145)</td>
</tr>
<tr>
<td></td>
<td>BLJPS3</td>
<td>33979 (289)</td>
<td>142.5 (9.1)</td>
<td>7472 (58)</td>
<td>7689 (160)</td>
</tr>
<tr>
<td></td>
<td>BLJPS4</td>
<td>6340 (282)</td>
<td>225.5 (13)</td>
<td>11084 (71)</td>
<td>11199 (72)</td>
</tr>
<tr>
<td></td>
<td>BLJPS5</td>
<td>6334 (218)</td>
<td>305 (15.5)</td>
<td>16302 (198)</td>
<td>18047 (204)</td>
</tr>
</tbody>
</table>

Table 5.3: Experimental Results: BLJPS variations. Bold represents the best result. Each result is given in the form X (Y) where X is the average and Y is the standard deviation.
5.4 Conclusion and Further Work

BLJPS5 substantially increased search speed and was able to outperform SubGoal in three out of four data sets. However, while BLJPS5 exhibited the overall fastest search times, it was a challenging problem to adapt it maintain its performance in a dynamic environment due to the amount of cached data it uses. Future work...
may look at extending the BLJPS5 algorithm to further prune connections between forced neighbours. Further effort could also be made into reducing the search speed on obstacle dense maps such as the Rooms data set.

Each new version of BLJPS increased search speed at the cost of longer preprocessing times and increased size of cached pre-processed data. BLJPS3-5 required significantly more time than BLJPS2 to reconstruct cached data on dynamic maps because the backwards searching to determine invalidated paths was computationally expensive. BLJPS2 improved search speed relative to BLJPS, but could reconstruct cached data with little overhead relative to BLJPS3-5. BLJPS2 was therefore selected as the pathfinding algorithm to support later work.
Chapter 6

Search and Recall

6.1 Introduction

This chapter explores the notion put forward in Section 2.9 that Case-Based Reasoning (CBR) could be combined with Search to decouple Search from the processing constraints of the game loop, and to provide CBR with a more effective ability to adapt to novel situations. This chapter focuses on addresses the research question "How can adaptive tactical RTS AI be improved to more effectively find solutions to complex, previously unseen scenarios?" (Section 1.4). The method developed is referred to as "Search and Recall".

CBR methods have been used successfully to create adaptive RTS agents. In general, such methods store plans with an associated game state and use this data to reason about future encounters.

However, while CBR has been successful in creating adaptive RTS agents, they face a number of challenges. Responses derived from human players can be of inconsistent quality due to the diversity of human player skills and the nature of human play. Standard CBR approaches are also ill-equipped to make decisions if there is no similar recorded context.
Recently, search based methods, and in particular Monte Carlo simulations have gained the interest of the RTS AI research community (Balla and Fern, 2009; Chung, Buro, and Schaeffer, 2005; Churchill and Buro, 2012; Churchill, Saffidine, and Buro, 2012; Zhe et al., 2012). Search-based methods enable an agent to adapt in real-time to whatever circumstances it is currently facing, assuming the simulator can correctly predict the outcome of a given response action. Significant research on adaptive agents using search-based techniques has been performed in the context Chess and GO (Marcolino and Matsubara, 2011) and the application of such techniques to RTS games is an attractive prospect.

The central issue in applying Search techniques such as MCTS to RTS games is the very large search space, which takes a long time to explore. The complexity inherent in commercial RTS games places huge computational demands on the simulations required to perform a search for a tactical solution. The problem is that simulations conducted within the game loop are heavily constrained to execute in an extremely limited amount of time, due to the real-time aspect of RTS games. For this reason, most of the published research in search based RTS AI use highly simplified scenarios relative to the demands of fully realised commercial game agents and ignore issues such as terrain, pathfinding, and collisions between units. Simplifying the simulators used to evaluate different samples from the search space allows a large number of simulations to be conducted within a short time (see Section 2.7). Including computationally intensive aspects of RTS games such as dealing with unit collisions, pathfinding and terrain greatly increases the processing load associated with evaluating potential solutions and reduces the number of samples that can be evaluated. If Search is conducted within the time constraints of the game loop (typically allowing no more than 40ms per iteration), then including computationally intensive tasks in evaluation of potential solutions renders Search techniques non-feasible as an RTS AI approach.
However, complexities such as pathfinding and collision detection are required for an agent to appropriately handle commercial game type tactical situations. Such situations include moving units in an environment affected by terrain, or engaging armies of many varied unit types, some with special abilities. Furthermore, complex scenarios built around interacting with terrain (such as those involving choke points) cannot be solved without taking into account pathfinding and collision detection (see Chapter 7). Ignoring terrain in these scenarios would be similar to ignoring how real world medieval castles utilised moats and bridges for defensive purposes.

It would be extremely useful if Search could be decoupled from the game loop, or even conducted offline from the game itself. This would allow computationally intensive tasks to be included in ‘high fidelity’ evaluations of potential solutions which would make the results of simulation based evaluations fare more applicable to the complex scenarios actually encountered in a real RTS game.

CBR is an example of a technique that decouples the discovery of potential solutions from the game loop and the game. In CBR, potential solutions are recorded in a database of previously played games as shown in (Ontañón et al., 2007). It seems possible that solutions in a CBR database could be generated through simulation. This would allow Search to be conducted separately from an actual game, and make the results available within the game with almost no computational cost. This would make it possible to include computationally expensive aspects of RTS game in ‘high fidelity’ Searches, while the results could later be ‘Recalled’ in an actual game. It would also allow the CBR to be extended to new games and, depending on the quality of the AI agent provide better more consistent solutions. Solutions could be generated off-line providing a rapid in-game response while allowing a thorough exploration of the game state space.
In the rest of this chapter, a hybrid Search/CBR approach called Search and Recall (S&R) is presented. S&R enables the asynchronous execution of simulations capable of dealing with commercial grade RTS game complexity while offering CBR level in-game performance. These capabilities are demonstrated in the context of small tactical engagements in Brood War.

In this study, a comparison is made with the work in (Churchill and Buro, 2012) which uses the SparCraft (SparCraft - BroodWar simulator) simulator. SparCraft is a simplified simulator which ignores terrain, unit collisions, pathfinding, and terrain. Churchill’s work shows that simulators without these features can still generate successful solutions to some scenarios. The current work implements both a simplified simulator similar to SparCraft (Basic Simulator), and a high-fidelity simulator that supports unit collisions, pathfinding and interaction with terrain (Complex Simulator). It shows that including pathfinding and collision detection increases the performance of an agent (see Section 6.4 and 6.5, Basic vs Complex Simulator models). The BLJPS2 algorithm developed in Chapter 5 is used to implement fast pathfinding in the Complex Simulator.

The main contribution of this chapter is to demonstrate the feasibility and utility of storing responses generated using Search simulations as recorded responses in a CBR database. The work in this chapter demonstrates the feasibility of making computationally intensive search simulations available in the context of a real-time game and was presented in the following conference paper:

6.2 Search and Recall - Overview

Search and Recall (S&R) is a novel method of tactical decision making which is a hybrid of Search and Case-Based Reasoning (CBR) methods. It allows an agent to learn and retain strategies discovered over the agent’s history of play while adapting quickly to novel circumstances.

Similarly to CBR methods, S&R uses a database of previously discovered successful responses associated with a collection of identified game states. S&R agents use these responses to quickly identify a solution without extensive simulation within the game loop. However, unlike other CBR methods, S&R does not populate its response database with responses identified from previously played games. Rather, it populates the database dynamically with the results of search simulations conducted in response to actual game states encountered during play. By combining the adaptive learning of Search with the memory of CBR, S&R allows an agent to improve the quality of its responses both within the game and over the course of multiple games.

In essence, the search tasks are decoupled from the game loop by allowing them to execute asynchronously and in parallel with the game loop. Searches are pushed into concurrent threads, allowing them to take as long as necessary without delaying game rendering. The agent makes its decisions based on its current database of solutions, and the search tasks update that database asynchronously with the results of new simulations based on possible responses to the current game state. As many searches can be carried out as are appropriate to the CPU resources available to the game.

Search time is limited only by the length of a match or an arbitrary stopping condition and is substantially longer than the 5ms generally allocated for an agent’s decision-making process within the standard game loop. The downside is that
the longer it takes to evaluate potential decisions the more likely it is that the response will come too late to be useful in the current situation. However, the next time a similar situation is encountered, the simulation results will be available in the CBR database (response library) ready for near-instant access.

Search results are used to update the CBR database as they become available, and the AI task within the game loop is reduced to selecting the appropriate response as in a conventional CBR system.

The architecture is applied in the context of Starcraft Brood War. Starcraft is an immensely popular and sophisticated RTS game, famous for its balanced asymmetric gameplay and status as a professional spectator sport in Korea. Starcraft Brood War is a version of Starcraft for which an external programming interface has been developed called the Brood War API (BWAPI). The availability of BWAPI has made Brood War an attractive platform for RTS AI research.

### 6.3 Search and Recall - Agent Components

The S&R agent is composed of a response-playback simulator (RPS) component, a concurrent search simulator component (CS), and a response library (RL). This basic architecture is illustrated in Figure 6.1. The RPS component acts as coordinator for the agent and interacts with the BWAPI interface. As soon as a Brood War game begins the S&R agent starts the response-playback component and initialises the concurrent search component with a number of threads.

