2013

University of North Carolina Wilmington

Master of Science in

Computer Science and Information Systems

Proceedings

https://csbapp.uncw.edu/mscsis

DEVELOPMENT OF A NOVEL GAME WITH ADAPTIVE LEARNING AGENTS

Rebecca Brown

A Thesis Submitted to the
University of North Carolina Wilmington in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

Department of Computer Science
Department of Information Systems and Operations Management

University of North Carolina Wilmington

2013

Approved by

Advisory Committee

Ron Vetter ______________________

Bryan Reinicke ______________________

Curry Guinn
____
Chair

Accepted By

____
Dean, Graduate School

Table of Contents

Page
Abstract 1

Chapter 1: Introduction 2
1.1 Computer game development 2
1.2 Artificial intelligence in computer games 5
1.3 Boundary 6
1.4 Research Plan 8

Chapter 2: Review of Literature 12
2.1 Computer game development 12
2.2 Artificial intelligence in computer strategy games 15

Chapter 3: Methodology 27
3.1 Development of Boundary 27
3.2 Play testing 37
3.3 Balancing 38
3.4 Basic strategies 39
3.5 Development of adaptive player 43
3.6 Human-computer games 46

Chapter 4: Results 48
4.1 Balancing basic strategies 48
4.2 Human vs. computer experiment 51

Chapter 5: Discussion and Conclusion 60
5.1 Hypotheses 60
5.2 Future work 62
5.3 Conclusion 63

References 64

Appendices
A. Experiment Survey 66
B. Score Differentials by Static Opponent 69

Tables
1 Win Rates (%) for Basic Strategies – First Test 48
2 Win Rates (%) for Basic Strategies – Intermediate Test 49
3 Win Rates (%) for Basic Strategies – Test Used for Adaptive AI 49
4 Win Rates (%) for Basic Strategies – Final Test 50
5 Win Rates (%) for Adaptive Strategy vs. Basic Strategies 51

Figures

1 Boundary screenshot with annotations 7
2 Diagram of client-server structure 28
3 Sequence diagram showing how Clients connect and start games 31
4 Human win rates over the course of the experiment....................52
5 Differential between human and AI scores over the course of the experiment.....53
6 Differential between human and AI scores, broken out by basic AI opponent.....54
7 Human win rates for the different parts of the experiment....................55
8 Comparison of subject perception of static and adaptive AI difficulty 55
9 Subject perception of difficulty predicting AI opponent 57
10 Subject enjoyment of gameplay....................58

Abstract

Development of a Novel Game with Adaptive Learning Agents. Brown, Rebecca, 2013. Master's Thesis, University of North Carolina Wilmington.

This thesis describes the development of a novel web-delivered computer game where human players vie against each other or computer agents that use adaptive learning to modify playing strategies. The game, Boundary, is based on original ideas with features not present in any previous or current game. This novelty presents challenges in game development both in terms of game playability and enjoyment as well as designing intelligent game agents. This thesis describes the game development process involving agile development and player testing. Six simple strategies for playing the game, drawn from different play styles and game objectives, are each implemented with a static AI agent. The adaptive agent classifies its opponent's play during the game by simulating what moves each simple strategy would make and identifying the strategy that produces the closest approximation to the opponent's actions. During development, through computer-computer simulations, the relative strength of each strategy versus the others was determined. Thus, once an opponent's moves are matched to the closest known strategy, the best counter-strategy can be selected by the computer agent. This thesis describes the results of those computer-computer simulations as well as the results of human-computer games.

Chapter 1: Introduction

1.1 Computer game development

Modern games are generally developed by large teams of people in carefully managed projects. A high-profile new title represents a large investment into a very complicated project, and planning out the roles and processes involved helps maximize the developer's chances of creating a successful product.

Game development brings together a wide variety of disciplines to work on different aspects of the project. Designers and artists create the content of the game, programmers implement its mechanics and make the game work, and testers make sure the game works as intended. Within these groups are further specializations - for example, there may be concept artists and animators within the art department, programmers who specialize in game AI or graphics, and testers who focus on compatibility or playability (Novak, 2005). Tying the project together are team managers, who play leadership roles and represent their departments for inter-departmental communication.

As the technology behind games becomes more complex and the market for games grows, the focus of game development is shifting toward higher-level design and implementation. Most games being developed aim to present new content rather than technological innovations, so it makes sense that the code that makes up the foundation of a game is becoming somewhat standardized. Less of a game's code needs to be custom-made because basic components like physics engines are available to license (Rabin, 2010). The shift of importance from the programming team's job to the design team's can be seen in initiatives to generate a game's executable code automatically from a high-level document written in a markup language that does not require much programming expertise to understand (Moreno-Ger et al., 2007) (Cutumisu et al., 2007). Programmers are still needed, but their focus is shifting

away from the actual game code and toward high-level tools for designers and the processing tools needed to generate code from design documents.

Regardless of their specialty, game programmers choose from the same methodologies as other software engineers: waterfall, agile, and iterative development.

Game producers and project managers may find waterfall game development particularly tempting, because it brings very precise forward planning to a high-profile project that can easily spin out of control as development proceeds. With a waterfall approach, the entire project is planned out ahead of time and executed in a linear fashion, completing each step only once and in order. If the project is well-understood ahead of time, the developer can extensively plan out the game, the development schedule, and the budget, then just execute the plan to finish everything. However, this approach is not very suitable for game development, because it does not allow for the unexpected, and the unexpected is particularly common in the fast-moving game industry (Rabin, 2010). Nor does the development itself benefit from a waterfall style: game development typically does not have the rigorous standards and meticulous policies of those industries that lend themselves well to the waterfall method.

Iterative development is more flexible than waterfall, so it is a better candidate for this industry. Basic iterations of the game are planned out ahead of time, with sets of goals planned for certain milestones (Rabin, 2010). Each iteration has new features, and each cycle involves more detailed planning, development, and testing specific to that set of features. This method offers a compromise between the unpredictable nature of game development and the need to get the game developed according to a given schedule.

Agile development is well-suited to several aspects of game development. Under an agile methodology, plans are only made to cover short periods of time, and iterations of the game are rapidly produced on this short cycle. Plans are thus easy to adapt to unexpected events or new

information gained in the course of development (Rabin, 2010). The fast pace of agile development fits easily with the fast and competitive game industry, and playable prototypes can be produced quickly to test out an idea for a change or new feature in the game. However, agile development does not offer any comfort to the game's stakeholders, who generally prefer a more reliable schedule (Rabin, 2010).

Whatever methodology is used to create them, games must be tested extensively before they are finished and released to the public. Different people test the game looking for different problems at different points in the development process.

Programmers test the game code as they write it using unit tests and acceptance tests. Unit tests are low-level tests to be sure small parts of the code work as they should in isolation, while acceptance tests check basic high-level functionality (Rabin, 2010).

Compatibility and playability testing are done by professional testers before the game reaches its target players. Compatibility testing ensures that the game works on all supported platforms and equipment, and playability testers focus more on gameplay, giving reactions to the game experience and suggestions on how to make it better (Novak, 2005).

When the game is ready for its intended consumers, the studio recruits volunteers to participate in beta testing. This generally involves a larger number of people than the previous levels of testing, so a wide range of reactions and feedback can be gathered from the game's target market (Novak, 2005).

1.2 Artificial intelligence in computer games

Artificial agents can perform a variety of functions in games. Competitive games like chess may employ artificial players to serve as adversaries. A computer may act against a player

by controlling enemies inside a game's virtual world. In other games, an AI might be a player's teammate.

Different games may have different goals for the AI agents they incorporate. In a single-player game that requires a computer component, the AI player must perform well at the game in order to provide a challenge. Even in multiplayer games, an AI must play the game well if it is to serve as a good always-available practice opponent. Other games may focus more on the AI as a substitute for a human player, in which case it is more important that the AI's play be human-like than optimal. However, this type of AI player must also be able to play well in order to simulate a human who is good at the game.

In order for an AI player to play a strategy game successfully, it cannot simply pick a strategy and use it over and over. Its human opponent would be able to figure out its patterns and play around them. The AI player must observe its opponent and the game environment and have an intelligent, adaptive response in order to emulate a competitive human player. An irrational or unresponsive AI is frustrating to have on one's team and cannot provide engaging, competitive play. Even if the primary goal for the game is to have a human-like AI rather than a competitive one, this adaptation is still important because it reflects how humans play games. No human player who is invested in a game will play it exactly the same regardless of the game state or the actions of other players, and neither should an AI player that is trying to emulate human behavior.

Despite this, modern commercial game AI typically uses static scripting to plan its moves instead of adapting to its opponents (Spronck et al., 2005). Games have been so focused on improving their graphics technology that there has been little innovation in any of AI's potential game roles (Buro, 2004). Predictable AI scripts are a poor substitute for humans as adversaries or allies, and studios tend to make the AI "play better" by simply allowing it to cheat. To save

development time and money and avoid the risks of innovation, AIs that are meant to be challenging are given extra resources or knowledge of hidden information instead of better reasoning skills (Buro, 2004). This does not add interest or entertainment value, and it is either ineffective and boring or too effective and frustrating - even a very competent opponent is not satisfying to play if it is obviously cheating.

1.3 Boundary

Most commercial strategy games are very complex, with a multitude of factors that a player must consider in order to construct an elaborate course of action. Such complicated behavior is hard to model, but it can be broken down into simpler parts. These simpler strategies can be modeled and understood by an AI, then reassembled into the desired complex behavior.

With this in mind, I set out to develop a simple turn-based strategy game with few factors a player had to process, that would still have multiple viable strategies. An AI player that could play this game intelligently would show that the method was effective, so the same method could be applied in layers to more complicated games.

In Boundary, each player starts the game with a set number of pieces placed at random on a continuous field. Figure 1 shows a typical early game state. These pieces exert an influence on the territory around them, shown as areas of that player's color. Allied pieces that are close enough to each other combine their territories into one "blob" of influence. Pieces can move a distance proportional to the number of pieces in their blob. If a large blob's pieces spread out too much, the influence around them will shrink until they split off from that blob.

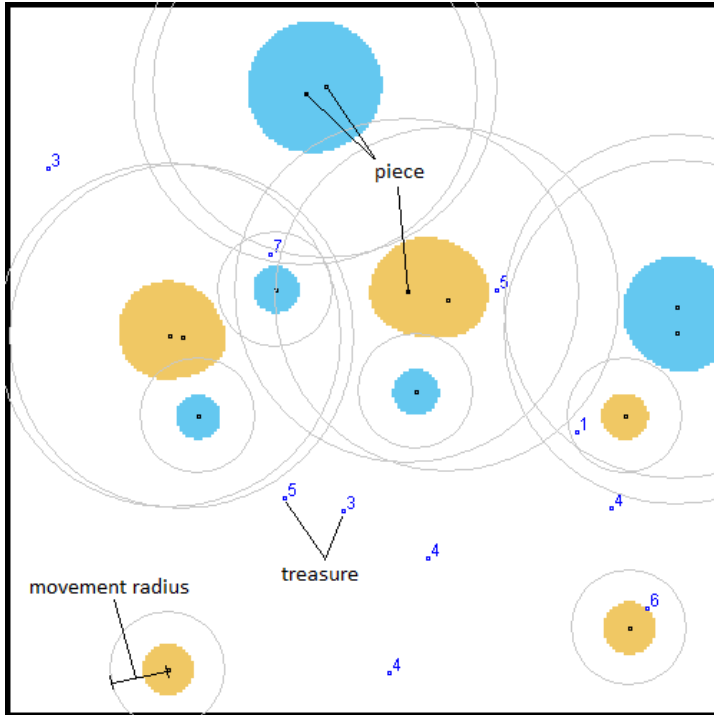


Figure 1. Boundary screenshot with annotations.

The object of the game is to accumulate points either by capturing treasures that are distributed randomly across the field, or by capturing enemy pieces by surrounding them with one's color. Each turn, players have a set amount of time to plan their moves simultaneously. A player cannot see his or her opponent's planned moves, so correctly anticipating an opponent's move is advantageous. At the end of the planning phase, pieces move simultaneously, influence shifts as the pieces move, treasures or pieces may be captured, and pieces may split into separate blobs or merge into new ones. The game ends when either all treasures are captured or one player has no pieces left.

The game's rules were developed with the goal of allowing for various strategies. Players may focus on treasures or capturing enemy pieces (treasures are easier to capture, but enemy blobs are worth more points and put the opponent at a disadvantage), and the blob movement mechanic allows for defensive and aggressive strategies. A defensive player will have the initial

disadvantage of spending turns to group up his or her pieces into larger blobs, with a late-game payoff of increased move distance and capturing potential. An aggressive player spreads out his or her pieces in groups of one or two and tries to end the game early by either taking all the treasures or gaining an insurmountable capture advantage before the defensive player's strategy can pay off.

1.4 Research Plan

Boundary represents a simple stage on which to test AI algorithms for strategic movement of limited forces. Using basic strategies suggested by the game's different win conditions, I developed an artificial player that surpasses these basic methods by adapting to its opponent's play.

1.4.1 Basic Strategies

The most obvious strategy in Boundary is sending each piece after the closest treasure - hereafter known as Greedy Treasure. Greedy Treasure seeks to win in the early game by capturing all the treasures before its pieces are captured, taking advantage of the fact that the game ends when all the treasures are gone.

Another aggressive treasure-focused method is Best Treasure. Instead of using the shortest distance to decide on a piece's treasure target, Best Treasure targets the treasure with the largest value-to-distance ratio. As an aggressive strategy, Best Treasure also looks to win the game early, but it is better equipped to compete against other treasure strategies by getting more value out of its piece movements.

Points can also be scored by capturing enemy pieces, so some basic strategies must focus on capture. Greedy Capture groups its pieces into blobs of two or more, then sends each of those blobs after the closest enemy piece, disregarding treasure. Piece capture is worth more than treasure, so a capture-based strategy like Greedy Capture is a somewhat risky way to pursue

substantial point payoffs. Piece capture also offers the additional advantage of crippling the opponent's board presence, offsetting the inherent risk somewhat.

Boundary rewards the grouping of pieces into larger blobs by allowing more movement across the board and decreased risk of capture, so it follows that some basic strategies would form large blobs. Defensive Treasure groups its pieces into blobs of three or more, then sends these large blobs in pursuit of the closest treasure. As the name implies, this less-aggressive treasure strategy sacrifices speed for more resilience to capture attempts. Because it has one or two large blobs moving around the board, it often scores additional points by accidentally capturing enemy pieces that get in the way of its pursuit of treasure.