#### 6.3.1 Response/Playback Simulator (RPS) Component

The RPS component matches the current game state against the game states currently recorded in the Response Library. In this implementation, game states in
6.3. Search and Recall - Agent Components

The database are identified by a simplified descriptor containing only the number and types of units present.

The RPS then retrieves the response associated with the current game state from the response library. The response associated with a game state is always the most favourable response generated by the search simulations carried out in the search component. In the current implementation, if no matching game state is found, the RPS assigns random behaviours to the agent’s units. If a response was loaded earlier from a previous game state, then those previous behaviours are not changed.

The RPS simulator is similar to those being used for searching. It simulates a single time step using the unit actions specified in the response. This step is carried out in order to map from the actions specified in response to a set of Brood War commands that must be issued through the BWAPI interface. The raw actions that the units must perform are recorded (e.g. move[x,y], attack[unitId]) and forwarded to the BWAPI.

Although games states are identified in the RL only by the number and type of units present, the actual game state is defined with considerably more information on unit positions, current unit states, what projectiles have been created, which units are damaged, and which weapons have entered their cool down periods. All of this information is captured from the BWAPI and sent through to the

![Figure 6.1: Search and recall agent process](image)
Concurrent Search component (CS) in addition to the number and type of units present in the scenario. The RPS buffers these changes in actual game state for the CS, updating the information used by that component as a basis for simulation only after 200 simulations have completed. This allows a sufficient number of searches associated with a particular game state to complete to be useful in subsequent games.

The execution of the RPS is constrained to take less than 5ms per frame since it executes as a part of the main game loop. This constraint is easily achieved since the simulator used to calculate the BWAPI commands simulates only a single time step.

6.3.2 Concurrent Search Component (CS)

The search component is represented in Figure 6.1 as the Concurrent Search simulators (CS). It consists of a number of search threads which repeatedly run simulations for the combat scenario utilising the current actual game state, a simulator engine, and a set of actions assigned to each unit in the scenario.

At the beginning of a simulation, each search thread is given the current identifying game state (unit numbers and types) as well as information describing the actual game state (terrain, unit positions, unit health, current unit action states, etc). In the current implementation, no structured search mechanism such as MCTS is implemented. Instead, behaviours are randomly assigned to each unit for each simulation so the system can evaluate the effect of utilising different tactics on the outcome of a battle.

If simulations complete quickly, many different possible outcomes can be calculated and used to update the solution available to the RPS before the time allotted to its execution within the game loop (5ms) expires. However, if it takes longer to
6.3. Search and Recall - Agent Components

Simulate an outcome than the time Brood War allows, then the result will not be available to the RPS during the current game loop. This may result in the game agent taking longer than a single game loop cycle to respond effectively to a game state, as simulation results become available to the RPS over the next few game loops as they complete.

When a simulation completes, the quality of the response is calculated as the total health percentage of the remaining allied units at the end of the simulation. A quality of 0 is given for a prediction in which all allied units are killed. This formula favours victories with lower casualties and ranks all losses equally.

As in (Churchill, Saffidine, and Buro, 2012), the simulator is a mathematical model of the combat mechanics implemented in Brood War, that allows simulations to be run without any frame rate derived speed limitations. It is not an exact model of the combat mechanics implemented in Brood War.

6.3.2.1 Simulators

Asynchronous execution of search allows the complexity and execution cost of the simulation engine used to be increased arbitrarily. In this work, the effect of increasing the complexity of the simulation engine used is investigated by evaluating the performance of two different simulators. These are:

1) Basic Simulator (BAS): This simulator handles unit health, shields, healing, attacking, and movement without collision or pathfinding. It can complete up to 2000 combat simulations per second per thread. The Basic simulator is very similar to the SparCraft simulator used in (Churchill and Buro, 2012)). The Basic simulator models the same health and attacking data as SparCraft and similarly does not implement collision detection and pathfinding. The Basic Simulator doesn’t implement influence maps (Uriarte and Ontanón,
2012), so it cannot support a kiting. Kiting is a highly successful behaviour that fast-moving ranged units can use against slower units. Kiting is the act of attacking an enemy unit and then moving away while reloading. However, the Basic simulator does take into account unit turning which SparCraft does not. Unit turning affects the timing between targeting, attacking and movement commands. The Basic simulator can execute up to 2000 combat simulations per thread per second.

2) Complex Simulator (COM): This simulator handles unit health, shields, healing, and attacking as in the Basic Simulator. However, in the Complex Simulator the movement function detects collisions and finds paths around obstacles such as terrain and other units. The Complex Simulator also implements influence maps to support a ‘kiting’ behaviour which has been added to the list of available behaviours. The Complex Simulator can complete only up to 200 combat simulations per thread per second.

The Basic Simulator was built as part of the Complex Simulator in order to carry out the experiments for this chapter and Chapter 7. Since the Basic Simulator provided similar functionality to the SparCraft simulator, the SparCraft simulator was not used in this thesis. Requests for a copy of the simulators can be made to jtraish@csu.edu.au.

6.3.2.2 Response Divergence

A response grows stale the longer it is in effect. This is due to differences in mechanics between the Brood War game and the simulator that even a very sophisticated model will find challenging to eliminate, in particular, because there are random elements built into the Brood War game engine. The difference between the simulated outcome and what actually happens in Brood War is referred to
as divergence. Divergence represents the cumulative error between the detailed game states of the simulation and Brood War as time passes.

Different game systems suffer differing amounts of divergence. While systems like health regeneration and attack damage are straightforward, other components such as attack cool-downs are randomised slightly, introducing small changes in combat outcomes. The precise mechanics of other systems such as pathfinding are unknown, and this also increases the divergence of simulations from actual game encounters. Furthermore, an opponent model is not necessarily a precise model of the Brood War AI, and this also leads to a large amount of divergence. Finally, the actual precise game state used to drive the search simulation that generated the response recorded in the database may differ from the precise current game state. If differences in precise unit location and health affect the outcome of the battle, divergence will occur.

6.3.2.3 Opponent Models

In order to combat the effects of divergence, solutions that generalise well are sought. The simulation outcome is heavily dependent on the strategy used by the opponent, so an attempt was made to find generalised solutions by taking the minimum of the solution quality score over a small set of opponent models. This favoured the selection of robust strategies that were successful against a variety of opponent models for the response library (RL). In the current work, the set of opponent models contains only 2 strategies; one using an ‘Attack Wounded’ strategy, and the other using an ‘Attack Closest’ strategy.
6.3.2.4 Unit Behaviours and Grouping

A behaviour describes what action the unit should take in any given circumstance. A behaviour is a high-level script consisting of a series of actions which a unit executes in sequence, moving on to the next action when the previous action is complete or appropriate conditions are met. For each behaviour, a primary action is identified, and a secondary action which is applied if multiple targets are identified for the primary action. For example, if ‘Attack Wounded’ is the primary action, and all enemy units have the same health, then the secondary action ‘Attack Closest’, is applied. Behaviours are described in Table 6.1.

In order to allow the S&R agent some flexibility in terms of choosing and targeting particular units or types of enemy units, the agent is provided with the ability to separate the enemy into groups.

When setting up a simulation, not only are a random set of behaviours assigned to the agent’s units, but the enemy is divided into 4 random groups. Actions are then made specific to groups. For example, the generic "Attack Closest" behaviour becomes "Attack Closest in Group 1". Commands to attack specific groups are referred within Table 6.1 as G1, G2, G3 and G4. Grouping allows the agent to create plans that can focus fire individual or groups of units. This greatly reduces the degrees of freedom contained by the solution search space, allowing an agent to more effectively explore for solutions. This can allow the agent to find an effective solution more quickly.

6.3.3 Response Library Component (RL)

The S&R agent receives its Recall ability from the use of the Response Library. The Response Library is responsible for the storage and communication of the
recorded best responses from the Search simulations. The RL database is updated asynchronously by the CS component and queried from within the game loop by the RPC. It acts as a constantly growing and improving database of best-seen responses to recorded tactical situations.

In this work, using game state descriptors based on the number and type of units (see Section 6.3.2.1), the RL case-base database size is determined by the number of units of each unit type for each player within a scenario. The upper limit for the number of recorded cases in each scenario (see Section 6.4) is given by Equation 6.1:

\[
Scenario\_Number\_Cases\_Upper\_Limit = Player_1 \cdot Player_2 \quad (6.1)
\]

\[
Player_x = (\prod_{i=1}^{Num\_Types_x} (Num\_Unit\_Type_x(i) + 1)) - 1 \quad (6.2)
\]

A) Player 1 has 1 unit type of 3 units. Player 2 has 1 unit type of 3 units. The upper limit of cases is 9.

B) Player 1 has 1 unit type of 2 units. Player 2 has 1 unit type of 6 units. The upper limit of cases is 12.

C) Player 1 has 2 unit types, both with 3 units. Player 2 has 2 unit types, both with 3 units. The upper limit of cases is 225.

D) Player 1 has 1 unit type of 8 units. Player 2 has 1 unit type of 8 units. The upper limit of cases is 64.
6.3.3.1 Game State and Response Descriptors

Preliminary testing identified that actual game state needed to be generalised for successful game state matching to occur. Furthermore, only a small number of game state attributes were required for the agent to adapt competently. Hence, the attributes used to identify game state within the RL include only the number and type of each unit involved in the current scenario. Adding more detailed game state descriptors such as those describing unit health or position causes an explosion of possible states, this drastically shortens the time that a match remains in a particular state, and makes it difficult to match the current game state with a state recorded in the response library.

Describing game state by only the number and type of units involved results in relatively stable states that recur sufficiently frequently to make matching effective, and balances the frequency of response adaptation. This approach effectively forces the chosen response to change only to when units are removed from or added to the game. The similarity metric used in this study is a Boolean value based on matching unit numbers and types. This is appropriate for the simple game state descriptor (based on unit numbers and types) used in the study. If a match is not detected then the recall component returns a random response and flags that no match was found. The agent changes its actions according to the new response or continues with its current actions if no match was found. If no match occurred, but no response is currently chosen then the random response is used to determine initial actions. Implementation details can be seen in Algorithm 7.

When a new response is loaded from the response library and assigned to the RPS, the RPS overrides the previous response with the new one. This represents the adaptation component of the S&R approach which is able to use the retrieved response without modification as it is a complete match against the current game
In addition to the game state information that is used as a key in the response library, each entry in the response database records the behaviour assigned to each unit, and the groupings assigned to the enemy units.

Response behaviours do not correspond with BWAPI commands: they need to be mapped into BWAPI commands by the simulator associated with the RPS.

Algorithm 7 Search and Recall

1: **procedure** RECALL(allied_Units, enemy_Units)
2:   No_Ally ← allied_Units.size()
3:   NoEnemy ← enemy_Units.size()
4:   for all response in All_Records do
5:       if No_Ally == response.No_Ally and No_Enemy == response.NoEnemy then
6:         match ← True
7:         for all id in allied_Units.size() do
8:             if allied_Units[id].type != response.allies[id].type then
9:                 match ← False
10:        for all id in enemy_Units.size() do
11:            if enemy_Units[id].type != response.enemies[id].type then
12:               match ← False
13:       if match==True then return response
14:   return RandomResponse

15: Current_Response = Null
16: **procedure** UPDATE_AGENT(GameState)
17:   setConcurrentSearchGameState(GameState)
18:   playbackSimulator.setGameState(GameState)
19:   newResponse ← ResponseLibrary.Recall(GameState.allied_Units, GameState.enemy_Units)
20:   if (newResponse==RandomResponse and Current_Response == Null) or newResponse ≠ RandomResponse then
21:       playbackSimulator.response ← newResponse
22:   return AlliedActionList ← playbackSimulator.simulateSingleFrame()
6.4 Experimental Setup

The following experiments contain four tactical scenarios that an agent cannot resolve with a singular response. These are illustrated in Figure 6.2 and listed below:

A) 3 Zealots vs. 3 Vultures (Attack Closest agent): This scenario pits 3 fast ranged units (Vultures) controlled by the agent against 3 slow close attack units (Zealots). This scenario favours the kiting strategy as it is extremely difficult to solve without it.

B) 6 Fast Zerglings vs. 2 Dragoons (Attack Closest agent): This scenario pits 2 strong ranged units (Dragoons) controlled by the agent against 6 fast close attack units (Fast Zerglings). Once again, a kiting solution is favoured, but far more precision is required to make this work.

C) 3 Zealots and 3 Dragoons vs. 3 Zealots and 3 Dragoons (Default AI): This is a symmetrical scenario pitting ranged (Dragoons) and close attack (Zealots) units against each other. Precise control over which unit attacks which enemy unit, as well as unit placement, is required to be successful.

D) 8 Dragoons vs. 8 Dragoons (Default AI): Once again this is a symmetrical scenario that pits equal numbers of ranged units against each other. Control
of attack strategy is important in this scenario, but unit placement is less important than in Scenario C.

The experiment described here was designed to evaluate the S&R method against similar scenarios as described in (Churchill and Buro, 2012). However, the experimental setups for scenarios A and B differ from Churchill’s implementation due to problems encountered with the default Brood War AI’s behaviour. The Brood War AI would not continue to respond to an attacking unit allowing it’s behaviour to be exploited. To eliminate this problem, the Brood War default AI was replaced with a scripted agent designed to constantly attack the closest unit in Scenarios A and B.

Churchill’s experiments with his ABCD (Alpha-Beta search Considering Durations) agent use the same unit compositions configurations as in this study (Scenarios A-D). His results are not replicated, instead they are listed from his publication (Churchill and Buro, 2012) (“Churchill ABCD Search” in Table 6.2).

Two scripted actions ‘attack wounded’, and ‘kite’ are provided to the search algorithm. These are similar to the scripted actions used by (Churchill and Buro, 2012). Another scripted action ‘attack closest’ is also provided. Other actions utilised in the search algorithm include ‘attack closest in group’ for four groups arbitrarily distinguished in the enemy. Scripted actions are widely used (Justesen et al., 2014; Wender and Watson, 2014; Spronck, Sprinkhuizen-Kuyper, and Postma, 2004; Barriga, Stanescu, and Buro, 2015; Churchill and Buro, 2013) to provide complex actions tailored for a particular domain. They increase the likelihood of an agent selecting a successful behaviour.

Each scenario is run against a particular configuration of the S&R agent for a total of 200 games at an acceleration of 5ms per frame. This is necessary since due to stochastic variation between games, the outcome of an actual game is not
completely deterministic. The scores recorded in Table 6.2 are defined by the following equation to the nearest percentage.

\[
Score = \frac{\text{wins} + \frac{\text{draws}}{2}}{200}
\]  

(6.3)

<table>
<thead>
<tr>
<th>Behaviour</th>
<th>Primary Function</th>
<th>Secondary Function</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attack Closest in Group. (G1, G2, G3 and G4)</td>
<td>Attack unit of least health in group X</td>
<td>Attack closest unit in group X</td>
<td>No units in group X</td>
</tr>
<tr>
<td>Attack Closest</td>
<td>Attack closest unit</td>
<td>Attack unit of least health</td>
<td>N/A</td>
</tr>
<tr>
<td>Attack Wounded</td>
<td>Attack unit of least health</td>
<td>Attack closest unit</td>
<td>N/A</td>
</tr>
<tr>
<td>Kite</td>
<td>Attack unit of least health in range when ready to fire</td>
<td>Move away from all enemies and terrain</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 6.1: Behaviour Descriptions

The experiments compare several different configurations of the S&R agent. The performance of the Basic (BAS) and the Complex simulator engine (COM) are compared in two modes: in pure Search mode (i.e., without access to any stored responses), and in combined Search and Recall mode (with access to stored responses). This tests whether there is any advantage in retaining results from earlier simulations. For comparison purposes, the performance of two scripted agents was also evaluated: one based on an ‘Attack Closest’ strategy, and another agent ‘Kiter’ which favours Kiting. Each configuration or agent is tested on the four scenarios listed above.

For the S&R agents, each configuration is initialised with a new empty response library at the beginning of the evaluations for all scenarios. All recorded responses are generated by simulations run during the actual games.
6.5. Results and Discussion

The results of the experiments for the scripted agents show clearly that to do well in all four scenarios required adaptive agent behaviour. The ‘Attack Closest’ scripted agent performs poorly in scenarios A and B, but is successful in scenarios C and D, while the reverse is the case for the ‘Kiting’ scripted agent.

The kiting agent performs poorly in scenarios C and D even though it is the only behaviour that allows for units to move away from enemy units. These scenarios present mirror match ups where there is no movement advantage between the

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.81</td>
<td>0</td>
<td>0.96</td>
<td>1.00</td>
<td>0</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0.65</td>
<td>0</td>
<td>0.65</td>
<td>0.92</td>
<td>0</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0.95</td>
<td>0.95</td>
<td>0.80</td>
<td>0.76</td>
<td>0.94</td>
<td>0.77</td>
<td>0.26</td>
</tr>
<tr>
<td>D</td>
<td>0.96</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.97</td>
<td>0.14</td>
</tr>
</tbody>
</table>

Table 6.2: Success rates of different agent configurations against ‘Attack Closest’ scripted agent in scenarios A and B and against the default Brood War AI in C and D. BAS: Basic simulator (no collisions or terrain, no kiting), COM: complex simulator (collisions and terrain plus kiting). Search: search only component. S&R: Search and Recall components. (Churchill and Buro, 2012) ABCD search results against a similar scripted ‘Attack Closest’ agent is shown for comparison in Column 2. The success of a standard scripted agent with a primary action of Attack Closest and a secondary action of Attack Wounded is shown (Attack Closest) in Column 7. The success of an agent based on Kiting only is shown in Column 8.

All S&R experiments utilise 4 threads within the SC for running simulations. Each search was limited to 2000 time steps although this number of steps was never reached. The results of the experiments are shown in Table 6.2.
allied and enemy units. Thus, using an action to move away from an enemy instead of attacking is generally a suboptimal action.

Results for the Basic Simulator, which does not have a kiting behaviour available are similar to the 'Attack Closest' scripted agent. This illustrates the importance of the simulator model containing a set of behaviours sufficient to cover what is required in a scenario.

On the other hand, results in scenarios A and B for the Complex Simulator show that agent clearly discovered and utilised the appropriate kiting behaviour. Results in Scenario A are stronger than in Scenario B, likely because the large speed difference between Vultures and Zerglings makes a wide range of successful kiting solutions relatively easy to find. In Scenario B, if the Dragoons performed a suboptimal action for even a small period they would lose to the larger numbers of Zerglings.

The results for the Complex Simulator with Recall enabled are better than for search alone, indicating that the recall capability provides a considerable advantage. The advantage conferred by the recall ability is much greater in Scenario B than in Scenario A. This suggests that the advantage of accumulating knowledge in the response database is greatest when solutions are relatively exact, and the exploration of the solution space is relatively slow.