A Defensive Treasure strategy suggests a Defensive Capture strategy to complement it. I implemented Defensive Capture, which grouped pieces into blobs of three or more and pursued the closest enemy pieces. It did so poorly in most of the balance testing that I did not use a Defensive Capture basic AI opponent in the eventual human-computer experiments and did not have the adaptive AI select this strategy, but the adaptive AI still needed to be able to recognize it in an opponent's play.

Capture strategies were underrepresented with the demise of Defensive Capture, so I also implemented a Hybrid Capture basic strategy. Hybrid Capture groups its pieces into blobs of two, then each blob pursues the nearest target, whether that is a treasure or an enemy piece. This opportunistic pursuit of whatever is close by may offer an improvement over single-minded capture strategies that ignore treasure, but in other matches it may seem to get "distracted" from higher-value capture opportunities by treasures.

1.4.2 Adaptive AI

The basic strategies are exaggerated versions of different patterns of play that humans may exhibit. In order for the adaptive player to identify these patterns when they are used against

it, it has to recognize similarity to the basic strategies. To do this, it performs each basic strategy using its opponent's piece positions at the start of the turn, acquiring for each strategy a prediction of what the opponent would have done if using it. The closest prediction, measured as distance from each piece in the prediction to its corresponding piece in how the opponent's pieces actually ended up that turn, indicates the closest basic strategy to what the opponent is using.

An exact match of a human opponent's strategy using this method is unlikely, because humans will generally have a more complex plan than the basic strategies. But the important aspects of any strategy make it stronger or weaker to the relevant aspects of other strategies, regardless of how nuanced the complete strategy is. So the adaptive AI can use this information about its opponent's play to select a strategy to use in response. In order for it to make a selection, it has to know which basic strategies are strong against which others. I had the AI play many games against itself using different strategies in order to determine these matchups and balance the game, and the results from those tests provided the basis for this strategy comparison. The adaptive AI selects a basic strategy that will be effective against its guess for its opponent's strategy. Since the opponent modeling is performed every turn, the adaptive player can continue to adjust its strategy as needed when it detects a change in its opponent's play.

To assess the adaptive player in comparison with the static strategies, I ran a trial with human players. Subjects played several games against two AI opponents: the first one used a basic strategy, and the second employed the adaptive AI technique. After each set of games, subjects filled out a survey so I could gather their perceptions of the different AI opponents.

1.4.3 Hypotheses

Hypothesis 1: Human players will learn to win against the adaptive AI more slowly than against the static AI.

Because the AI player is less predictable and better at the game, I hypothesized that it would be more difficult for humans to learn to play effectively against it. I expected the win rate of humans to go up as the games progressed, showing that the human players were learning to play better against their opponents. A slower increase in the human win rate against the adaptive player would show that the subjects learned more slowly against it.

Hypothesis 2: Human players will perceive the adaptive AI to be more challenging to play against than the static AI.

An AI that is objectively better at the game is a reasonable goal, but for any commercial game AI, the player's perception of it is just as important as its actual performance. For this experiment, survey questions asking about the difficulty of the games and the predictability of the different opponents were used to indicate human perception of the challenge levels of the different AI players.

Chapter 2: Review of Literature

2.1 Computer game development

2.1.1 Agile development of games

Coram and Bohner (2005) describe the advantages of agile development and which types of project are most and least suited to an agile approach. The great strength of agile methods is that they adapt very well to change. Plans are lightweight and easy to modify, and the team is not hampered by formal processes that get in the way of actual development. Rapid response to unexpected changes reduces the risks and costs associated with a project's uncertainties. However, for an agile approach to work well, the project has to have the right kind of people, from developers to managers to customers. Communication is critical, and if the team does not have the skill or the chemistry to work efficiently, the project could benefit from a more structured approach. Agile development is also not suitable for safety- or life-critical products, those with well-defined, contractually-obligated requirements, or very long projects.

In general, game development benefits greatly from a methodology that is resilient to change, since game studios compete with each other on the cutting edge of technology for the approval of a fickle public. Games do not have well-defined requirements, they do not have to meet strict safety standards, and they are typically short- to medium-term projects, so it is reasonable to consider an agile approach. The rapid iterations of agile methods also enables the team to prototype its game early and often.

2.1.2 Prototyping

Eladhari and Ollila (2012) define a prototype as "anything that can be interacted with and demonstrate how a system works." They discuss the benefits of prototyping in an iterative game design process. Prototypes let game designers test the target market's acceptance of the concepts in a game before doing all the work to fully implement those concepts. To avoid wasting time on

a game players don’t like or whose mechanics don’t work together, developers must prototype and test the various components of the game as early as possible. This testing is not limited to interactive software prototypes: developers can use paper or physical models to show concepts before they are coded. In order to test the game’s pieces together at different stages of development, an iterative methodology is needed, and designers, testers, and coders must work closely together.

There are two main categories of testing that are enabled by different kinds of prototypes. One is assessing player acceptance of the game, testing how players react to the game’s concepts. The other is testing functionality, finding errors in mechanics or implementation (QA testing) and balancing the game (balance testing).

2.1.3 Playtesting

Developers can write tools that test small parts of the game to see if they work as intended, but testing the game as a whole is generally done by simply having many people play the game over a long testing period, trying different actions and seeing if the game responds as it should. Salge (2008) showed that for strategy games, AI techniques can make large parts of this process more efficient. He simulated beta-testing with AI players that used genetic algorithms to find optimal strategies. The players could be tuned to target different exploits, like loopholes or sequences of actions that would crash the game. This approach was more efficient not only because AI players can play much faster than humans and do not get tired, but also because it removed the human bias toward “natural” or intended game actions.

2.1.4 Balancing

Game designers want games to be not only playable, but also fair to both players. Novak (2005) outlines different ways to achieve this balance: symmetry, in which players are given the

same resources so that neither is favored, and an intransitive ("rock-paper-scissors") relationship between the classes or units that need balancing.

Balance, or lack thereof, is especially noticeable in RTS games, which typically feature different groups ("races") that a player can choose to play, each with different types of units and abilities. If one race is better or worse than the other options, the distribution of players will skew as players pick the most powerful option. So to achieve balance, developers again fall back on human testers that play many games using different options. AI techniques may help with balance testing in the same way that they do QA testing: Fayard (2007) proposes speeding up the balancing process using simulated annealing. AI players simulate good human players by finding the best army a given race can make in a given time without opponent interference, and these armies for the different possible races should be equivalent. He notes that there is more to balance than mere attack strength: designers can create a balance of a variety of different units by altering cost-effectiveness, defensive power, and making some units counter others.

For a game like Boundary, in which the "units" are all the same, superficial balance is achieved by symmetry. Players start the game with the same number of units, and they are randomly distributed, so neither is favored (while one may randomly end up with an advantage in a single game by having pieces closer to treasures, the players have an equal chance of starting with this advantage). However, to encourage different strategies and to enable the adaptive AI to effectively play with different styles, the game rules had to be adjusted so that no strategy dominated all the others. Chapter 3 will discuss this process.

2.2 Artificial intelligence in computer strategy games

2.2.1 Current state of computer game AI

Efforts in AI research into classic board games like chess and Go are well-known, and computer players have surpassed humans in some games like Scrabble (Richards and Amir,

2007). But the artificial players that ship with most commercial games run on static scripts and so cannot provide good competition for advanced players (Spronck et al., 2006). In order to provide more of a challenge, the AI player will often simply cheat instead of reasoning better (Buro, 2004). This is partly because building good AI players for the complex strategy games of today is hard: a typical real-time strategy (RTS) game has a huge space of possible states and actions at any given point in the game, the game state is constantly updated with many interacting objects, visibility of the playing field is limited, and a player must make decisions very quickly. In addition, there is high demand for such games regardless of the AI player's performance, because other people are generally available to play, so developers focus on graphics or other ways of maximizing game realism rather than devoting time and money to AI experiments that might fail (Buro, 2004).

However, this high demand should spur research into better game AI, rather than stifle it. The game industry has already overtaken the film industry in revenue and continues to grow, and as increasing numbers of people play games, the demand for novelty beyond graphics enhancements grows (Bowling et al., 2006). Consumers want more content and variety in their games, and this can include new strides in AI. Even if many players are always available for a popular online RTS, the role of AI is not limited to human-substitution in adversarial games: it can assist the player with complex games by taking over low-level tasks, provide an ally or an intelligent team to command, control the environment in single-player games, or provide a tailored opponent to help a player practice against a certain style. With more innovation in game AI, new uses for it will probably be discovered as well. As the industry grows and markets for even more types of games appear, perhaps AI experimentation will be seen more often as a worthwhile investment rather than an unnecessary risk.

Computer strategy games present challenges that make research into AI interesting regardless of commercial game applications. Since the state and action spaces of these games are so large, an AI player has to use abstractions to reason about strategy at a high level, like humans do - there are too many potential low-level actions to consider each one (Buro, 2004). This high-level strategy must be applied to a variety of tasks in a single game, such as resource allocation, adjustment of plans as they are executed, and optimization problems (Hinrichs and Forbus, 2007). Unlike chess and Go but like poker and Scrabble, most computer strategy games hide some information about the game state from each player (Buro, 2004) (Hinrichs and Forbus, 2007). The AI player (and the human player) must account for the uncertainty of a partially-hidden and dynamic field of play. Computer strategy games also often present a playing space with terrain features that can strongly influence the outcomes of various maneuvers, so players must use more complicated spatial reasoning than they would for a typical board game. RTS games in particular enforce a time limit on a player's decision process: the longer the player takes to make a decision, the more they may fall behind an opponent who reasons faster, so AI players for RTS games must use efficient algorithms in order to keep up with the game. In any strategy game, it is advantageous to anticipate an opponent's moves, so an AI player can benefit from opponent modeling to mitigate uncertainty about the game (Buro, 2004). Depending on the game, an AI player might be allied with a player or other AIs and expected to communicate and cooperate to reach a common goal (Buro and Furtak, 2003).

All of these challenges could have various relevant analogies for non-game AI applications in the real world: for example, robots must learn about a dynamic environment as they go and consider the effect of the terrain around them on their actions. Computer strategy games provide a testbed for research that addresses these challenges, many of which are not

present in the types of games AI research traditionally focuses on, in an easy-to-simulate environment with clear measures of success or failure (Buro, 2004).

2.2.2 Adaptive game AI

Several researchers have successfully created adaptive artificial players of various computer strategy games. These experiments typically define success as consistently defeating AI players running static scripts rather than specifically trying to improve win rates against humans, but an adaptive player that can beat static strategies is still ahead of the current commercial standard. A dynamic system must meet four computational and four functional requirements, according to Spronck et al. (2006), to be usable in a commercial game.

Computationally,

1) It must be fast enough to learn while playing the game.
2) It must not learn behavior that is inferior to manually-scripted AI.
3) It must be robust to the game's randomness.
4) It must learn efficiently from a small number of situations.

Functionally,

1) It should produce results that game developers can easily interpret.
2) It should exhibit an entertaining variety of different strategies.
3) The number of interactions required to learn should be consistent regardless of its opponent's skill or randomness in the game.
4) It should be able to scale in difficulty to match its opponent's skill level.

Spronck et al. (2006) consider the idea of static scripting and turn it into an adaptive strategy by generating the AI player's script on the fly. Dynamic scripting draws a tactic for each stage of the game from a rulebase, with the probability of a tactic's selection based on that tactic's weight. This technique was applied to a simulation of combat in Neverwinter Nights, a

computer role-playing game. The AI learns which tactics are effective by trying them in combat, then adjusts the weights to favor successful tactics. The dynamically-scripted AI was tested against four basic tactics, each implemented with a static script, and three composite tactics, which were combinations of the basic tactics in successive encounters. As the adaptive AI repeatedly engaged an opponent, it learned which actions were effective against that opponent, and after a number of encounters it would start to win more than the static strategy. The researchers defined this number as the turning point, the index of the encounter at which the dynamic player's fitness over the last ten encounters is higher than that of the static player. The turning point varied by opponent, indicating that some of the static strategies were harder to learn to defeat than others. This turning point metric may prove more useful than pure win rates in evaluating an adaptive AI that slowly gets better against a given opponent.

Dynamic scripting can also be modified to scale in difficulty. The most effective means for the dynamic player to limit the effectiveness of its strategies was "top-culling": the system would have a maximum weight, derived from its historical win rate against a given human opponent, and tactics that exceeded this weight were passed over in favor of weaker ones in order to keep the AI's win rate at about 50% (Spronck et al., 2006). This automatic difficulty scaling is a valuable asset for a game AI, because most players want an AI opponent to be challenging without appearing to cheat, but not so challenging that it wins every time (Davis, 1999). A game AI that can automatically scale itself by adjusting the strategies it uses will also save time and effort for game developers who would otherwise have to impose difficulty scaling on the AI outside of its algorithm. This can be done, for example, by limiting the time that an AI is allowed to spend calculating its move (He et al., 2010). Such a technique requires tuning because different algorithms will take different amounts of time to make decisions, and depending on the algorithm, the quality of a solution might not increase monotonically with time.

The dynamic scripting system effectively demonstrated the need for balance between exploitation and exploration. While the researchers wanted the dynamic AI to start applying its learning and playing better as early as possible to be effective, it also needed to learn as much as possible from each encounter to be efficient (Spronck et al., 2006). If the system simply always used the most effective tactics it knew, it would not explore all the actions available to it, some of which might be better. The requirements for a dynamic system all aim to improve the AI's performance, but they can sometimes work against each other.

One of the challenges of creating an effective game AI is giving it the domain knowledge it needs to assess its possible actions, limit that huge action space to actions that are sensible in context, and evaluate its own performance (Spronck et al., 2007) (Weber and Mateas, 2009). Requiring that this knowledge be coded manually by a game designer or expert is less than ideal, both because it takes time that could be spent on other areas of game development and because it limits the AI's knowledge to that person's knowledge (He et al., 2010). It would be better if the AI could expand its domain knowledge on its own, exploring the space of possible strategies beyond what a human might consider.

Spronck et al. (2007) improved their dynamic-scripting agent, this time in the RTS domain, by having it generate domain knowledge using a genetic algorithm. They also used abstractions to make the large action space more manageable.

A player in an RTS can construct various types of buildings. Many types of buildings cannot be built unless the player already controls buildings of certain other types. Using these building dependencies, the researchers defined a set of possible states that players might play through in a game, with each possible combination of building types representing a state. This is a useful abstraction because it not only gives structure to the game, but also removes impossible

game states from the AI's consideration. The AI can then represent the game as a succession of stages, during which actions may be performed. Actions were also highly abstracted: rather than thinking about specific orders for individual units, the AI planned high-level actions like "attack" and "construct building X." This level of domain knowledge can easily be supplied to an AI without great risk of mistakes or omissions.