Results for the Basic Simulator are equivalent or better than the Complex Simulator for scenarios C and D. This indicates that the range of behaviours available to the Basic simulator are sufficient in these scenarios, and that the complexities introduced for the Complex Simulator have little impact in these scenarios. This result is not terribly surprising since the influence map affects only the kiting behaviour which is not necessary for these scenarios, and the close ranged combat,
and lack of terrain features in these scenarios reduces the impact of the pathfinding capability of the Complex Simulator. Given these considerations, it may be that the much greater number of simulations that the Basic simulator can perform (2000 vs. 200 per second) allows it to find better solutions than the Complex Simulator.

Results for Scenario C are the most varied. The winning solutions for this scenario required more complex behaviours than in the other scenarios. Scenario C is similar in some respects to Scenario B with its rigorous success requirements. Results for the Complex Simulator in Scenario C show a large difference between Search only and combined Search and Recall. Once again it appears that that the Recall capability becomes a significant advantage when solutions are hard to find, and the exploration of solution space is slow.

Results degrade when Recall is enabled for the Basic simulator. It is likely that this is an example of the effects of divergence. The simulator has discovered an action set that is effective in simulation, but that does not translate well into the actual game. This indicates the importance of the simulator’s combat model being a close match to the actual game’s.

Results for Scenario D are both extremely strong, and uniform across both the Basic and Complex Simulators, both with and without Recall enabled. This is probably a result of the scenario being relatively easy to solve, as indicated by the strong result also generated by the ‘Attack Closest’ scripted agent.

Over all scenarios, the strongest performance is shown by the Complex Simulator with Recall enabled. This configuration of the S&R agent adapts strongly to all scenarios, even though its performance without Recall enabled is relatively weak. The result is important since it indicates that the build-up of experience over many game cycles becomes more beneficial when solutions are hard to find,
and simulation rates are slow. This is exactly the situation faced when attempting to apply accurate simulation models to complex commercial grade RTS AI problems.

Note that for all the Search based configurations, results between zero and one are an indication of divergence, since the simulations return what they estimate as a winning solution or a loss. Solutions that win sometimes reflect differences between what the simulators calculate and what actually happens in Brood War. This impacts weaker solutions to a greater extent, resulting in lower scores where search is less effective.

It is an important result that the scores for the Complex Simulator with Recall enabled are consistently high across all scenarios. This reflects relatively little divergence between what the Complex Simulator predicts and what happens in Brood War, given a sufficient accumulation of simulations, and the capacity to retain the results. Given the divergence interpretation of evaluation scores, in comparison with the simulator used in (Churchill and Buro, 2012) the results suggest that the Complex Simulator provides a much closer approximation of the Brood War combat mechanics, and that the predictions made by the Complex Simulator are much more accurate.

An important result is for that across all scenarios, the results for the Complex Simulator with Recall enabled are much stronger, than without Recall. The simulation rate of the Complex Simulator is quite low (200 sims/thread/sec). Without Recall, the benefits of more accurate simulations are lost, because the number of possible solutions examined is so small. However, with Recall, the benefits of accurate simulation become marked. In comparison with Churchill’s (2012) results, the Complex Simulator with Recall enabled dominates by a large margin in all but Scenario C, where it is only marginally weaker. This is remarkable given that the Search component in this study uses only random sampling while Churchill’s
search component used ABCD search. This suggests that the benefits of using slow accurate simulations and accumulating the results across game cycles outweighs the advantages of using sophisticated, fast, directed search algorithms restricted to run within a game cycle.

Another important result is that the benefits of Recall are delivered to the agent relatively quickly. There is a marked improvement for the Complex Simulator with Recall enabled in the difficult scenarios even though the scenario is evolving in real-time. Once again, this indicates that the advantage of receiving high-quality solutions outweighs the disadvantage of them taking more than a game cycle to calculate.

6.6 Conclusions and Future Work

The results of the study support the notion that responses contained CBR databases can be usefully generated through Search. Overall, the results of this study can be summed up as: high-quality responses are worth remembering when solutions are hard to find, the exploration rate of the solution space is low, and when the fidelity of the simulations with the target game is high.

The results strongly indicate that retention of results from search simulations is worthwhile and that Search and Recall is a useful approach. S&R provides a first step towards eliminating the need for a huge and uneven quality database of pre-played games on which to base CBR, and allows the situations a game AI can respond to intelligently to grow over time. At the same time, it guarantees fast decision making within the game loop.
An important implication of the proposed architecture is that because simulations are decoupled from the game loop, they become amenable to parallel, distributed, or offline processing. The exact actual game states sent to the CS component, could instead be sent out over the network, or logged for later processing. Regardless of whether results arrive in time to advantage the S&R agent in the current game, the results generated would improve the response database over time. Another implication is that simulation results from many separate instances of a game can be shared between games, allowing games to cooperate in improving the AI for all games.

A final implication is that simulations are not restricted to the CPU capacity of an ordinary gaming PC. Simulations could be conducted on server farms or supercomputers in the cloud, and the results used to update a global database available to all instances of a game.

Because the constraints on execution times and hence simulation complexity have been eased, future work could extend simulation models to scenarios of greater complexity such as working with terrain and larger unit encounters. It would also be interesting to explore the feasibility and utility of more detailed game state descriptors, and the associated much larger response databases required.

Once response databases become larger and more populated, game progression paths through state space and discovering general patterns of game progression could prove interesting. The sensitivity of results to the range of available behaviours also indicates that further work into more complex behaviour sets is also warranted.

S&R removes computational execution time restrictions on search but retains the ability of search based agents to adapt to new situations. The S&R agent model allows simulators used in searches to use much more complex models to deal
with complex tactical situations. Simulators can include pathfinding, unit and terrain collision avoidance, and specialised behaviours. These complex simulators greatly improve the fidelity of the results produced, which reduces the divergence between predicted outcomes and those produced by the game. This makes the S&R method potentially useful in applying search techniques to commercial grade levels of combat scenario complexity.
Chapter 7

Advanced Tactical Agent - Dynamic Granularity

7.1 Introduction

Chapter 6 established that search could be decoupled from the game loop and the results of search communicated to a game agent through a CBR database. It established that a recall mechanism is useful in improving agent behaviours, particularly when solutions are hard to find, and the rate of exploration of the solution space is low. It also found that the fidelity of the simulations used to generate solutions had a major impact on the utility of those solutions when applied in the actual game. However, rapid effective searching is still the key to exploration of high dimensional search spaces and these problems remain, regardless of whether the search process is conducted within the game loop or not.

This chapter focusses on improving the efficiency and effectiveness of the search process to address the research question: "How can the effectiveness of MCTS for RTS games be improved?". It presents a mechanism of discovering the optimum number and composition of unit groups using MCTS. It proposes a way of structuring the MCTS search in such a way that simple groups of large numbers of
units are likely to be found early in the search process.

Recent research has examined the application of Monte-Carlo methods to the solution of tactical problems (Justesen et al., 2014; Zhe et al., 2012; Balla and Fern, 2009). This is because Monte-Carlo methods provide a general mechanism for solving complex problems, while the development and application of heuristic based approaches require considerable domain knowledge, the tailoring of solutions to individual scenarios, and in the end, require expert knowledge, possibly leading to non-optimal solutions.

However, Monte-Carlo methods have problems dealing with high-dimensional search spaces due to the ‘curse of dimensionality’ which makes exploration of the search space extremely slow. The dimensionality of the search space is given by the formula:

\[ D = L^U \]  (7.1)

Where: D is dimensionality, L is the average number of actions per unit, and U is the number of units in the game (Churchill and Buro, 2013). One of the main research problems in the application of Monte-Carlo methods to solving tactical scenarios in RTS game is the reduction of the game state space (complexity).

Addressing the number of units involved in a simulation has the greatest impact in reducing the dimensionality of the search space since it is unit number that controls the power term in Equation 7.1.

(Balla and Fern, 2009) pioneered the application of UCT (Upper Confidence bound applied to Trees) in RTS games. They applied UCT in the simplified Warcraft clone Wargus in scenarios containing up to 16 units. Units were variously grouped using spatial proximity into between 2 and 4 groups before each scenario. All
units were the same (footman) and only two actions were allowed: attack, and join, with combat controlled by the default AI.

(Churchill, Saffidine, and Buro, 2012) developed a fast mechanism to deal with durative moves and apply Alpha-Beta search in RTS games. They noted issues simulating using BWAPI due to speedup limits in Brood War, and issues relating to simulating every game frame. They developed their own simulator SparCraft which closely emulated combat in Brood War with some limitations and could execute independently from the game. They also developed a mechanism for ‘fast-forwarding’ between frames where new action decisions were possible for either player. This mechanism has been widely adopted in other search based algorithms as a way of handling durative actions. They also found that evaluating search results using playouts based on scripted agents performed much better than using evaluation functions such as Life-Time Damage (LTD). They found that their search algorithms could outperform commonly used scripts for relatively small combats (8 vs 8 scenarios), but faced problems dealing with larger numbers of units due to the ’enormous state and move complexity’. They proposed using spatial and unit group abstractions to reduce state spaces of more complex scenarios to a level that their fast heuristic search could deal with. They noted Balla and Fern’s work, but remarked that Balla and Fern’s UCT search was rather slow, and that their groups were preassigned.