This structure, representing a game as a succession of states that contained actions, lent itself to the use of genetic algorithms for learning entire game plans. The chromosome, or game plan, was divided into states, which included a series of genes, or game actions. The fitness function involved centered on a strategy's relative military success, since this is a good factor of performance in an RTS game. The researchers used relatively high rates of crossover and mutation to encourage exploration of the strategy space.

The strategies evolved through this process were used to build bases of effective actions at the different possible stages of the game. Dynamic scripting was then applied for the actual gameplay, using the automatically generated knowledge bases. The combined techniques were compared to dynamic scripting with manually-generated tactics against static strategies, and the genetic algorithm was more successful at finding effective counter-strategies.

Aha et al. (2005) sought to improve on this technique so that it would be effective against randomly-selected static opponents, instead of learning against the same one in successive games. They did so by adding another source of automatically-acquired domain knowledge: a library of cases that contain a game situation, a tactic to use in that situation, and how well the tactic has performed when used in the past. This case-based plan selection refines the AI's awareness of the game situation, because the cases contain not only the building-based state of the game, as in the Spronck team's work, but an additional set of features, including kill counts

and the number of worker units and combat units, at a given point in the game. Thus the AI can select with finer granularity an appropriate tactic for the situation.

2.2.3 Opponent modeling

Case-based systems are also used for plan recognition in opponent modeling, another useful tool for adaptive game AI. Inferring an opponent's plan allows the adaptive AI to predict its opponent's moves and adapt accordingly.

Fagan and Cunningham (2003) applied case-based plan recognition to Space Invaders, a fixed shooter game. They defined a simple set of states the player could be in: safe behind a bunker, unsafe but not under fire, and very unsafe, or in the open and under fire from the space invaders. The action set was also simple - the player could only fire, hide, emerge from cover, dodge enemy fire, or suicide. The system sought to predict a player's action, given a short string of observed actions. Cases, or subplans, were strings of state-action pairs four steps long. When three steps were seen that matched a case, the fourth step in that subplan was predicted as the player's next action. The researchers found that too large a plan library could adversely affect prediction. Because subplans were only four states long and the state and action spaces were not large, a larger library meant a much higher probability that there would be multiple subplans with the same three first steps, so the system would have no way of knowing which fourth step to predict. However, with an appropriate library size, this simple system produced fairly good results for a simple game.

Cheng and Thawonmas (2004) investigated the use of case-based plan recognition in an RTS game, with the aim of assisting the human player with management tasks rather than competing in the game. They suggest different levels of cases with different situation

information: a strategic case would include features like map positions and units in the vicinity, a tactical case would include units and resources available, and an operational case would be on the level of an individual unit, with its position and commands received. The researchers note that not only is the large state space problematic for case libraries, but the pieces in play can also change, so actions need to be mapped between different units as well as different environments.

Weber and Mateas (2009) claim that case-based plan recognition is unlikely to scale to the RTS domain, simply due to the huge increase in complexity. Their approach to domain knowledge and opponent modeling in StarCraft, a popular RTS, focused on learning game strategies by training on the large collections of StarCraft game logs available online. This is an intuitive approach for an established game because not only are many logs of professional play readily available, this type of study is a common way for humans to learn to get better at RTS games.

The replays gathered were converted into logs of game actions and the times at which they occurred. Logs were labeled with high-level strategies based on rules drawn from analysis of play. Each player's build order -- the sequence and timings of his or her production of buildings, units, and upgrades -- was encoded as a feature vector.

Strategy prediction was then treated as a multi-class classification problem, trying to match a log's feature vector against a set of known strategies. Various machine learning algorithms were tested against a classifier that used the same rules that were used to label the logs with strategies. Fairly early in the game, before the eight-minute mark, the machine learning algorithms performed better than the rule-based classifier. This is useful not only because it implies some level of prediction before the opponent has already carried out the strategy, but

because the early minutes of an RTS are a crucial time in deciding what strategy the opponent is using and how best to adapt.

This was not a very realistic application to actual gameplay, though, because RTS games hide information from each player. So Weber and Mateas introduced imperfect information by adding noise to features, which simulated a delayed realization that an opponent performed a given action, and by removing some features entirely, which simulated an inability to see a given action at any point. All of the machine learning algorithms degraded in precision under these conditions, but they were more resilient to imperfect information than the non-learning classifier.

This method proved effective at opponent modeling in a game with an established collection of logs available. However, it would obviously be of limited use in a game that was still being made or had not yet developed a high-level scene.

Schadd et al. (2007) used hierarchically structured models to represent opponent strategy in an RTS, classifying first on a general style and then on specific unit production. This division of the problem helped address the time limit inherent in RTS opponent modeling: classification of the opponent's play must be made in real-time, competing for system resources with the other aspects of the game, as quickly as possible in order to get the most advantage from it. The system used a fuzzy model to distinguish between an aggressive and a defensive strategy, based on the relative amount of time the opponent spent attacking. Since the opponent's actions could not be directly observed, an attack was defined as when the system's own units were lost.

The second classification depended on the first. If the opponent was aggressive, the second classification used observations gathered during attacks to determine the most prevalent type of unit the opponent was producing. For a defensive opponent, observations were gathered

during scouting trips to the opponent's base, and the classification was based on the type of buildings the opponent was producing.

While this study did not address adapting game strategy to the information gained through opponent modeling, it suggests that adaptation of the classification process itself based on preliminary results will be an effective way to reduce the computational costs of opponent modeling in a time- and resource-sensitive environment like an RTS game.

Laviers et al. (2009) used support vector machines to perform opponent modeling in a football simulation game. If the opponent's defensive play could be identified early enough, the offensive play could be changed to one that was historically more effective against that defense. The opponent classifier was trained on every combination of known offensive and defensive plays and starting configurations, and it achieved near-perfect classification at three timesteps into a play.

Candidate offensive plays were also analyzed to determine their similarity to each other, taking into account positions of players, angles of movement, and path distances. Once the defensive play was classified, the offensive strategy could only be changed to a more advantageous one if the candidate play was sufficiently similar to the one already in progress. This check was necessary to avoid harming the team's own strategy in the process of switching to a better one. The researchers also found that switching the tactics of only a subgroup of players improved performance, suggesting that a finer level of adjustment may benefit an adaptive game AI more than dramatic strategy changes.

Opponent modeling is also a major challenge in poker, because knowing the types of mistakes an opponent will make is essential to exploiting those mistakes (Davidson et al., 2000).

Baker and Cowling (2007) created an adaptive player that performs Bayesian opponent modeling in a simplified version of poker. Their distillation of poker used a ten-card deck, with four players each being dealt one card, and the winner being the player with the highest card.

They defined four distinctive styles on two axes of play: loose versus tight and aggressive versus passive. Four AI implementations of these styles were implemented with a simple deterministic design, using two probabilities to make decisions: a minimum win probability for remaining in a hand (checking if there is no additional risk, folding otherwise), and a minimum win probability for betting.

For each basic style, an "anti-player" was developed to beat that style, using probabilities derived from playing games against three players using the target style. The adaptive player, in a game with unknown opponents, used Bayes' theorem, along with a history of game records showing each basic style, to calculate the probability of each player using each basic style. With each game action, it recalculated the probabilities until it reached a confidence threshold and classified each opponent with a basic style. Then it prioritized its opponents by type: for example, it was most important to take out tight passive players first, because they required a very specific counter-strategy. Once the adaptive player had a target opponent to defeat, it switched to the anti-player for that style. If it defeated that player, it would change strategies to target the next opponent on the list, and so on.

The opponent modeling component of this study was isolated by testing the adaptive player against a player that was told the style of its opponents each turn, simulated games with those styles, and determined the best probabilities to use. The adaptive player that performed opponent modeling performed comparably to, and sometimes better than, the simulation-based player. Both achieved very high win rates, better against some sets of opponents than others.

Obviously, this study was limited to classifying and predicting static, deterministic, non-bluffing opponents in a simplified game, a task much easier than playing actual poker, but its implications for opponent modeling were promising for poker and simpler games. This study's methodology was so similar to the plan for this thesis that it will no doubt serve as a useful guide for this research.

Chapter 3: Methodology

3.1 Development of Boundary

3.1.1 Agile Programming

After considering the various coding methodologies game developers employ, I decided on an agile approach for Boundary. Rapid iterations of the game early in the project would help clarify the game concepts in relation to the thesis topic. Agile development would enable me to test new features and rules regularly and decide whether they should be a part of the game. On a tight schedule, having a new playable prototype at every meeting with my thesis committee chairman helped keep us on track as the game grew. As the game moved into testing, I would be prepared to make changes to the game's rules and quickly produce new builds for testers to try out.

3.1.2 Platform

The game client is distributed as a Java applet that communicates over sockets with a Java server program hosted on a virtual Linux server. The original proof of concept for the game was a Java applet with moveable pieces in a single blob, and I built features onto that to develop the full game client. In hindsight, it would probably have been better to write a JavaScript client that used WebSockets to communicate with a servlet, but despite the agile approach, this did not become apparent until it was infeasible to rewrite the game for this project.

3.1.3 Client-Server Structure

The structure of the game's client side, illustrated in Figure 2, is fairly simple. A Client object gathers user input and performs all the necessary calculations to progress the game. It has helper objects to show the game lobby and to generate the incremental stages between the given state and a planned state. Another helper object, the ClientRep, listens continuously on the client-

server socket, passes message objects it receives from the Game to the Client, and sends any necessary message objects from the Client to the Game.

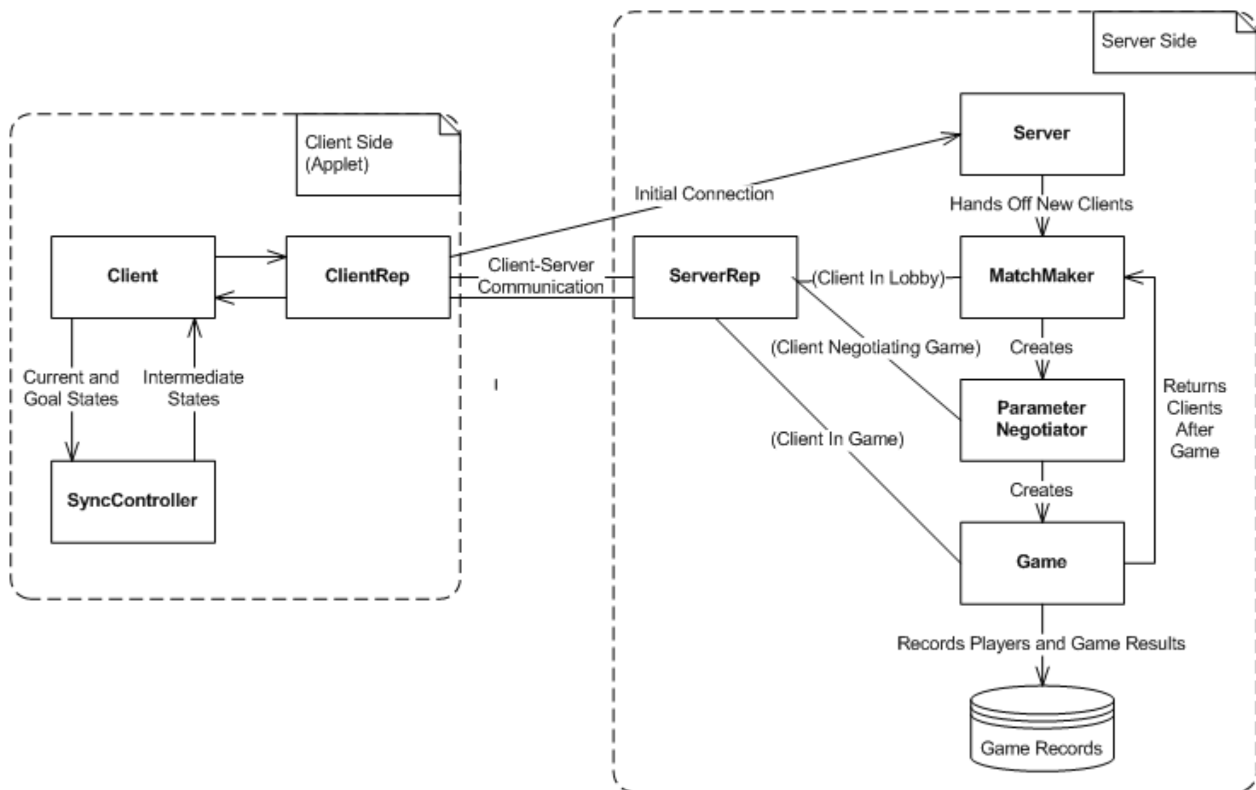


Figure 2. Diagram of client-server structure. Only one of ServerRep's connections to a server-side entity exists at a time, depending on the state of the client.

On the server side, a Server object listens on a welcome socket to catch new Clients connecting to the game. When the Server gets a new connection, it creates a new socket for that Client and passes the socket to a MatchMaker object.

The MatchMaker maintains a list of Clients that are waiting for games. This list is shown to a user in the form of the game lobby, from which a user can send a game request to another waiting player or request a game with an AI player. When the MatchMaker first receives a new Client connection from the Server, it creates a ServerRep for that socket, which is the server-side complement to the ClientRep object - it listens on the socket for whatever server-side object

needs to communicate with the Client on the other end and performs any necessary message passing.

When a Client requests a game, the MatchMaker handles the request. If the user is requesting an AI game, the MatchMaker creates a ClientAI object, which then connects to the Server like any other client. The ClientAI is created with a record of which Client requested it, so it is immediately matched with that Client when it is passed from the Server to the MatchMaker. If the user is requesting a game with another player, that Client gets sent a new game request which, if accepted, initiates the game.

Once two clients (two Client objects or a Client and a ClientAI) are matched, they are sent to the ParamNegotiator. Moving Clients between server objects consists mainly of setting which server object their ServerRep communicates with. The point of the ParamNegotiator was to allow testers to set different parameters for the game, such as number of pieces per team, blob size, and how far pieces can move. To control the experiments, these parameters are locked, and the ParamNegotiator is short-circuited by having both clients immediately accept the default parameters without showing the negotiation screen to the user. When enabled, the ParamNegotiator keeps track of the current parameter values that are being shown to each player, detects changes made by either one, and shows those changes. Both players must agree to the same set of parameters before the game can begin.

When players have agreed upon the parameters for the game, the ParamNegotiator creates a Game object with those parameters and passes the clients off to it. This object controls the game itself. It keeps the clients synchronized through the turns, collects each player's planned moves and sends out the full planned state for each turn, allows players to communicate through chat, and determines when to end the game. When the game is over, the Game records

the results in a database, then passes all Clients back to the MatchMaker if they wish to play another game.

The typical sequence for a Client requesting and playing games is shown in Figure 3. Messages are labeled above their arrows, and unlabeled arrows represent forwarding received messages.

3.1.4 Client-Server Communication

Clients and server objects communicate by sending Message objects over object streams. A Message has one of several types, covering player information, game requests, reports of captures, indications to move to the next phase, board states, chat messages, errors, and anything else a client or server object might need to send. The most common and complex Message is a BoardState, a serialized representation of the current state of each blob and treasure on the board. A BoardState contains Team objects, which are groups of Blobs, which are groups of Points, which are either pieces or treasures, so it contains anything that is displayed on the game board. Every turn, clients receive BoardStates from the Game, alter them to indicate where they plan to move their pieces, and send them back to the Game. When it has received all the plans, the Game puts them together into one BoardState that represents everyone's plans for the turn and sends it out. It is then the Client's responsibility to determine what happens when these plans are put into action. The Client shows the turn playing out to the user, and when it has done all the moves called for in the plans, it sends back the resulting BoardState, with appropriate blob groupings and with any captured pieces removed.

The ClientAI does not perform these calculations, because it doesn't have a player to watch them unfold. There would be no point in calculating all the intermediate states in a turn on both the client and server side, so the ClientAI relies on the end-of-turn BoardState from the Client to know what happened in a turn.

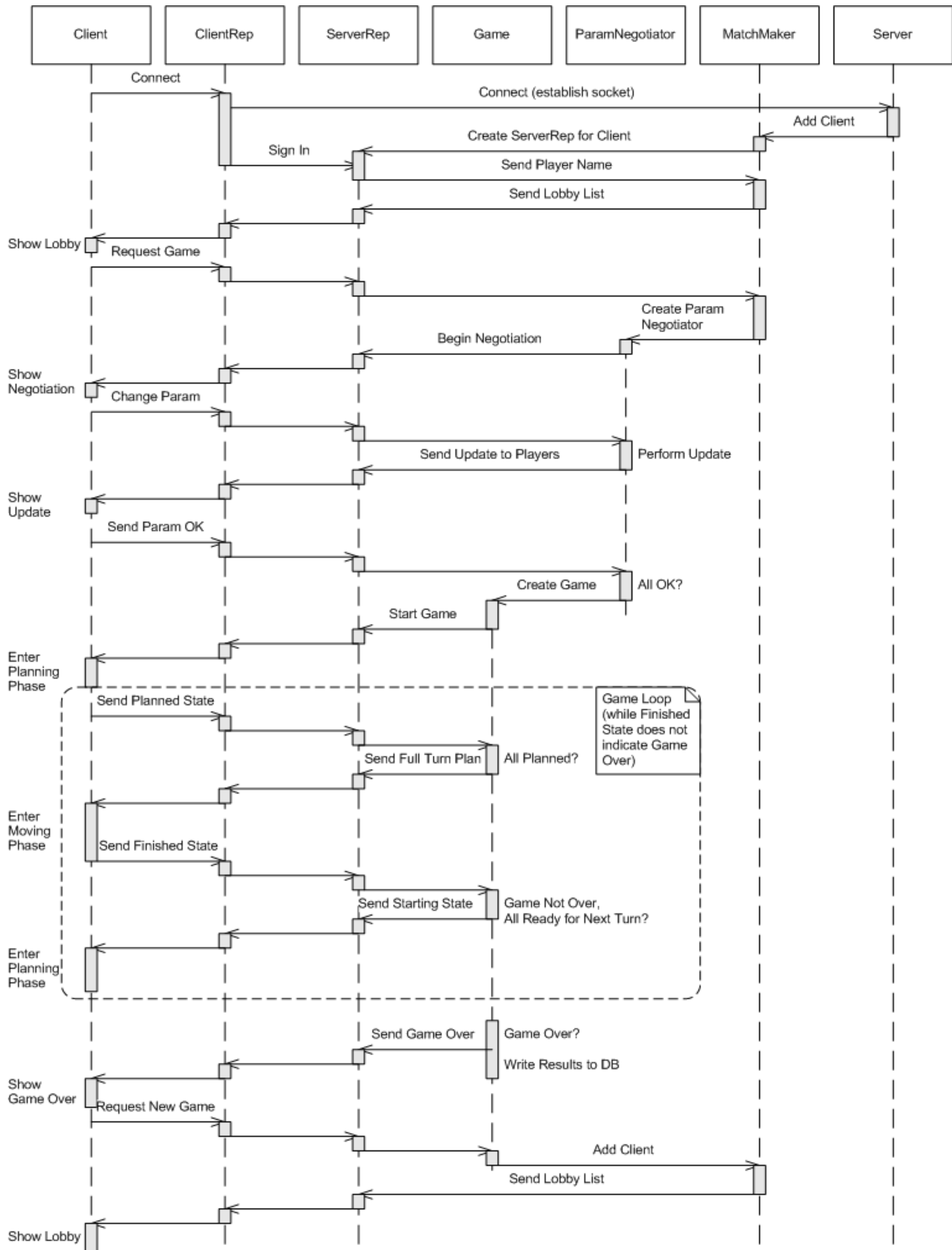


Figure 3. Sequence diagram showing how Clients connect and start games.

3.1.5 Game Calculations

The Client is responsible for determining the game state at every point in the game. This is the Client's responsibility and not the Game's because the Client has to show shifting influence and changing blobs as the user drags pieces during the planning phase, before any plans are finalized or sent to the server. Asking the Game to calculate influence every time the board state changes would require a lot of unnecessary network traffic, likely lag the game display, and drastically increase the load on the server, so it makes sense to have the Client take care of it. The Game and any ClientAIs trust the Client to make these calculations.

3.1.5.1 Influence

Every time the board state changes, the influence pieces have on the pixels around them has to be recalculated. This is done pixel by pixel. To improve performance, only one quarter of the actual pixels on the game board go through this calculation - the upper left pixel of each square of four is used to determine the color of that square. To cut down further on the number of pixels processed, only those pixels that are within a certain radius of a piece (proportional to the number of pieces in that piece's blob) are used, and it is assumed that any other pixels are outside any blob's influence.

First, the distance from the pixel to each blob is found. The distance from a pixel to a blob is equal to the geometric mean of the distances from each of that blob's pieces to the pixel. For each team, the closest blob and its distance are stored.

Next, the distance is modified to account for the other team's influence. The formula is

$$dc = d_1 + \frac{{d_1}^2}{d_2}$$

where dc1 is the modified distance, d_1 is the closest blob distance for that team, and d_2 is the closest distance for the other team. The modified distance allows enemy blobs to "push" on each other's influenced territory and gives a more natural, blob-like look.

A maximum radius is also stored for each team: this is equal to the radius of that team's closest blob to the pixel, which is proportional to the number of pieces in the blob.

The modified distance and the radius can then be used to determine which, if any, team is influencing that pixel. If neither distance is within its team's maximum radius, the pixel (more precisely, the pixel and the three other pixels in its group of four) will be uninfluenced and colored white. If one team's distance is within its radius and the other is not, the pixel will be influenced by the former team and colored in that team's color. If both distances are within their radii, the smaller distance wins the pixel's influence.

3.1.5.2 Capture

After influence is calculated, all treasures and pieces on the board must be checked to see whether they have been captured by an enemy. Pieces are checked first.

To check a piece for capture, the Client starts at the piece's location on the board and moves out in eight directions, one pixel at a time, for 30 pixels. A surround counter starts at 0. If three pixels in a row for a given direction are the opponent's color, the surround counter is incremented. If the piece's own team's color is seen at the end of the line, the surround counter is decremented, in an effort to give pieces credit for being in a large blob. The surround counter is also incremented if a wall is seen. Walls can count toward the surround counter up to three times. Finally, if the surround count is at least 6, the piece is considered captured.

Treasures are checked the same way, but they only check one pixel in each direction. If that pixel is the enemy color, it counts toward the surround count. Therefore, a piece need only overlap a treasure with its influence in order to capture that treasure.

Pieces and treasures may only be captured in the move phase, because the board state will be different when both teams' moves are considered.

3.1.5.3 Merging and Splitting

After a piece is checked for capture, its distance from the rest of its own blob and from its allied blobs is checked, to see if it needs to be split off from its current blob or merged in with another blob.

A merge distance is set based on the size of the nearest allied blob. Larger blobs have a larger merge distance and will therefore subsume allied blobs more readily. A split distance is also determined based on the size of the piece's own blob.

To end merging and splitting loops, a count is kept each turn of how many times a piece has been involved in a merge or split. If this count is over 20, the piece will not initiate a split or merge.

If the nearest other allied piece is in the same blob as the piece being considered and the distance to that piece is less than the split distance, a split will occur. The piece will be added to a new blob, which is added to the team, and its old identity in its old blob will be deleted.

If the nearest other allied piece is in a different blob and the distance to that piece is less than the merge distance, a merge will occur. The piece's blob and the nearest allied blob (the one with the nearest allied piece in it) will have all their pieces put into a new blob, and the old blobs are deleted.

3.1.5.4 Incrementing State

In order to move from the current state to a planned state, intermediate states must be calculated. This allows animation of the states for the user, in addition to capture in passing. The object that takes care of this is the ClientSyncController, which reports back to a Client.

In the Client's paint method, it calls a method, level(), that does all the calculations described above. At the start of this method is a call to newState(), which retrieves the next intermediate state from the ClientSyncController. After it gets this new state, newState calls repaint(), which ends up calling level() again. In this way, the Client builds up a call stack layered with BoardState retrieval, influence and capture calculation, and displaying the board, and as this stack unrolls the user sees the animated incremental states.

Whenever a new planned state is received from the Game, the Client hands it to the ClientSyncController, which needs to know the goal state in order to generate intermediate states. The Client can then ask the ClientSyncController for the next board state, passing it the current state. The ClientSyncController will then compare this state to its planned state, and if it can make them closer to equal by moving pieces, it will begin generating the next state.

To increment a state, the ClientSyncController first shuffles all the blobs from both teams together into a random order, so neither player has a consistent advantage of moving first. Then for each piece, it gets the planned coordinates from the planned state and compares them to the current coordinates of that piece. It calculates the slope of the line from the old location to the new, then finds new X and Y coordinates that lie on that line one pixel closer to the goal point (here, (x,y) represents the integer coordinates of the current point, and distance is set to 1 in order to get a small step):

$$newX = \begin{Bmatrix} x + \dfrac{distance}{\sqrt{1 + slope^2}} & plannedX > x \\ x - \dfrac{distance}{\sqrt{1 + slope^2}} & otherwise \end{Bmatrix}$$

$$newY = slope(newX - x) + y$$

Since this may yield non-integer values for newX and newY, and the new coordinates have to be integers, the difference must be accrued or movement will not be in a straight line. A

Point object stores an accruedX and accruedY value. NewX and newY are rounded, and any spillover is added to the appropriate accrued value for that Point. Then if the absolute value of the point's accruedX or accruedY value is at least 1, the integer part of that accrued value is added to the point's new rounded coordinate and subtracted from the accrual variable. Finally, the new coordinates are checked to be sure they are within the bounds of the board and are not intersecting a wall, and the piece's old coordinates are replaced with the new.

3.1.6 Logging

Because everything about a game can be represented as or derived from a series of BoardStates, and a BoardState is serializable, the Game object can easily record complete logs of games. It does so in two parts: a text file contains the names of the players, the parameters used, and any chat or system messages that were sent, and a .blobreplay file contains a BoardState representing the combined plans for each turn. The game can be watched one turn at a time using a ReplayViewer, which reads BoardStates from the replay file and performs the same calculations that a Client does. The name of the replay file is stored in the database record that holds the results of the game.

To avoid trawling through text and replay files to get game statistics, I created a simple MySQL database on the server where information on Players and Games could be stored. The Player table stores an ID, the player's username (with spaces removed), number of wins, number of losses, whether it is an AI player, and an ID for the experiment the player was associated with. The numbers of wins and losses are automatically updated with a trigger whenever a Game is recorded, as long as it is not a tie, and are simply there to provide a faster way to see a player's win rate. Players are inserted, if their usernames do not exist in the Player table, at the start of a game. At the end of a game, a record is inserted into the Game table, which contains the winner's ID, the loser's ID, the name of the replay file, a timestamp on creation, whether the game was a

tie (if it was, the winner and loser fields simply store the game's players in order by team), and the game parameters.

3.2 Play testing

A small group of people agreed to serve as alpha testers for the game. The client for the alpha testing phase allowed players to set various parameters before starting the game, such as the number of pieces per team. Alpha testers were asked to play as much as they liked, try out different parameter options, and give free-form feedback if they found bugs, found a preference for any particular settings, or had any comments about the game or its strategies. The goals behind this testing were threefold: to find bugs (or player confusion about the rules that would be mistaken for bugs), to find out what players did or did not like about the game in general or about specific parameters, and to gather data about what strategies players used so that these strategies could be kept in balance.

In practice, this alpha testing was not very effective. Players did not play very many games and often did not vary the parameters for the games they did play. A few bugs were identified, and some feedback was gathered about how players expected the game to work versus how it actually worked. The shrinking of influence when pieces spread apart and the ability of smaller blobs to capture larger blobs did not match intuitive expectations of the mechanic, resulting in player frustration. However, changing this mechanic would have involved fundamentally changing the way influence was calculated, resulting in a radically different game. I did change the distances at which pieces merged into and split from blobs to address some of these complaints. Perhaps with more alpha testers, better guidance or structure in the testing, or more time for the players to play their games, this would have been more effective. As it turned out, I could not pursue the alpha testing as much as I would have liked because I needed to move on with the rest of the experiments.

3.3 Balancing

To move forward with the experiments and the development of the adaptive player, I needed to establish a balanced set of simple AI strategies: each had to be favored and unfavored against at least one other strategy, preferably achieving an intransitive relationship as described by Novak (2005). Instead of trying to piece together which strategies worked and why from human alpha testing games, I sped up the process with artificial players, much like Fayard (2007) and Salge (2008). I wrote six simple strategies, had the computer play against itself using each combination of strategies without any consideration of how its opponent was playing, and gathered the data on the various matchups. If a strategy was universally favored or unfavored, I adjusted the game parameters to strengthen that strategy or weaken others and ran the test again.

This process simultaneously produced three necessary components of future experiments: a set of distinct basic strategies, game rules that would encourage a balance between the strategies, and win rates that the adaptive AI could use to determine the best strategy to use against a given opponent.