(Justesen et al., 2014) also used the UCT algorithm to apply a Monte-Carlo approach to solving tactical scenarios. They used complex scripts to simplify the number of possible actions and searched for successful sequences of scripts. They also applied grouping to reduce complexity but in contrast to (Balla and Fern, 2009) they determined groups using K-Means clustering based on unit type and physical proximity. They found considerable improvement in success rates in scenarios involving more than 32 units, but also found that the clustering approach
was less effective in small combats.

(Ontanón, 2013) applied a Combinatorial Multi-Armed Bandits (CMAB) formulation to the MCTS method. This approach switches between exploring the action space selecting by selecting actions for units independently, and exploiting favourable combinations of actions, based on the notion that combining favourable independent actions results in a favourable combination of actions. The notion that individual actions can be selected independently is referred to as the ‘naive assumption’ and the tree strategy based on this assumption is referred to as Naive Sampling. The benefit of Naive Sampling applied in the context of CMAB (NaiveMCTS) is that if a given unit action is found to be beneficial on average, then player actions containing that action are more likely to be sampled. The overall benefit of NaiveMCTS is that it finds successful strategies faster than standard MCTS, so it is more effective in high dimensional search spaces, given highly constrained time windows for search.

(Churchill and Buro, 2013) used a technique called ‘Portfolio Greedy Search’ that uses scripts and a hill-climbing procedure for search. They showed that their hill-climbing algorithm outperformed UCT and ABCD search in scenarios with large numbers of units (32 -50). In particular, they found that the complexity of their algorithm grew linearly with the number of units rather than exponentially as is the case for Monte-Carlo search. While extremely interesting, their method is not applicable to the current research which addresses the issue of dealing with dimensionality in Monte-Carlo methods. It is also unclear whether their hill-climbing approach will find the global optimal solution as guaranteed by Monte-Carlo search (in the limit), or get trapped in some local optima.

(Uriarte and Ontañón, 2014) reduced game state complexity by defining a simplified abstract game state. Their game state was based on decomposing a game map into regions and grouping units based on which region of the map they
were within and their type. They used three actions in their work: move to a
different region, attack a region or wait. Similarly to the earlier work of (Balla
and Fern, 2009), they defer combat decisions to the default AI. The state space
representation developed in (Uriarte and Ontañón, 2014) was applied in (Uriarte
and Ontanón, 2014). They used ABCD search and a new variant of MCTS, MCTS
Considering Durations (MCTSCD) to choose actions based on evaluations of pre-
dicted high-level game states. They found that the use of high level game states
reduced the branching factor of RTS games while still producing meaningful ac-
tions.

(Barriga, Stanescu, and Buro, 2015) introduced a technique called ‘Puppet Search’.
This technique drastically reduced the search space in a game by exposing only
a few chosen ‘choice points’ within complex non-deterministic scripts. Any type
of search algorithm can be applied to determine what choices should be made at
each choice point. Puppet search relies on having detailed and successful strate-
gies implemented as scripts with choices limited to choosing branches within a
script. In the example provided, four types of rush strategies are implemented,
combined into a single script, and a single choice is provided to select which
strategy is chosen at the beginning of the script.

(Ontanón and Buro, 2015) applied Hierarchical Task-Network Planning in RTS
games to create a technique called ‘Adversarial Hierarchical-Task Network Plan-
ning’ (AHTN). AHTN recognises high level tasks which can be strung together.
Each task has a method, which can contain subtasks. AHTN drastically reduces
the search space for games because it constrains search to the number of possi-
ble HTN decompositions available to each player which is expected to be much
smaller than the number of raw sequences (of primitive actions). AHTN relies on
the availability of ’effective task decompositions’; that is, it relies on expert do-
main knowledge in regard to the structure of possible methods (ie identity and
sequence of subtasks) each non-primitive task may use. In the example provided, the authors use AHTN to manage a complete game within microRTS.

The discussion above indicates that there have been two main themes in dealing with the problem of high dimensionality for MCTS search in RTS games. The first has been to reduce the number of units through clustering. However, the approaches tried so far have either arbitrarily assigned units to a specified number of clusters, or based unit clusters on spatial proximity. The other approach has been to reduce dimensionality through the use of complex high-level scripts. These reduce the action space and reduce the depth and branching factor of the tree that needs to be searched.

The work described in this chapter contributes to dealing with high dimensionality in MCTS by reducing the effective number of units involved in search. It proposes a new structured sampling policy which dynamically determines the optimum number of unit clusters. The approach is referred to as Dynamic Granularity (DG). It differs from previous approaches to reducing unit numbers through clustering in that the number of clusters is optimised as part of MCTS search, while the sampling strategy is such that low numbers of clusters are favoured. The main advantage of the technique is that effective solutions involving low numbers of unit clusters are more likely to be sampled early in the search procedure, making it more likely that successful solutions will be found in a time constrained search.

Dynamic Granularity (DG) differs from the approach used in (Balla and Fern, 2009) in that they start with the maximum number of groups possible, and discover advantageous larger groupings through exploration. The Dynamic Granularity (DG) approach starts with a single large group and discovers advantageous divisions. Also, in this work search is extended to control combat through choosing from a number of generalised or specific attack actions.
In contrast to (Justesen et al., 2014), DG does not assign an arbitrary number of clusters. Instead, DG dynamically determines the best grouping for a particular scenario, retaining the dimensional reduction advantages of grouping for large, many unit scenarios, while allowing more detailed exploration in smaller scenarios with fewer units.

(Uriarte and Ontañón, 2014; Uriarte and Ontañón, 2014) define groups based on map regions and unit types, and leave combat micro to the default AI. This contrasts to the DG approach where optimum group numbers and compositions are determined through search and full combat responses are defined. Furthermore, in this work DG is built on a complex simulator that utilises pathfinding, collisions and terrain where (Uriarte and Ontañón, 2014; Uriarte and Ontañón, 2014) used the SparCraft simulator which does not support these features.

In this chapter, a novel mechanism (DG) is proposed for addressing the issue of complexity reduction. The proposed mechanism uses scripting to reduce the number of actions, but most importantly it proposes a top-down splitting approach to grouping within a layered MCTS action selection framework to determine the appropriate number and composition of unit clusters. The mechanism directly targets the factor most responsible for high dimensionality: large effective unit numbers. The proposed mechanism is intended to find effective groups to support coordinated actions faster than a standard MCTS.

### 7.2 Dynamic Granularity

Dynamic Granularity (DG) takes a top-down approach to discovering the appropriate number and composition of groups, and sets of actions involved in an individual solution to be evaluated in MCTS search. DG organises units into groups
which facilitates finding coordinated solutions. This is important because coordinated actions can be more effective than individual actions. Grouping units becomes more beneficial the larger the combat engagements become. DG uses MCTS to determine the best grouping structures. This enables little or no grouping in small combat scenarios with only a handful of units. This is important as these scenarios represent significantly smaller state spaces and individual unit actions can be more fully explored to find more optimal solutions. DG allows the agent itself to determine how to best split its army and efficiently find responses using different group compositions.

DG recognises three separate categories of action, which are arranged in a hierarchy. The categories are grouping actions, general actions, and specific actions. These are illustrated in Tables 7.1, 7.2, 7.3 respectively.

At the top of the hierarchy are grouping actions. These actions determine the granularity of the solution. Grouping actions are chosen iteratively until an arrangement of unit groupings is selected for a particular solution. Iterations continue until the grouping actions 'leave group as it is' or 'split group into individual units' are selected. Simple solutions are favoured since two of the four possible grouping actions result in either no further increase or a reduction in group complexity.

Once unit groupings are determined, actions in the next level of the hierarchy, generalised actions, are selected. All generalised actions consist of complex scripts except for 'attack specified unit'. There are two types of generalised actions, terminating, and non-terminating. Terminating actions include 'hold position', 'move to bottleneck', 'kite', and 'attack specified unit'. If a terminating action is chosen, then the group will begin the action selection process again from grouping actions. In this way sequences of actions can be created.
'Hold position', 'move to bottleneck', and 'kite' are examples of complex behaviours requiring scripts, rather than simple actions. 'Hold position' requires a unit to remain in a single location for a specified duration, attacking enemy units that come within range. 'Kite' requires a unit to attack the most wounded enemy within range, but to move out of the enemy’s range after it attacks. 'Move to bottleneck' requires a unit to move to predefined locations on the game map identified as 'bottlenecks'. Bottlenecks are highly defensible choke points on a map which restrict unit movement and access for attacking units. Such choke-points only have meaning and significance if unit collisions and terrain impassibility are considerations in the solution. For the purpose of this work, bottleneck locations are hard coded, but future implementations could evaluate positions of advantage based on analysis of maps such as in the work of (Perkins, 2010) and utilised in (Uriarte and Ontanón, 2014).

If the general action ‘attack specific unit’ is chosen, then search continues to the third level, selecting individual specific actions. In this layer, a choice is made as to which specific unit is to be attacked. This is represented in the ‘specific action’ level of the hierarchy by the ‘attack specified unit’ command. There are as many ‘attack specified unit’ commands as there are units in the opposing force. All specific action commands are terminating commands.

There are two non-terminating actions: ‘kite’ and ‘attack closest’. If a non- terminating action is chosen, then that action continues for the duration of the game. Nonterminating actions are the last action in the sequence of scripts chosen for that particular grouping of units.