Testing the basic strategies against each other led to some surprising changes to the game rules. Initially, each player started with ten pieces, piece capture was worth 20 points, and there were 10 treasures that ranged in value from 1 to 15 points. This seemed like it would be a reasonable balance between the treasure-based and capture-based camps. However, treasure-based strategies were far too powerful with these settings. With so few treasures, they could end the game very quickly, and with some treasures being worth as much as a capture, there was not enough incentive to capture. What ended up balancing the strategies was reducing the number of pieces to 7 per person, raising the value of piece capture to 30, and having 15 treasures on the board that ranged in value from 1 to 7 points. This resulted in a longer game that gave capture strategies a chance at acquiring points. The reduction in treasure value seems drastic, but it did

not hurt treasure strategies too much. A treasure strategy will still try to end the game early on treasures without losing pieces to capture, and if that happens the treasure strategy will still win, just with a lower score than when treasures were worth more.

While I was unable to achieve a perfect balance between basic strategies by the time of the human-computer experiment, repeated tests and adjustments did result in an improvement in the balance of the game. In the interest of time, I had to end the balancing adjustments at some point, but the process is sound. Even with some imbalance remaining, balance testing and analysis of the effects of different adjustments gives insight into the strengths and weaknesses of the different strategies. I took these observations into account, in addition to the actual tested win rates, when choosing the adaptive player's reaction to the different basic strategies.

3.4 Basic Strategies

3.4.1 Greedy Treasure

The simplest and most obvious basic strategy is Greedy Treasure. Each piece seeks out the closest treasure to it and pursues that treasure until it is gone. Greedy Treasure is an aggressive strategy that seeks to win by taking all the treasures in the early game before its opponent can start capturing its pieces.

Greedy Treasure uses persistent targeting. Piece IDs and the IDs of the treasures they are targeting are stored in a hash map. When a piece has no target, at the start of the game or because the treasure was captured, the piece will choose a new target. Both the nearest treasure that is already being targeted by some piece and the nearest untargeted treasure are considered, with the untargeted one being preferred. So Greedy Treasure will target all treasures on the board, nearest first, before targeting one twice. The reason for this persistent targeting is to avoid wasting movement sending multiple pieces after a treasure when only one is needed to capture it.

Greedy Treasure's only strength is its speed, so it really only does well against very slow strategies like Defensive Treasure. Aggressive capture strategies and smarter treasure targeting are favored against it.

When a piece in either Greedy Treasure or Best Treasure plans to move toward a target, it does so with a calculation much like the one used by ClientSyncController in incrementing a state, the difference being that the distance used is equal to the piece's movement radius.

3.4.2 Best Treasure

Since treasures have different point values and these values are visible to both players, it makes sense to have a strategy that takes those values into account in targeting. Best Treasure uses the same persistent treasure targeting algorithm as Greedy Treasure, except that instead of trying to minimize distance, it tries to maximize the ratio of piece value to distance.

Best Treasure is favored against Greedy Treasure because their strategies are so similar, but its moves are generally worth more points. In testing, Best Treasure often fell victim to capture-focused strategies, somewhat more so than Greedy Treasure.

3.4.3 Greedy Capture

Greedy Capture seeks to exploit the high value of enemy piece capture compared to treasure. It groups its pieces into blobs of two and sends each blob after the closest enemy piece. If a blob has only one piece in it, it will target the nearest allied piece, preferring pieces that are also alone in their blobs; otherwise, it will target the nearest enemy piece.

Greedy Capture uses persistent targeting; however, a blob will recalculate its target when either that target is gone or another piece joins the blob and has a different target. All pieces in the blob must agree on a non-null target, or the blob will re-target.

When any capture strategy has a target, it will try to position its points at a good angle to capture that target. When attempting a capture, it is generally better to spread pieces out slightly

along a line perpendicular to the angle of approach, rather than send pieces in single file. So to send pieces in for a capture, the capture strategies will send one piece slightly to one side of the target point (in relation to the line of approach) and one slightly to the other. If there are three pieces in the attacking blob, the third will approach the target directly. If there are more than three, the pieces will be split roughly in half, alternating which side they target. Non-capture strategies do not know how to do this, because they only ever intentionally target treasure or themselves.

Greedy Capture is slightly less aggressive than the treasure strategies above, because it has to take a few turns at the start of the game to group its pieces before going hunting. It makes up for this later start with more power in the mid- to late-game, but it can be outpaced by a fast start from an aggressive strategy. Being a capture strategy, it is riskier than the treasure varieties (choosing to approach the enemy directly may result in either capturing or being captured), and tends to do poorly against strategies that form large blobs like Defensive Treasure.

3.4.4 Hybrid Capture

Hybrid Capture is the only one of the basic strategies that may target either treasures or enemy pieces. When trying to find a target for one of its blobs, it has two sets of potential targets, a priority set and a secondary set. For a blob with only one piece in it, the priority target set is allied blobs (preferring those with only one piece), and the secondary is treasures. Larger blobs target enemy pieces first, with the secondary goal being treasures as well. The end result is a capture strategy that can basically be momentarily distracted by near enough treasures.

The actual targeting involves finding the best target for the primary target set, then the best target for the secondary set. If the distance to the primary target is greater than the distance to the secondary target, the secondary target is chosen. Otherwise, the primary target is chosen. When trying to find the best target from among allied pieces, Hybrid Capture does the same

calculation as Greedy Capture, preferring isolated pieces. When trying to find the best enemy or treasure target, it does the same distance-based check as Greedy Capture or Greedy Treasure.

Hybrid Capture tends to perform best against the aggressive treasure-based strategies. Its offhand interest in treasure weakens it against Greedy Capture's single-minded capture focus. Since it is so similar to Greedy Capture, I expected it to perform poorly against Defensive Treasure, but it did surprisingly well, perhaps because its pursuit of treasure ends the game earlier and Defensive Treasure is strongest in the late game.

3.4.5 Defensive Treasure

Defensive Treasure gathers its pieces into one or two large blobs to avoid capture, then sends those large blobs around the board after treasures. A blob must have at least three pieces in it before it can start targeting treasures, and the treasure targeting is the same as Greedy Treasure. This is clearly the slowest treasure strategy, and that works against it in the Greedy Treasure and Best Treasure matchups, but it is powerful in the late game because its one blob has a large movement radius and will generally capture anything in its wake in its pursuit of the last treasures on the board. As the name suggests, it is most effective against capture-based strategies, so the adaptive AI uses it against capture strategies.

3.4.6 Defensive Capture

Defensive Capture was intended to round out the space of strategy archetypes suggested by the game's rules. It groups its pieces into blobs of three or more like Defensive Treasure, then those blobs go after enemy pieces like Greedy Capture. Through most of testing, this strategy was terrible against every other basic strategy, so it is never chosen by the adaptive player. Last-minute changes to the basic strategy algorithms resulted in a much more competitive Defensive Capture, and the adaptive AI should have reconsidered it, particularly as an alternative to

Defensive Treasure against Hybrid Capture. Regardless, it still needs to be included in testing because the adaptive AI has to be able to recognize it when used by an opponent.

3.5 Development of adaptive player

The adaptive AI player has three advantages over the simple AI players: it changes its strategy mid-game (and is thus less predictable), it chooses strategies it thinks will do well against its opponent, and it predicts its opponent's moves with more accuracy and reacts appropriately. To enable this behavior, it needs both an idea of which strategies perform well against which others and a way to characterize its opponent's strategy.

Strategy selection is fairly simple, given the results of the computer versus computer testing. The adaptive player looks up the opponent's simple strategy and chooses the best simple strategy to use against it, following a similar approach to Baker and Cowling (2007). The chosen simple strategy forms the basis of the adaptive AI's behavior, but if it is using a capture strategy, it augments the basic plan with predictions as to what its opponent will do.

Initially, I planned to match an unknown strategy to the closest known one by developing a fingerprint for each simple strategy using a set of metrics gathered from computer play. This fingerprinting approach has been used in related work in opponent modeling, using relative frequency of actions in poker to establish probabilities that a given strategy will perform those actions (Baker and Cowling, 2007). The metrics I used were average number of treasures captured per turn, average points gained through treasure capture per turn, average number of captures per turn, average blob size, average change in distance both between all allied points and between allied points and enemy points, and average number of pieces that got captured per turn. The adaptive player would then be able to match the metrics to the closest fingerprint using either a decision tree or a nearest-neighbor classifier.

While this was the most intuitive solution from a human point of view, early testing of e fingerprinting plan was not promising. I used WEKA (Hall et al., 2009) to derive a C4.5 decision tree from the logs of the AI vs. AI games, and it only reached a peak of about 40% accuracy several turns into the game. Naive Bayes and SVM classifiers produced similarly poor results. The adaptive player has to be able to accurately classify an opponent in a very small number of moves in order for a switch to be effective, so I abandoned the fingerprinting approach.

Since the adaptive AI has access to the algorithms that are used to enact all of the basic strategies, I decided to simply use those to classify the opponent's play. As soon as a turn ends, the adaptive AI has access to the board states at the start and the end of the turn. These states include all of the opponent's pieces, since players have perfect information. Therefore, the adaptive AI can execute each basic strategy on its opponent's behalf on the board state at the start of the turn and compare the resulting "prediction" to the opponent's positions at the end of the turn. The basic strategy whose ending state most closely matches the opponent's actual ending state, with distance calculated as the total distance from each piece to its corresponding piece in the other state, is the closest match to the opponent's actual strategy.

The adaptive AI plays the game in three stages, using a different approach for strategy selection in each. These stages are early-game, mid-game, and late-game.

The early game is defined as turns 0, 1, and 2. For the very first turn of the game, the adaptive AI has no way of classifying its opponent because no moves have been made, so it chooses randomly among the six basic strategies to plan its moves for turn 0. On turn 1, it cannot be certain what its opponent is doing, but it has some idea of whether its opponent is using an aggressive treasure strategy or some other strategy. Like Schadd et al. (2007), it can use this partial information to plan against larger classes of strategies without having to know yet which specific strategy the opponent is using. In the aggressive-treasure case, the adaptive AI switches

to Greedy Capture. For the other cases (the opponent has started grouping pieces, but the specific strategy is not yet apparent), the adaptive AI uses Hybrid Capture. The reasoning behind this is that it maintains flexibility for future strategy changes: Hybrid Capture will start to form groups, which will be good for a transition into a grouping strategy, but it still does not fall too far behind in treasure collection in case it needs to transition into an aggressive treasure strategy later. On turn 2, the adaptive AI can distinguish fairly well whether its opponent is playing a defensive strategy or a more midrange capture focus. It switches to Defensive Treasure if it predicts Greedy or Hybrid Capture, and uses Best Treasure otherwise. Best Treasure punishes the slow defensive strategies, and Defensive Treasure prepares to defend against capture.

In the mid-game, the adaptive AI classifies its opponent's strategy each turn by using each basic strategy to simulate the moves for the turn, as described above. To refine this approach, it will not classify an opponent as aggressive-treasure (Greedy Treasure or Best Treasure) if its opponent's average blob size is above 1.3 pieces. Even if the opponent is using an aggressive treasure strategy and the pieces just happen to be grouped together and moving in the same direction, it is to the adaptive AI's advantage to classify that as a different strategy that uses larger blobs, because it should be careful to avoid being captured by those large blobs. Also, to avoid switching between strategies too often, the classifier will keep the same guess as the last turn if that strategy's prediction is either closest or second-closest to the board state.

Once it has classified its opponent in each mid-game turn, the adaptive AI switches to a strategy that is strong against the classified strategy. To determine which strategies to use, I considered the balance testing results from the computer vs. computer games, aspects of the different matchups discovered from watching and playing strategies against each other, and an effort to use the same strategy against similar opponent approaches when possible. The adaptive AI uses Greedy Capture against both Greedy Treasure and Best Treasure, Defensive Treasure

against Greedy Capture or Hybrid Capture, Best Treasure against Defensive Treasure, and Greedy Treasure against Defensive Capture.

The adaptive AI uses late-game decision-making whenever the difference in the players' scores is greater than the total number of points in treasures left on the board. Planning should be different in this case because treasures become irrelevant to the player who is behind. Therefore, if the adaptive AI is leading by this large margin, it will transition into Defensive Treasure to avoid capture while trying to end the game on treasures. If it is losing, it will use Greedy Capture in a last attempt to capture one or more opponent pieces. Whichever it uses, it will ignore its opponent classifier and assume the opponent is using the same end-game reasoning: so if it is using Defensive Treasure, it will classify its opponent as Greedy Capture, and vice versa. If the losing player makes enough captures to catch up, and the score difference is no longer greater than the treasure point total, the adaptive AI reverts to its mid-game decision process.

The opponent classification component is used when the adaptive AI is executing a capture strategy. While the basic capture strategies use simple "inertia" to predict where an opponent's piece will be (that is, presume it will keep moving in the direction it was moving last turn), the adaptive AI executes its opponent's strategy on its opponent's pieces to predict where they will go, then uses those predicted locations as targets for its own capture strategy. Opponent prediction could also be used to avoid capture or compete more intelligently for points, but since this type of behavior had no analogue in the basic strategies, I decided not to include it in the adaptive AI's techniques.

3.6 Human-computer games

To test the hypothesis that human players would learn to win more slowly against the adaptive AI than against the basic strategies and would judge it to be more challenging, I ran an experiment with a class of fifty undergraduate business students. Each student was asked to play

seven games (the first two being “practice games” to familiarize them with the game’s mechanics) against one AI (using one of five of the basic strategies), then fill out a brief survey about those games, then play another five games against a different AI (the adaptive one), then answer the same questions about those games. The survey can be found under Appendix A. The survey questions asked about how difficult each AI was to beat and to predict, and how enjoyable the subject found each set of games. The actual win rates for each game would show how quickly the subjects learned to play against their opponents, and the survey would show their perceptions of the difficulty.

Unfortunately, the server side of the game was not prepared to deal with large numbers of people playing large numbers of games against AI opponents. Most of the subjects only got through 2-4 of the first set of games -- enough to form perceptions about the static opponent, but not the adaptive. After fixing the problem, I ran the experiment again with a different class. This one went much better, with most of the students completing 8-9 games plus the practice games (in the latter half of the experiment, games would sometimes end immediately in a tie, not giving the subject a chance to play). Results of this experiment are discussed in Chapter 4.