After a potential solution is evaluated through simulation, the selection probabilities of each branch in the command tree are updated using standard MCTS back-propagation techniques. In this way, MCTS can explore different groupings and sequences of commands for each group. The command type hierarchy favours
Chapter 7. Advanced Tactical Agent - Dynamic Granularity

<table>
<thead>
<tr>
<th>Grouping Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leave group as is</td>
</tr>
<tr>
<td>Split group into individual units</td>
</tr>
<tr>
<td>Split group in half along x axis</td>
</tr>
<tr>
<td>Merge Groups</td>
</tr>
</tbody>
</table>

**TABLE 7.1: Grouping Actions**

<table>
<thead>
<tr>
<th>Behaviour</th>
<th>Function</th>
<th>Terminal Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hold Position</td>
<td>Hold position &amp; attack closest unit</td>
<td>Time X has passed</td>
</tr>
<tr>
<td>Retreat to Bottleneck</td>
<td>Move to specified bottleneck location</td>
<td>Time X has passed</td>
</tr>
<tr>
<td>Attack Closest</td>
<td>Attack closest unit</td>
<td>N/A</td>
</tr>
<tr>
<td>Attack Specific unit</td>
<td>Attack specified unit</td>
<td>Specified unit dies</td>
</tr>
<tr>
<td>Kite</td>
<td>Attack most wounded unit in range when ready to fire. Move away from all enemies and terrain.</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**TABLE 7.2: General Action Descriptions**

the early discovery of relatively simple, large scale groupings.

Later exploration of the search space tends to refine command sequences for group arrangements that are associated with higher levels of success.

Once an action reaches its terminal condition then the group selects a new action from the MCTS starting again from the grouping action sequence. In this way, the model achieves the recursive nature of the MCTS method where new decisions are sampled as required starting from grouping decisions then general and possibly specific actions. For example, once initial grouping and general/specific actions have been chosen they are used until a terminating condition is reached by the group. This could occur in the case where the group was targeting a specific unit that was killed. The group then requests new actions from the MCTS starting with grouping actions. The group may be split into more groups, left as is or merged with another group. If the group is merged with another group
### 7.2. Dynamic Granularity

<table>
<thead>
<tr>
<th>Behaviour</th>
<th>Function</th>
<th>Terminal Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attack Specific unit</td>
<td>Attack specified unit</td>
<td>Specified unit dies</td>
</tr>
</tbody>
</table>

**Table 7.3: Specific Action Description**

then all units are given the target groups actions. Otherwise, if the group is not merged, it selects new actions from the general actions, and finally, possibly specific actions.

This process repeats until all enemy or allied units have been terminated.

Details of the Dynamic Granularity search algorithm are given in Algorithm 8.
Algorithm 8 Dynamic Granularity

1: **procedure** START_GAME(allied_Units, enemy_Units)
2:    allied_Units.set_Group(0)  \(\triangleright\) isReady ← False
3:    Update_Groups(allied_Units, enemy_Units)

4: **procedure** UPDATE_GROUPS(allied_Units, enemy_Units)
5:    **for all** allie in allied_Units **do**
6:        if allie.group.isReady == False **then**
7:            \(\triangleright\) Iteratively update grouping until it is merged with a processed group or is finished
8:                repeat
9:                    groupingAction ← MCTS.getGroupingAction()
10:                    allie.group.applyGroupAction(groupingAction)
11:                until (groupingAction != Leave_Group_As_Is) or (groupingAction == Merge_Groups and allie.group.isReady)
12:                \(\triangleright\) If the last grouping action was a merge action then no new action is needed
13:        if groupingAction != Merge_Groups **then**
14:            generalAction ← MCTS.getGeneralAction()
15:            if generalAction == Attack_Specific_Unit **then**
16:                specificAction ← MCTS.getSpecificAction(enemy_Units)
17:                allie.group.setAction(specificAction)
18:            else
19:                allie.group.setAction(generalAction)
20:            allie.group.isReady ← True
21:    **procedure** UPDATE_GAME(allied_Units, enemy_Units)
22:    runSimulation()
23:    **for all** group in groups **do**
24:        if group.actionTerminated == True **then** allie.group.isReady ← False
25:    Update_Groups(allied_Units, enemy_Units)
7.3 Experimental Approach

A set of experiments were designed to evaluate the effect of each level of the command hierarchy. The experiments were performed on a simulated version of Brood War which includes terrain, collisions, and mimics unit damage, fire rates, and unit movement as closely as possible. All experiments were conducted against a scripted agent running the behaviour ‘attack closest enemy’ for individual units, similar to that used in (Churchill, Saffidine, and Buro, 2012).

The simulator used in for these experiments is an extension of the Complex simulator presented in Chapter 6. The simulator was further enhanced to make use of the DG approach described in Section 7.2. The simulator uses pathfinding, collisions and terrain to work within the experimental scenarios. Because a number of scenarios examined in this work require pathfinding, collisions and interaction with terrain, the work presented here is not directly comparable with work based on the SparCraft simulator (which does not support those features, and was not used in any of these experiments), and so no direct comparisons with results gained in that simulator are made.

In each experiment, MCTS was run in parallel across twelve threads to evaluate a total of 30,000 separate grouping and action sequence configurations. The default playout policy (default action after all actions in the chosen grouping and command set are enacted) is ‘attack closest’. Using random playout actions had a significantly negative impact on the search, particularly in regard to specific attack commands because many randomly chosen attack commands contributed no damage to enemy units if the specified enemy unit was not on the front line of battle.

The experiments evaluated all combinations of command hierarchy levels:
1. Grouping with general and specific actions

2. Generalized and specific actions (with no grouping)

3. Grouping with generalized action

4. Grouping with specific action

5. Generalized action only (with no grouping, or specific attack actions)

6. Specific actions only

7. Default action - attack closest

Each combination of command types was evaluated across a range of six tactical scenarios in order to test the diversity of behaviours that could be found using the proposed model. Scenarios 1 and 2 are replicated from Chapter 6, and are the same as Scenarios C and D in (Churchill and Buro, 2012). Scenario 3 is similar to Scenario 2 but involves many more units (25 vs 25 compared to 8 vs 8). Scenario 4 introduces a bias against the agent in the form of a larger opposing army to force discovery of non-obvious or ‘intelligent’ solutions. Bias is defined as the relative value of each agent’s army in terms of resources required to produce it (minerals + gas) and is shown for each experiment. Scenarios 5 and 6 demonstrate situations where pathfinding and collision detection and terrain consideration is necessary.

Scenarios 5 and 6 make use of bottlenecks which are highly defensible choke points such as a bridge over a moat. Bottlenecks restrict movement and make pathfinding difficult or impossible provided that units cannot interpenetrate, and that some terrain is impassable. Fighting within a bottleneck offers a significant advantage to the defending army as it limits the number of enemy units that can attack, thus reducing the impact of the larger attacking armies. Bottlenecks are
essential in Brood War for players to defend against aggressive strategies. However, recognising and exploiting bottlenecks is impossible in simulators (such as SparCraft) which do not implement collision detection or incorporate pathfinding around obstacles. Scenarios 5 and 6 test whether DG can help an agent find solutions based on coordinated attacks from different groups of units depending on the scenario. Furthermore, these experiments also examine whether an agent using DG can take advantage of terrain similarly to human players.

The scenarios consisted of the following:

1. **Scenario 1 - 3Z & 3D - 3 Zealots vs 3 Zealots vs 3 Dragoons** (Simple scenario, 0% bias). See Figure:7.1.

2. **Scenario 2 - 8D - 8 Dragoons vs 8 Dragoons** (Simple scenario, 0% bias). See Figure:7.2.

3. **Scenario 3 - 25D - 25 Dragoons vs 25 Dragoons** (Many units, 0% bias). See Figure:7.3.

4. **Scenario 4 - V & T - 16 Vultures and 4 Tanks vs 15 Zealots and 9 Dragoons** (Asymmetric scenario, Value:2200 vs 3075, Opponent 40% bias). See Figure:7.4.

5. **Scenario 5 - 1B - 28 Marines vs 79 Fast Zerglings on Terrain with a single bottleneck** (Many units, terrain effects, asymmetric scenario, Value:1400 vs 1975, Opponent 41% bias). Zerglings are upgraded with metabolic speed (Move much faster than standard Zerglings). The MCTS army starts in an exposed position that the fast melee enemy units can take advantage of. See Figure:7.5.

6. **Scenario 6 - 2B - 48 un-upgraded Zerglings vs 18 Marines on Terrain with a two bottlenecks** (Many units, complex terrain effects, asymmetric scenario,
Value: 1200 vs 900, Allied 33% bias). In this scenario, the enemy units are established within a bottleneck. The scripted enemy AI in this scenario will not move its units so that it maximises the advantage of its position within the bottleneck. It should be noted in this scenario that even though the MCTS agent possesses a larger army of Melee units, the enemy army of ranged units still possesses a large advantage given their position within the choke point. The scenario is illustrated in Figure: 7.6.

**Figure 7.1:** Scenario 1. Red: MCTS, Green: Opponent. Simple Mirror Match. 3Z & 3D: 3 Zealots and 3 Dragoons

**Figure 7.2:** Scenario 2. Red: MCTS, Green: Opponent. Simple Mirror Match. 8D: 8 Dragoons vs. 8 Dragoons
7.3. Experimental Approach

Figure 7.3: Scenario 3. Red: MCTS, Green: Opponent. Mirror Match: 25D: 25 Dragoons

Figure 7.4: Scenario 4. Red: MCTS, Green: Opponent. V & T: 16 Vultures and 4 Tanks vs. 15 Zealots and 9 Dragoons. Opponent 40% bias
In Scenario 6, the enemy units are established within a bottleneck. The scripted enemy AI in this scenario will not move its units so that it maximises the advantage of its position within the bottleneck. It should be noted in this scenario that even though the MCTS agent possesses a larger army of Melee units, the enemy
army of ranged units still possesses a large advantage given their position within the choke point. The scenario is illustrated in Figure 7.6.