The successful test was run with a total of 40 undergraduate computer science students enrolled at UNCW in CSC 221, Introduction to Computer Science II. There were two sections, one with 21 students and one with 19. The class was held in a lab, with a computer for each student. The students were not notified beforehand that they would be invited to participate in an experiment that day. Demographic data collected in the survey indicated that there were 8 female students and 32 male. Four students did not give an age, but in those who answered, ages ranged from 19 to 62 years, with an average age of 24.3 years. The survey also asked what types of computer games the subjects typically played. 20 students included strategy games in their answer, and 7 indicated that they did not typically play games.

Chapter 4: Results

4.1 Balancing basic strategies

In preliminary balance testing, treasure-based strategies were vastly superior to capture-based ones. There were only 10 treasures on the board, worth anywhere from 1 to 15 points each, a piece had to be surrounded on 7 out of 8 sides to be captured, and piece capture was only worth 20 points. This resulted in Best Treasure dominating, with capture strategies (at the time limited to Greedy and Defensive) unable to achieve a favorable win rate. The table shows the win rates for each strategy in each matchup, with the numbers on a strategy's row corresponding to the percentage of games that that strategy won.

Table 1

Win Rates (%) for Basic Strategies – First Test

Strategy	Greedy Treasure	Best Treasure	Greedy Capture	Def. Treasure	Def. Capture
GreedyTreasure		32	**80**	**90**	**95**
BestTreasure	**68**		**85**	**80**	**80**
GreedyCapture	20	15		20	**70**
Def.Treasure	10	20	**80**		**80**
Def.Capture	5	20	30	20	

To balance these strategies, I had to make capturing easier and worth more, while preventing the aggressive strategies from winning the game too early. Increasing the number of treasures on the board to 20 with values from 1 to 8, reducing the surround requirement to 6 sides out of 8, and raising piece value to 30 helped the capture strategies, but Defensive Treasure was strengthened too much. I also introduced Hybrid Capture, because Defensive Capture appeared to be hopelessly terrible.

Table 2

Win Rates (%) for Basic Strategies – Intermediate Test

Strategy	Greedy Treasure	Best Treasure	Greedy Capture	Def. Treasure	Hybrid Capture
GreedyTreasure		41	44	48	46
BestTreasure	59		48	50	42
GreedyCapture	56	52		36	64
Def.Treasure	52	50	64		56
HybridCapture	54	58	36	44	

After many iterations with different parameter settings and tweaks to the AI logic, I achieved the best results with the current game rules: 15 treasures valued 1-7, a surround requirement of 6 sides, and a piece value of 30. This test, shown in Table 3, was my primary consideration when choosing what strategies the adaptive AI would use against different opponent strategies.

Table 3

Win Rates (%) for Basic Strategies – Test Used for Adaptive AI

Strategy	Greedy Treasure	Best Treasure	Greedy Capture	Def. Treasure	Hybrid Capture	Def. Capture
GreedyTreasure		46	52	48	49	78
BestTreasure	54		49	55	50	72
GreedyCapture	48	51		45	54	69
Def.Treasure	52	45	55		52	65
HybridCapture	51	50	46	48		65
Def.Capture	22	28	31	35	35	

In re-examining the Defensive Treasure code, I found that I had mistakenly given it a different method for grouping its pieces than the other grouping strategies. A piece that was

seeking an ally would look at where the target piece planned to go that turn instead of where it was, which would make this grouping method more efficient in many cases. After I altered the other grouping strategies to use this same method, Defensive Treasure did much worse in testing, shown in Table 4. Though this change affected the adaptive AI, as will become apparent in the experiment results, I mistakenly did not change the strategies the AI uses in response to its opponent to reflect these new test results.

Table 4

Win Rates (%) for Basic Strategies – Final Test

Strategy	Greedy Treasure	Best Treasure	Greedy Capture	Def. Treasure	Hybrid Capture	Def. Capture
GreedyTreasure		46	38	64	43	54
BestTreasure	54		43	61	41	49
GreedyCapture	62	57		51	58	54
Def.Treasure	36	39	49		40	52
HybridCapture	57	59	42	60		46
Def.Capture	46	51	46	48	54	

Further adjustment of game rules and AI techniques, perhaps reducing the value of capture or making the defensive strategies keep their grouped pieces closer together, could restore the utility of Defensive Treasure while ending the tyranny of Greedy Capture. I was unable to balance the strategies any more before conducting the human versus computer trials.

Both this imbalance and the incorrect strategy choices of the adaptive AI became apparent when I tested the adaptive AI against each basic strategy. It should do well against all of them, since it can classify them very accurately and in theory exploit their weaknesses. Results of the test are shown in Table 5.

Table 5

Win Rates (%) for Adaptive Strategy vs. Basic Strategies

Strategy	Greedy Treasure	Best Treasure	Greedy Capture	Def. Treasure	Hybrid Capture	Def. Capture
Adaptive	51	55	44	59	42	62

This shows both the new weakness of Defensive Treasure, which is the strategy the adaptive AI uses against Greedy Capture, and the poor choice of Defensive Treasure against Hybrid Capture, when Greedy or perhaps Defensive Capture would have been better.

4.2 Human vs. computer experiment

I hypothesized that humans would learn to beat the adaptive AI more slowly than they would learn to beat a static AI. To see if the experiment subjects would actually learn to beat either AI opponent over the course of several games, I looked at the percentage of human wins for each game in the sequence. Each human played two practice games and five experiment games against the same basic AI player. Figure 4 shows that while the humans faltered in the middle of these games, there is a general upward trend, indicating that they learned to play better against this basic opponent over the course of seven games.

Games 6 through 9 were played against the adaptive AI. Subjects were told that the AI player would be different after Game 5. While the human win rate actually went up for the first game against the adaptive player, the downward trend for these games indicates that the humans did not learn to play effectively against it in the course of four games. The experiment was designed to go through Game 10, but few people finished all ten games due to technical difficulties, so it was not included in the analysis.

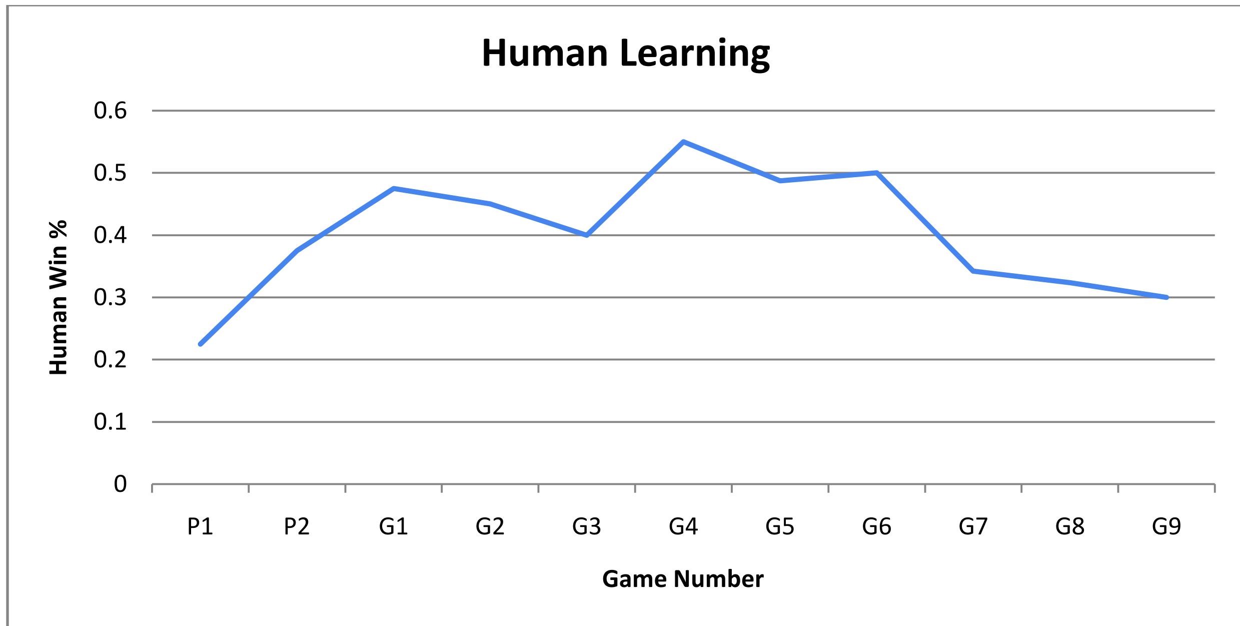


Figure 4. Human win rates over the course of the experiment, from Practice Game 1 (P1) through Game 9 (G9).

Since each player scores points and the point totals are used to decide the winner, the difference between the average human score and the average AI score provides a more fine-grained description of how the humans did over the games. If humans were losing just as many games, but by fewer points, this would also suggest learning. Again, though, the upward trend (with the exception of the dramatic drop in Game 3) against the basic AI and the downward trend against the adaptive are apparent in Figure 5. These trends support Hypothesis 1: not only did the humans improve more slowly against the adaptive AI, they did not show appreciable improvement against it at all in the limited number of games they played, while they did improve against the static opponent.

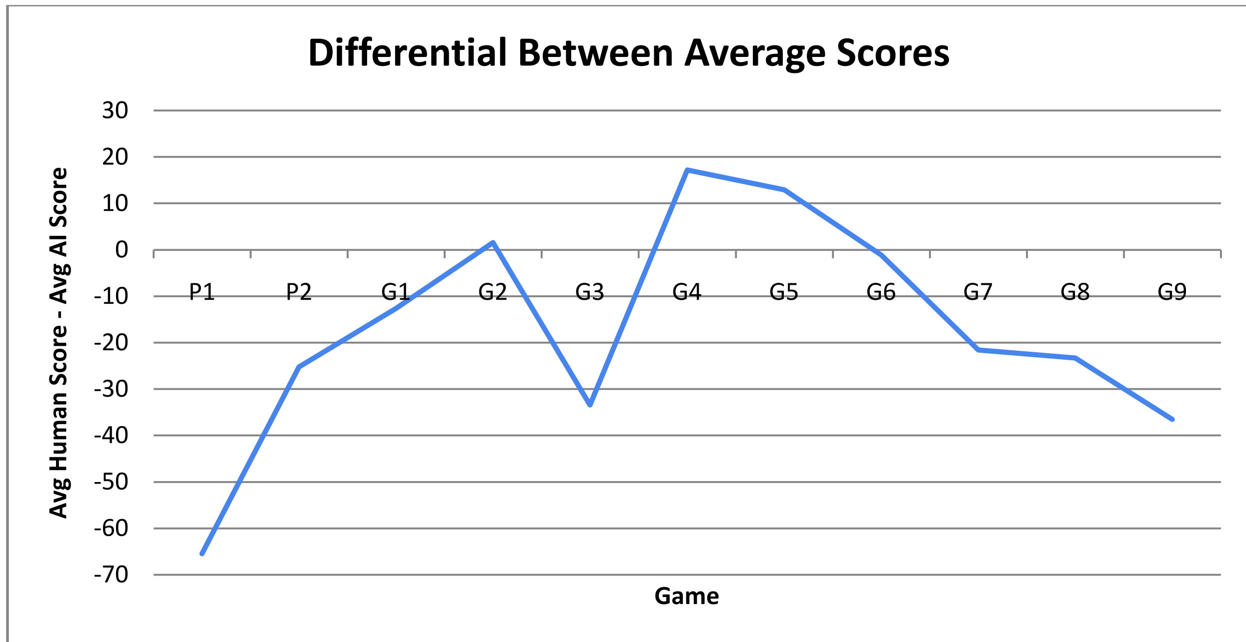


Figure 5. Differential between human and AI scores over the course of the experiment.

Since different human subjects played against different basic AI strategies for the first half of the experiment, I broke out the score differentials by AI opponent to see if learning rates were different (Figure 6). Seven to nine humans played against each basic strategy. Only the humans whose basic opponent was Best Treasure showed notable improvement against the adaptive AI. The different series from Figure 6 can be viewed separately in Appendix B.

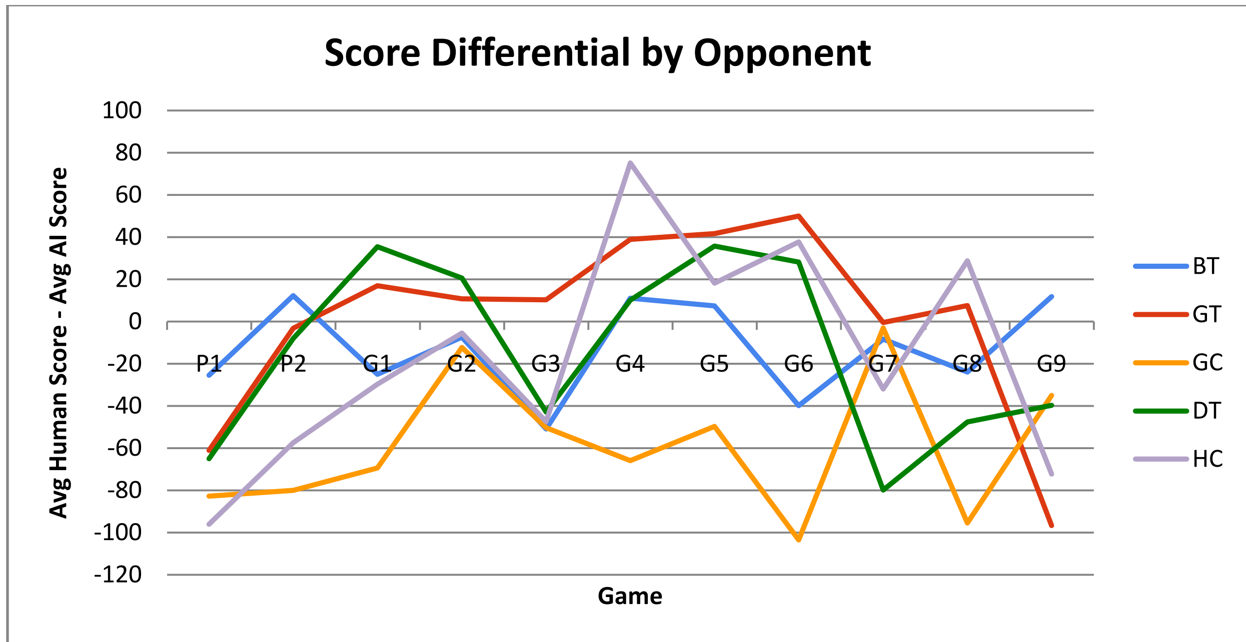


Figure 6. Differential between human and AI scores, broken out by basic AI opponent in games P1-G5.

These different learning curves for the different static AIs suggest that the different groups of subjects had very different experiences over the course of the games. Figure 7 shows human win rates against the static AI, against the adaptive AI, and overall, by group. Notably, Greedy Treasure appears to be adequate preparation for the adaptive AI, but Best Treasure better prepares the subjects to improve on their static performance. All of the other groups did worse against the adaptive AI than against their static opponents.