In Scenario 5, the only way the marines can win is to retreat to the bottleneck where the numerical superiority of the Zerglings is negated, and the ranged damage dealt by the marines can be exploited. It is exceptionally difficult for standard ungrouped Monte-Carlo solutions to find this solution since it only works if all units retreat to the bottleneck at the same time.

In Scenario 6, the only viable solution is to split the attacking force, direct part of the force through the second bottleneck, and then mount coordinated attacks on the fixed enemy from both sides of the first bottleneck. Once again, this solution is exceptionally difficult to find using individual commands due to the complexity of the command space. It requires an ability to issue separate sequences of commands to different groups of units while coordinating their actions.

Each scenario was run 10 times. The simulator ran in parallel and performed 6 playouts concurrently to speed up evaluations. The experiments were run on a Windows 10 64-bit machine with 32gb of ram and 4 cores at 3.3ghz. Additionally, the simulator used an adaptation of the BLJPS2 algorithm modified to work in dynamic environments, which significantly improved simulation times over using BLJPS.

The actions used in this set of experiments are an extension of those presented in Chapter 6 with the removal of the ‘Attack Wounded’ action. Two additional scripted actions were added. The ‘Hold Position’ action represents a basic action where the unit will not move but will attack enemies within range. This action simplifies coordination between groups, where engaging an opponent at the same time from different angles maximises the effectiveness of the attack. The ‘Retreat to Bottleneck’ action represents a high-level movement action where
an agent can move to pre-analysed map positions. This action significantly re-
duces the search space to coordinate and flank enemies without considering ev-
ery movement action. Both of these actions are common RTS actions that play-
ers use frequently throughout their matches. Additionally, the agent can always
use the most basic action of attack enemy unit at any point instead of a scripted
action. The agent can therefore choose whichever action policy maximises its
performance. This approach to utilising high level scripts is similar to that used
in Puppet Search (Barriga, Stanescu, and Buro, 2015), Portfolio Greedy Search
(Churchill and Buro, 2013), and in Script and Cluster Based UCT (Justesen et al.,
2014). However, this work also allows for primitive attacking actions, where a
specific individual enemy can be targeted.

7.4 Results and Discussion

The results of evaluating the different command hierarchy combinations across
the six different tactical scenarios are shown in Table 7.4. Results are given as
the percentage of the teams remaining hit points. A victory with all allied units
taking no damage will result in a score of 100%. A loss where no enemy units
take any damage will result in a score of -100%.
Table 7.4 shows that the complete command hierarchy gave the best results in every scenario. While specific commands alone worked well in the simplest scenario, specific attack commands alone gave the poorest results in every other scenario. Specific and general commands performed similarly, giving a relatively good performance in simpler scenarios. However, as soon as the tactical scenario became difficult (i.e. required discovery of relatively low probability solutions), the specific, general, and specific and general command combinations failed badly. The introduction of dynamic grouping to either of the other command types significantly improved the performance of the agent. The effect is most pronounced when combining dynamic grouping with general (i.e. high level scripted) behaviours. Combining all three command types gave the best results. In contrast, the results show that the baseline ‘Attack Closest’ scripted agent performed the worst amongst all the evaluated methods. This demonstrates that any search method performed better than the single scripted approach.

The results showed that it was exceptionally difficult for MCTS based on individual units to find solutions that required coordinated group actions. This is shown
by the results of the non-grouped agent configurations which perform relatively strongly in Scenarios 1 through 3, but poorly in scenarios 4 through 7. (Churchill and Buro, 2012) work applied the ABCD search algorithm to individual units and the closest equivalent in the current study would be the ‘General and Specific’ agent configuration. As can be seen in Table 7.4, this performs relatively strongly in Scenarios 1 through 3, but fails in Scenarios 4 through 6 where coordinated actions are desirable.

The performance gap in these experiments between the full DG agent configuration with grouping (Grouping, General and Specific) and the agent configuration without grouping (General and Specific) expands in more complex scenarios. This result is similar to that obtained by (Ontanón, 2013) using NaiveMCTS which gained a significant advantage over UCT as the branching factor grew. This is also similar to the results obtained by (Ontanón and Buro, 2015) using AHTN.

In the case of Scenario 5 (retreat to bottleneck), early explorations are likely to result in one or only a few units being tasked to retreat. This gives a poorer result than if all units perform some type of attack. However, the poorer results of a partial retreat reduce the probability of solutions involving retreat being explored. However, when the dynamic granularity approach is used, early exploration favours coordinated actions, and the advantage of a combined retreat is easily discovered. Similarly, it is extremely unlikely that the solution to Scenario 6 (flanking to attack bottleneck) will be found through testing random combinations of commands to individual units, or even clusters of units. Once again DG provides a way of discovering favourable unit groupings early in the search process.

The most notable feature is that DG improved the performance of the agent in all scenarios, even simple scenarios where a search on individual commands might
be thought to be able to discover very specific advantageous attack patterns. It seems that the ability to dynamically discover an optimum grouping of units and then to be able to issue commands for coordinated action for members of a group is a significant advantage. Also, the approach of progressively increasing the complexity of the search space by incrementally refining the granularity of the solutions explored is an effective approach to dealing with the 'curse of dimensionality' in regard to finding solutions to complex tactical problems. This incremental increase in solution complexity explored is in contrast to the approach used by (Balla and Fern, 2009) in which the most complex unit groupings are explored first.

The essence of this method, is the hierarchical approach, whereby unit grouping decisions are at the top of the MCTS search tree, generalised actions come next, and specific attack actions follow. This results in favourable combinations of units being reinforced, with later exploration favouring investigation of different generalised command sequences for the different groupings, and finally different specific attack patterns being investigated around successful high-level command sequences.

In addition, the speed of the simulations was investigated. First, a benchmark using no pathfinding, collision testing or influence maps was measured. This benchmark is similar to the simulators such as SparCraft used by many works described in Section 2.6. Without these components, the simulator runs at an average of 5000 simulations per second (Sims/s) on Scenario 1 (3Z & 3D) utilising 12 threads. When influence maps are enabled, the simulation rate drops to 900 Sims/s. It drops even further to 58 Sims/s when collisions and pathfinding are enabled. The simulation rates for each scenario using the standard experimental set-up (with pathfinding, collisions, and influence maps) are as follows.

1. 3Z & 3D: 58 Sims/s
2. 8D: 108 Sims/s
3. 25D: 33 Sims/s
4. V & T: 14 Sims/s
5. 1B: 3.7 Sims/s
6. 2B: 5.4 Sims/s

These results show the impact of path finding on simulation speeds, particularly in scenarios where it is difficult to find a path. The simulation rate drops markedly as can be seen in the results for scenarios with bottlenecks.

7.5 Conclusions and Future Work

The major factor increasing the complexity of MCTS applied in tactical scenarios is the number of units involved. Arbitrary clustering approaches such as used by (Balla and Fern, 2009) based on the nearest neighbour or unit similarity reduce complexity but lack flexibility. The Dynamic Granularity approach, working as it does from the top down, tends to find the simplest solutions first. It makes it easier to find coordinated solutions involving a small number of unit groupings. However, when a relatively successful grouping and general command sequence is discovered, it facilitates more detailed exploration of more specific variants of those more general patterns.

Later work should examine more generalised approaches to identifying positions of advantage (such as (Perkins, 2010)), perhaps enabling ranged units to locate themselves on high ground, or bottlenecks to be identified through analysis of influence maps rather than the hard-coded approach adopted here. Such an extension could build on the work of (Uriarte and Ontañón, 2014) which utilises
7.5. Conclusions and Future Work

regional and bottleneck information with MCTS within the SparCraft simulator (Without pathfinding). Another area that could be explored would be offering a range of hold durations for hold commands which could improve coordination of different group actions. This could be extended to specifying duration for some non-terminating attack commands, allowing strategies such as ‘attack then hold’, or other more sophisticated battle strategies.

The work in this chapter demonstrates a successful approach to solving complex tactical scenarios involving terrain, pathfinding and collision detection in a variety of scenarios. It demonstrates that a hierarchical command structure prioritising the discovery of an appropriate grouping of units (granularity) at the beginning of the search tree is an effective approach to dealing with the high dimensional search space of tactical combat in RTS games. It facilitates the rapid discovery of coordinated solutions which are difficult for unstructured MCTS approaches to find. However, the experiments also demonstrated that search times including collisions and pathfinding, while necessary to find realistic solutions, are far too computationally expensive to fit within the time constraints of the game loop. Therefore, an offline technique such as that developed in Chapter 6 is essential in order to incorporate such accurate simulators.
Chapter 8

Conclusion

This chapter reviews the work presented in this thesis and how they address the original questions stated in Chapter 1. Each chapter contributed to this thesis’s RTS AI framework which resulted in an advanced tactical agent as explored in Chapter 7.

In regard to the first research question:

"How can game state data be collected from RTS games with either noisy or no game log data?"