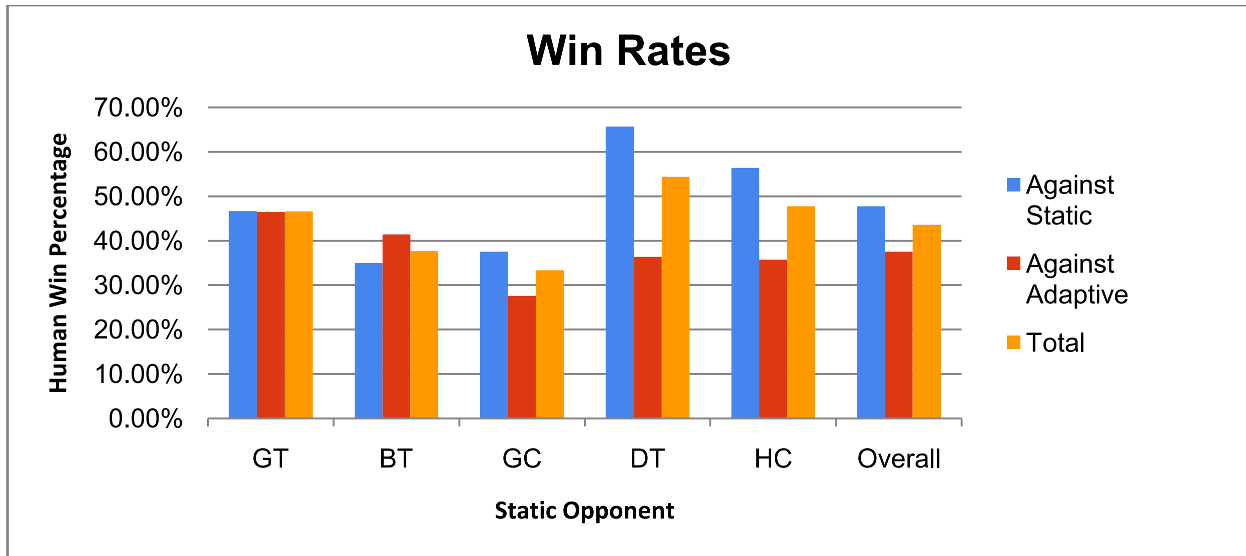


Figure 7. Human win rates for the different parts of the experiment, broken out by static opponent.

I also hypothesized that subjects would judge the adaptive AI to be more challenging than a static one. To test this, I gathered data about the subjects' perception of their opponents and the game through survey questions. For each half of the experiment, subjects were asked what strategy they thought the AI was using, how difficult the AI was to beat and to predict, and how enjoyable the games were. Subject perception of difficulty is shown in Figure 8.

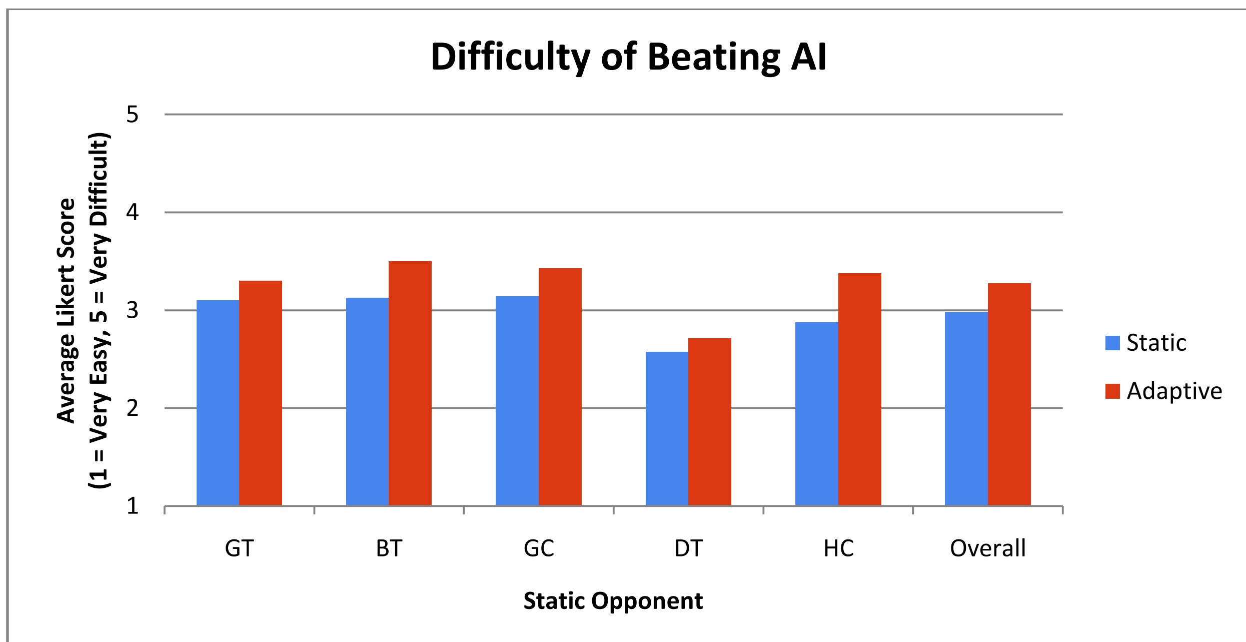


Figure 8. Comparison of subject perception of static and adaptive AI difficulty.

These results indicate that the subjects who played Defensive Treasure or Hybrid Capture accurately perceived those as generally easier to beat than the other static strategies. The other groups scored their strategies about the same, despite Best Treasure and Greedy Capture being particularly hard to beat. Also, the Defensive Treasure group registered the smallest increase in difficulty from static to adaptive, despite the fact that their adaptive win rate was dramatically lower than their rate against static. All the groups ranked the adaptive opponent as harder to beat, even the Best Treasure group that did better against it than against the static opponent. This supports Hypothesis 2, which was that humans would think the adaptive AI was more challenging than the static one. A t-test on the static vs. adaptive difficulty perception scores gave a p value of 0.0482, indicating a statistically significant increase in difficulty perception from static to adaptive.

The question about the predictability of the AI opponent also produced interesting results, shown in Figure 9. All of the groups saw the adaptive AI as being harder to predict, as expected. The p value for this t-test was 0.00313, again indicating a statistically significant difference between the static and adaptive AIs. Perceptions of predictability did not correspond with perceived ease of defeating an opponent: the Defensive Treasure group reported having the hardest time predicting their static opponent, yet they had the highest static win rate and perceived their static opponent as being easy to beat. Meanwhile, the Best Treasure and Greedy Capture groups thought that their static opponents were easy to predict, yet they had the lowest static win rates.

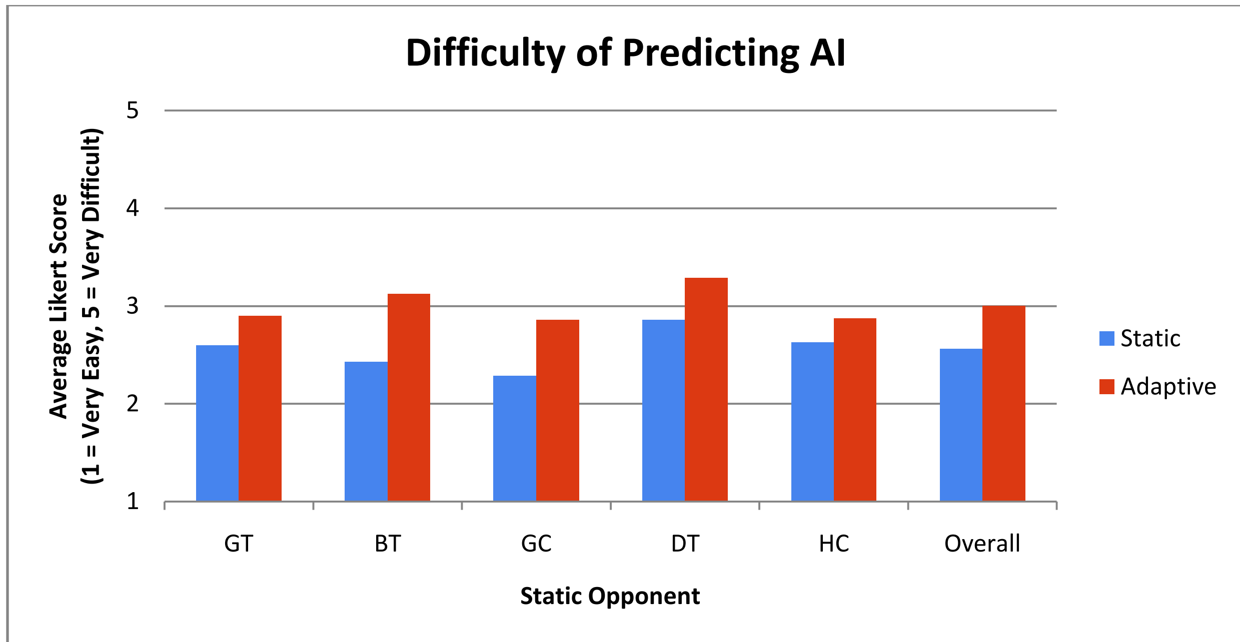


Figure 9. Subject perception of difficulty predicting AI opponent, by static opponent.

I was also curious to see whether this group of subjects enjoyed their games against the adaptive AI more than those against the static one. Game enjoyability results are shown in Figure 10. Of course, different people have different preferences about how much challenge and how much predictability they prefer in their games, so this should vary considerably. Interestingly, no group found the adaptive games more enjoyable, and the Greedy Capture group, which had a low static win rate and found their static opponent to be very predictable, reported the most enjoyment of the static games. Differences in enjoyment levels between static and adaptive games were small except for the Greedy Capture group, and they were not statistically significant, with a t-test giving a p value of 0.0677.

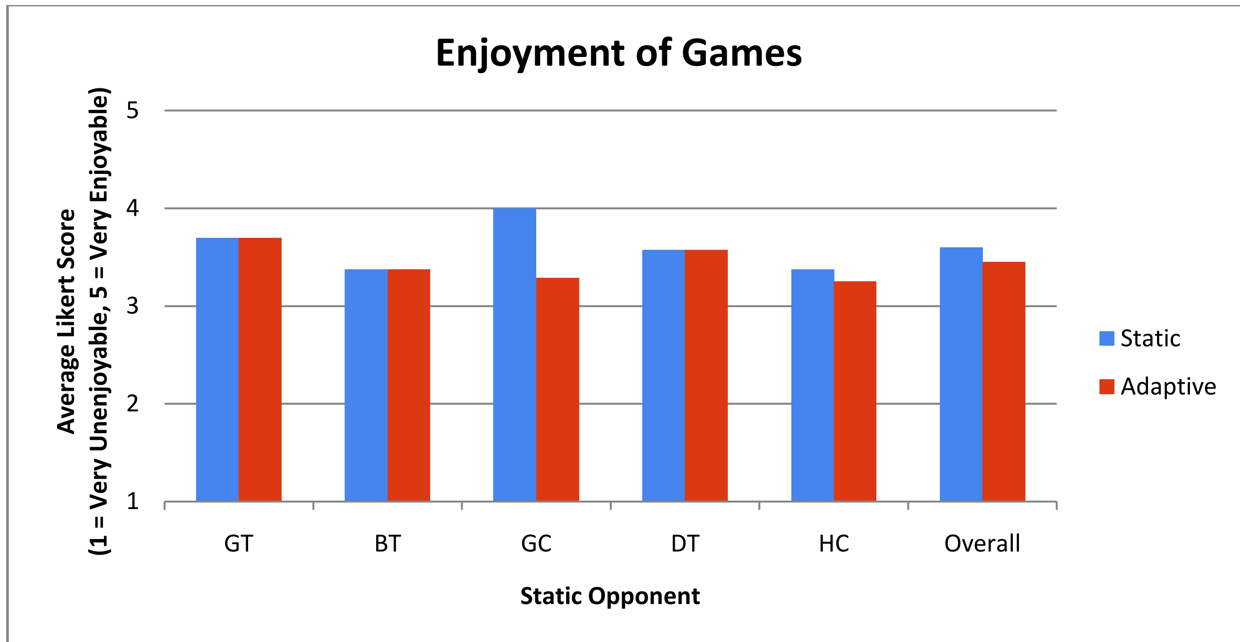


Figure 10. Subject enjoyment of gameplay, by static opponent.

I asked the subjects to describe what strategy they thought the opponent was using for two main reasons: to get an idea of whether they could accurately predict their opponents' moves (however easy they might perceive that to be), and to see whether they could figure out that the adaptive AI was using a more complex strategy than the basic one. Since the answers were freeform text responses, categorizing them in a useful way was somewhat subjective, but I scored their answers for the static opponent as 0, 1, or 2, with 0 indicating that they were entirely wrong, 2 indicating acceptable accuracy (considering that they had not played the game before, had only seen one strategy, and didn't know the different possible categories), and 1 being about half right (e.g., they understood that it was grouping its pieces but thought it was going for capture rather than treasure). The average score overall was 1.15, with the Greedy Capture and Defensive Treasure groups leading the pack with an average of 1.43 each, and Greedy Treasure trailing with an average of 0.90. I found only a very weak correlation between strategy guess points and win rates against the static opponent.

As expected, none of the subjects guessed exactly what the adaptive AI was doing. In order to detect an increase in the complexity of the guessed strategies, I counted the number of people who included any of the following in their descriptions of the AI's strategy:

1) Multiple objectives (which implies some means of prioritizing different goals)
2) Game context sensitivity (e.g. "pursue the treasures to win once in the lead")
3) Special tactics not present in the basic strategies (e.g. flanking or fleeing opponent)

Overall, 10 of the 40 participants used one or more of these in their descriptions of the static AI. One of those was someone correctly describing the multi-objective Hybrid Capture. In describing the adaptive AI, 17 of the 40 subjects used one or more of these complexity indicators, including 7 references to context sensitivity in particular, which was the closest approximation to the actual major improvement of the adaptive AI over the basic ones. Most of the subjects described essentially a basic strategy when asked to characterize the adaptive AI, indicating that it is not easy for humans who are new to the game to detect a more complex AI strategy as it is being used against them.

I gathered some demographic information in the first part of the survey: subjects indicated their age and gender and what types of computer games, if any, they typically played. Since the experiment was run on a group of undergraduate computer science students, most of the subjects were in the same narrow age range (20-23). Most were male, and most indicated that they played some types of computer games. There were not enough female or non-gamer subjects to draw any conclusions about their win rates or survey answers. Half of the subjects answered that they typically played strategy games, which I thought might give them an advantage since Boundary is a strategy game, but there was no notable correlation between whether subjects played strategy games and win rates or survey answers.