The work presented in Chapter 3 demonstrated a screen capture based approach. This approach was generalizable and allowed data to be collected from any game/applications where the state data could be displayed in a 2D format. Chapter 3 showed that the screen capture method could be used to extract game state traces from Starcraft 2 with far fewer errors than a file based Starcraft 2 replay game log parser.

Furthermore, Chapter 3 showed that the screen capture technique was capable of accessing game data that is not logged. This was demonstrated by retrieving Dota 2 hero selections at the start of the game. Furthermore, this showed the technique was applicable to different applications.
Chapter 3 showed that one answer to this research question is to use screen capture to directly capture game state data from the game.

In regard to the second research question:

"How can adaptive tactical RTS AI be improved to more effectively find solutions to complex, previously unseen scenarios?"

This question was extensively addressed in Chapters 4 through 7. The answer gained is compound. The different aspects of the answer may be summarised as follows:

1. By decoupling Search from the processing constraints imposed by the game loop. This result is developed in Chapter 6 with the Search and Recall framework.

2. By integrating Search and CBR approaches to give Search the capability of executing asynchronously from the game loop, and to give CBR the capacity to develop new solutions to novel situations. This result was also developed within Chapter 6.

3. By using high-fidelity simulations which minimise the divergence between simulator results and results obtained in the actual game. This result is demonstrated by the work in Chapter 6 comparing results using the Basic Simulator to those using the Complex Simulator. The simulation environment should include all aspects of the game that affect the choice of action, including unit collisions, pathfinding, and interaction with terrain.

4. By improving the speed of pathfinding in dynamic environments to reduce the processing bottleneck encountered when including pathfinding in high-fidelity simulations. Work carried out in Chapters 6 and 7 both relied on pathfinding improvements developed in Chapters 4 and 5 to increase the
number of potential solutions evaluated under Search. This has a major impact on the quality of solution found.

5. By dynamically choosing an optimum grouping of units when performing MCTS search. This approach facilitates finding effective coordinated player actions early in the search process (i.e. in a relatively low number of samples of the solution space). This result was demonstrated in the work presented in Chapter 7.

In regard to the third research question:

"How can an effective adaptive tactical RTS AI respond successfully with little or no perceptual lag?"

This question was addressed in Chapter 6 by integrating a Case-Based Reasoning (CBR) approach with Search to provide an agent’s response using the Search and Recall technique. Using Search and Recall, an agent could discover new solutions through search and populate a database of responses to be used in similar future scenarios. The framework showed that an agent could always respond with a decision within the time constraints of the game loop, even though developing the best solution to a novel situation might take longer than a single game loop. The solution available to an agent would improve over several game cycles if one was not initially available. The new solution would be available instantly the next time a similar situation was encountered.

The question was further addressed by the work on Dynamic Granularity in MCTS within Chapter 6 which addressed dealing with the large solution space involved in complex tactical scenarios. DG addressed the issue of responding successfully by increasing the chance of finding a successful solution early in the search process. The use of complex simulators also improves the chance of a search based agent responding successfully, although it has a major impact on the
speed of simulations. The simulation speed issue is addressed largely by Search and Recall framework which allows simulations to be conducted in parallel and offline. The pathfinding work in Chapters 4 and 5 that significantly improved simulation rates for complex simulators, also contributing to the likelihood of finding a successful response within a short time.

1. By decoupling search from the game loop so that using such high-fidelity simulators becomes feasible.

In regard to the fourth and final research question:

"How can a simulator’s pathfinding be improved?"

Once again, the simple answer is obvious although the implementation is difficult: By making it faster and capable of dealing efficiently with dynamic environments. This question is addressed clearly in Chapter 4 and Chapter 5 with the development of the BLJPS algorithm and its application in dynamic environments. It is shown that this algorithm outperformed several other pathfinding alternatives and the success of these algorithms contributed greatly to the work involved in answering the previous research questions.

8.1 Future Work

From the work completed in this thesis, multiple interesting future work avenues have presented themselves.

Further work could explore the use of BLJPS 3, 4 and 5 (Chapter 5) within dynamic environments, similar to the work on BLJPS. This would need to focus on if and how well these algorithms could adapt their pre-processed information to a dynamic environment. Further work could also investigate the possible use of flocking and sub-optimal pathfinding algorithms to increase search speed.
8.1. Future Work

Each type of unit requires a separate copy of the BLJPS2 algorithm in Chapter 6. This increases the cost of pathfinding as more types of units are introduced into the environment and thus affected simulation speed. Future work could look at addressing this cost of performing pathfinding for multiple unit types.

Further work could also look at increasing the efficiency of these pathfinding algorithms by caching frequently searched paths. During MCTS in Chapter 7, many branches containing pathfinding were iterated over many times in order to reach lower level decisions. Branches that require a pathfinding query could cache the results to avoid searching for the same solution many times.

Chapter 6 implemented only a very basic version of the S&R framework. Given that the technique is promising, much more work could be done in this area to investigate more complex game state descriptors, more sophisticated matching algorithms, and more sophisticated mechanisms of populating the game state/response library. The random sample search mechanism could also be replaced with a more effective method. The possibilities of using network resources to conduct search, and combining results from different games in a central database should also be explored.

The Dynamic Granularity mechanism could also benefit from further work, possibly investigating the integration of army formations into simulations. While some researchers have considered the integration of potential fields and pathfinding, these approaches generally have armies that try to surround an enemy. However, it would be worth investigating how an agent could organise an army into formations before an engagement. For example, an agent could determine the placement of melee units in front of ranged units. Determining the shape and density of army formations is important when facing ranged attacks.

Lastly, further work should be conducted to integrate the techniques in this thesis
into a fully playable Brood War agent capable of taking part in competitions and providing a more in-depth comparison with other current methods.
Bibliography


Avontuur, Tetske, Pieter Spronck, and Menno Van Zaanen (2013). “Player Skill Modeling in Starcraft II”. In: AIIDE.


Botea, Adi (2012). “Fast, Optimal Pathfinding with Compressed Path Databases”. In: SOCS.


Browne, Cameron (2013). “A problem case for UCT”. In: IEEE Transactions on Computational Intelligence and AI in Games 5.1, pp. 69–74.


Chung, Michael, Michael Buro, and Jonathan Schaeffer (2005). “Monte Carlo Planning in RTS Games”. In: IEEE Symposium on Computational Intelligence And Games (Cig).


Guivant, José E, Brett Seton, and Mark Albert Whitty (2012). “2D Path Planning Based on Dijkstra’s Algorithm and Pseudo Priority Queues”. In: Symposium on Combinatorial Search (SOCS).


Harabor, Daniel Damir and Alban Grastien (2011). “Online Graph Pruning for Pathfinding On Grid Maps”. In: Association for the Advancement of Artificial Intelligence (AAAI).


Kao, Kuo-Yuan, I-Chen Wu, Shi-Jim Yen, and Yi-Chang Shan (2013). “Incentive Learning in Monte Carlo Tree Search”. In: IEEE Transactions on Computational Intelligence and AI in Games 5.4, pp. 346–352.


Li, Jiawei and Graham Kendall (2015). “A hyper-heuristic methodology to generate adaptive strategies for games”. In: IEEE Transactions on Computational Intelligence & AI in Games.


Naveed, Munir, Diane Kitchin, Andrew Crampton, Lukáš Chrpa, and Peter Gregory (2012). “A Monte-Carlo path planner for dynamic and partially observable environments”. In: *2012 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, pp. 211–218.


Ng, Peter HF, Simon CK Shiu, and Haibo Wang (2009). “Learning player behaviors in real time strategy games from real data”. In: *International Workshop*
Nguyen, Kien Quang and Ruck Thawonmas (2013). “Monte carlo tree search for collaboration control of ghosts in ms. pac-man”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 5.1, pp. 57–68.


ORTS Homepage. [Computer software]. URL: https://skatgame.net/mburo/orts/.


Powley, Edward J, Daniel Whitehouse, and Peter I Cowling (2012). “Monte carlo tree search with macro-actions and heuristic route planning for the physical


*Spring RTS Engine*. [Computer software]. URL: [https://springrts.com/](https://springrts.com/).


Stanescu, Marius and Michal Čerticky (2016). “Predicting Opponent’s Production in Real-Time Strategy Games With Answer Set Programming”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 8.1, pp. 89–94.


Uriarte, Alberto and Santiago Ontanón (2012). “Kiting in RTS Games Using Influence Maps”. In: Eighth Artificial Intelligence and Interactive Digital Entertainment Conference.
Uriarte, Alberto and Santiago Ontañón (2014). “Game-tree search over high-level game states in RTS games”. In: *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Uriarte, Alberto and Santiago Ontanón (2014). “High-level representations for game-tree search in RTS games”. In: *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*.


Wargus/wargus: Importer and scripts for Warcraft II: Tides of Darkness, the expansion Beyond the Dark Portal, and Aleonas Tales. [Computer software]. URL: https://github.com/WARGUS/WARGUS.


Wehr, Daniel and Jörg Denzinger (2015). “Mining game logs to create a playbook for unit AIs”. In: 2015 IEEE Conference on Computational Intelligence and Games (CIG). IEEE, pp. 391–398.


Yang, Pu and David L Roberts (2013). “Knowledge discovery for characterizing team success or failure in (A)RTS games”. In: 2013 IEEE Conference on Computational Intelligence in Games (CIG), pp. 1–8. DOI: 10.1109/CIG.2013.6633645.


Young, Jay and Nick Hawes (2012). “Evolutionary Learning of Goal Priorities in a Real-Time Strategy Game”. In: AIIDE.