Chapter 5: Discussion and Conclusion

5.1 Hypotheses

5.1.1 Hypothesis 1: Human Learning

One measure for determining how well humans learned to play the game is examining the win rate as the games progressed, since effective play leads to winning games. The subjects won more games by the time they got to game 4 or 5 (55% and 49%) against the static AI than they did in practice (23% and 38%) or in games 1-2 (48% and 45%). While there was an unexplained drop in the win rate in game 3 (40%), the trend was generally toward a higher win rate as the humans played more games against the static AI, with the slope of the best-fit line being 0.037.

The comparison of this increase to the trend for the adaptive games was my test for Hypothesis 1. The expectation was that the subjects' win rate would have a large positive slope for the first half of the experiment, drop sharply at the divide between static and adaptive, and then have a smaller positive slope over the adaptive games. The results did not exactly follow this pattern, because the subjects actually got progressively worse against the adaptive player, with a win rate slope of -0.061. The increase over the static games was indeed larger than the one over the adaptive games, so the win rates indicating the subjects' learning support Hypothesis 1.

The win rate for game 6 (50%) was particularly surprising. Even though game 6 brought an entirely new AI opponent, subjects maintained their high win rate of games 4-5 for this game, then did dramatically worse in games 7-9 (34%, 32%, and 30%). This cannot be explained by the AI alone, because the adaptive AI did not change its overall strategy from one game to the next. We currently do not have a strong hypothesis for why subjects did so well in game 6, but I speculate that a psychological effect on the subjects influenced their performance. Since players demonstrated that they learned to play better against the static AI, the typical player probably developed a fairly focused strategy in the first half of the experiment, and then carried that

strategy into game 6 against the adaptive AI. Because the strategy had a strong focus, it did reasonably well in that game, but the player perceived the game as going worse than games 4 and 5 because the opponent changed and the game was less straightforward and predictable. Because of the perceived unpredictability of the AI opponent, in games 7-9, the player adjusted and hybridized the strategy. The player's strategy lost its focus and performed poorly, and there were not enough games remaining in the experiment to develop a more effective one. Whatever the explanation, the unexpected win rate for game 6 suggests that the changes in human win rates were due to more than the specific differences between the AI opponents' strategies. In the Future Work section, we will discuss future experiments that may help elucidate the cause of the rise and drop of human win rates.

5.1.2 Hypothesis 2: Human Perception of Difficulty

The survey asked the subjects directly how challenging they found the games against the different AIs, and their responses indicated that they found the adaptive AI significantly more challenging. The overall average difficulty rating for the static AI was a 2.975 on a scale of 1 to 5 (with 1 being "Very Easy" and 5 being "Very Difficult"), while the average for the adaptive AI was 3.275. These results support Hypothesis 2, and the t-test gave a p-value of 0.0482, so the difference is statistically significant. The amount of increase varied by which static AI a subject played against, but the average difficulty rating for each group did increase.

Interestingly, subjects who played against the easiest AI to beat, Defensive Treasure, gave the lowest challenge rating of all the groups to the adaptive AI as well as the static one, even though their adaptive win rate was dramatically lower than their static one and was on par with the other groups' adaptive win rates. Furthermore, the Best Treasure group, which did better against the adaptive AI than against the static one, still ranked the adaptive AI as more

challenging. These results suggest that there are more factors involved in human perception of difficulty than just the actual difficulty level as indicated by game wins.

5.2 Future work

While the results of the experiment support the hypotheses of this thesis, they also raise additional questions. At what point would the humans have started to improve against the adaptive AI? At what point would their win rate have leveled off against the static AI, and would more games in the first half have led to faster improvement in the second? Another test with more games per opponent would further explore Hypothesis 1.

The results from the questions about game enjoyment were neither conclusive nor directly relevant to my hypotheses, but they raise interesting questions as well. The Greedy Capture group, which ranked its static opponent as most predictable, also gave it the highest enjoyment score of the static AIs and much preferred it to the adaptive player. Do players actually prefer a more predictable opponent? These were essentially first impressions from players entirely new to the game – how many games would it take for the static opponent's high predictability to bring down the difficulty and enjoyment scores? An experiment with more detailed survey questions, more subjects, and more games might shed some light on the (probably complicated) relationship among amount of game experience, human win rates, perceived challenge level of the AI player, actual and perceived predictability of the AI player's moves, and enjoyment of the game.

The literature suggested that adaptive difficulty scaling was desirable in an AI player because it would provide a consistent challenge over many games, but it would not be so challenging as to discourage the human player. Adding difficulty scaling to this adaptive AI would probably increase player enjoyment, but perhaps not over such a small number of games. A truly adaptive AI player should adapt to its opponent not only over the course of one game, but

over its entire series of games with a player. The current adaptive AI has no memory of previous games, but it could be improved to keep a profile of a player, with some measure of skill level and preferred strategies. This customized AI would give a better impression of an opponent that adapted and changed over time, like a human practice partner would, and this would add value to the player's experience with the AI.

5.3 Conclusion

There is plenty of room for improvement in current commercial game AI, and research in this field has shown that it can be improved using simple techniques without monopolizing the resources of the game developers. An adaptive game agent is a worthwhile addition to a game not only because it retains the human player's interest better, but also because it does not require a game expert to write long, complicated scripts for it to follow. Researchers in game AI have approached the problem of adaptive agents in various ways, focusing on different areas of improvement.

I developed Boundary as a simplified testbed to show that an adaptive agent could be written to approach one in-game problem – strategic piece movement – by combining fairly simple ideas and techniques into something better. The parts of the game development process that were exclusive to the adaptive AI were fairly small; besides the opponent classification and the specialized early- and late-game reasoning, everything contributed to the game itself, because the basic strategies and computer-vs.-computer tests helped to produce a balanced game.

Finally, I tested this adaptive AI against humans, comparing its performance to that of the static agents. The results of the experiment supported my hypotheses that players would find it more challenging and would learn to beat it more slowly. I can conclude that from simple pieces, I have produced a more advanced game agent that will provide a more lasting challenge to its human opponents.

References

D. Aha et al., "Learning to win: Case-based plan selection in a real-time strategy game," in Proceedings of the 6th International Conference on Case-Based Reasoning, 2005.

R. Baker and P. Cowling, "Bayesian opponent modeling in a simple poker environment," in Computational Intelligence and Games, 2007.

M. Bowling et al., "Machine learning and games," in Machine Learning, 2006.

M. Buro, "Call for AI research in RTS games," in Proceedings of the AAAI-04 Workshop on Challenges in Game AI, 2004.

M. Buro and T. Furtak, "RTS games as test-bed for real-time AI research," in Proceedings of the 7th Joint Conference on Information Science, 2003.

D. Cheng and R. Thawonmas, "Case-based plan recognition for real-time strategy games," in Proceedings of the Fifth Game-On International Conference, 2004.

M. Coram and S. Bohner, "The impact of agile methods on software project management," in Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, 2005.

M. Cutumisu et al., "ScriptEase: A generative/adaptive programming paradigm for game scripting," in *Science of Computer Programming*, 2007.

A. Davidson et al., "Improved opponent modeling in poker," in International Conference on Artificial Intelligence, 2000.

I. Davis, "Strategies for strategy game AI," in Proceedings of the AAAI Spring Symposium on Artificial Intelligence and Computer Games, 1999.

M. Eladhari and E. Ollila, "Design for research results: experimental prototyping and play testing using iterative game design," in Simulation Gaming, 2012.

M. Fagan and P. Cunningham, "Case-based plan recognition in computer games," in Proceedings of the 5th International Conference on Case-Based Reasoning, 2003.

T. Fayard, "Using a planner to balance real time strategy video game," in Workshop on Planning in Games, ICAPS, 2007.

M. Hall et al., “The WEKA data mining software: an update,” in SIGKDD Explorations, 2009.

S. He et al., "Creating challengeable and satisfactory game opponent by the use of CI approaches," in International Journal of Advancements in Computing Technology, 2010.

T. Hinrichs and K. Forbus, "Analogical learning in a turn-based strategy game," in Proceedings of the 20th International Joint Conference on Artificial Intelligence, 2007.

K. Laviers et al., "Improving offensive performance through opponent modeling," in Proceedings of the Fifth Artificial Intelligence for Interactive Digital Entertainment Conference, 2009.

P. Moreno-Ger et al., "A documental approach to adventure game development," in *Science of Computer Programming*, 2007.

J. Novak, *Game Development Essentials.* Clifton Park, NY: Thomson Delmar Learning, 2005.

M. Ponsen et al., "Knowledge acquisition for adaptive game AI," in Science of Computer Programming, 2007.

S. Rabin, *Introduction to Game Development, Second Edition*. Boston, MA: Cengage Learning, 2010.

M. Richards and E. Amir, "Opponent modeling in Scrabble," in Proceedings of the 20th International Joint Conference on Artificial Intelligence, 2007.

C. Salge et al., "Using genetically optimized artificial intelligence to improve gameplaying fun for strategical games," in Proceedings of the 2008 ACM SIGGRAPH symposium on Video games, 2008.

F. Schadd et al., "Opponent modeling in real-time strategy games," in GAMEON, 2007.

P. Spronck et al., "Adaptive game AI with dynamic scripting," in Machine Learning, 2006.

B. Weber and M. Mateas, "A data mining approach to strategy prediction," in Computational Intelligence and Games, 2009.

Appendix A

Experiment Survey

Appendix A

Survey

Age: ______

Gender (circle): Male Female

What types of computer games do you typically play? (Check all that apply.)

- ❑ Board/Card (e.g. Risk, poker)
- ❑ Puzzle (e.g. Tetris)
- ❑ First Person Shooters (e.g. Call of Duty)
- ❑ Strategy (e.g. StarCraft, League of Legends)
- ❑ Fighting (e.g. Street Fighter)
- ❑ Casual (e.g. Angry Birds)
- ❑ Platformers (e.g. Mario)
- ❑ Simulation (e.g. The Sims)
- ❑ Role-playing (e.g. Final Fantasy, Skyrim)
- ❑ Sports (e.g. FIFA)
- ❑ Other
- ❑ None / I don't typically play computer games

Part 1: Games 1-5

1. In 1-2 sentences, describe what strategy you think the computer is using.

2. How difficult is it to beat the computer?

○	○	○	○	○
Very Easy	Easy	Neutral	Difficult	Very Difficult

3. How difficult is it to predict the computer's moves?

○	○	○	○	○
Very Easy	Easy	Neutral	Difficult	Very Difficult

4. How enjoyable did you find these games?

○	○	○	○	○
Very Unenjoyable	Unenjoyable	Neutral	Enjoyable	Very Enjoyable

Part 2: Games 6-10

1. In 1-2 sentences, describe what strategy you think the computer is using.

2. How difficult is it to beat the computer?

○	○	○	○	○
Very Easy	Easy	Neutral	Difficult	Very Difficult

3. How difficult is it to predict the computer's moves?

○	○	○	○	○
Very Easy	Easy	Neutral	Difficult	Very Difficult

4. How enjoyable did you find these games?

○	○	○	○	○
Very Unenjoyable	Unenjoyable	Neutral	Enjoyable	Very Enjoyable

5. How much do you agree with the following statement?

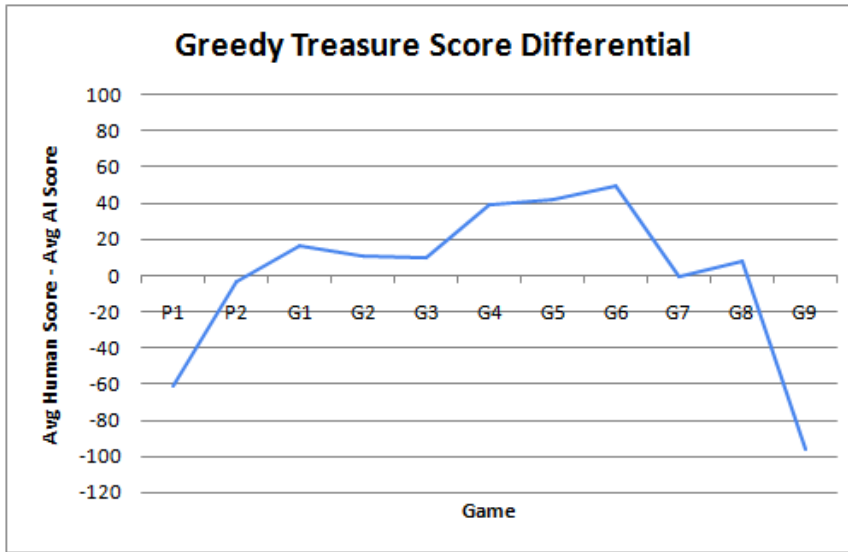
The first strategy played more like a human than the second strategy did.

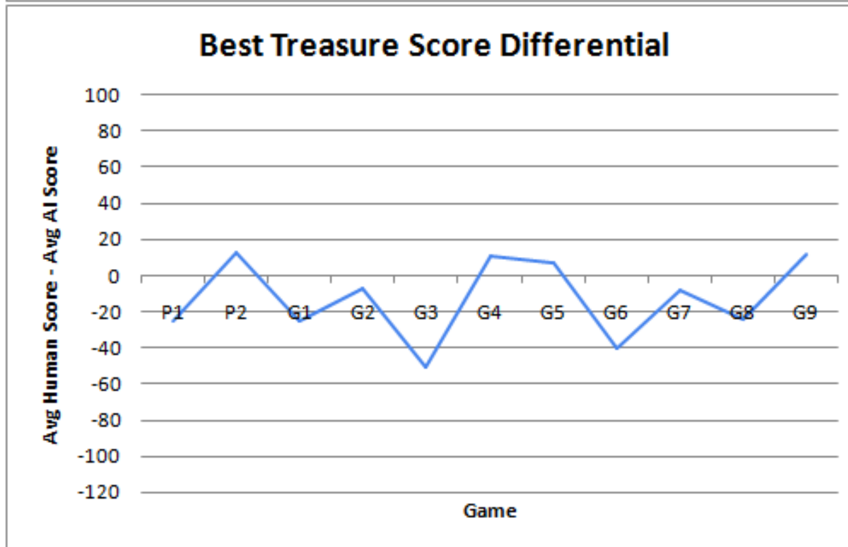
○	○	○	○	○
Strongly Agree	Agree	Neither Agree nor Disagree	Disagree	Strongly Disagree

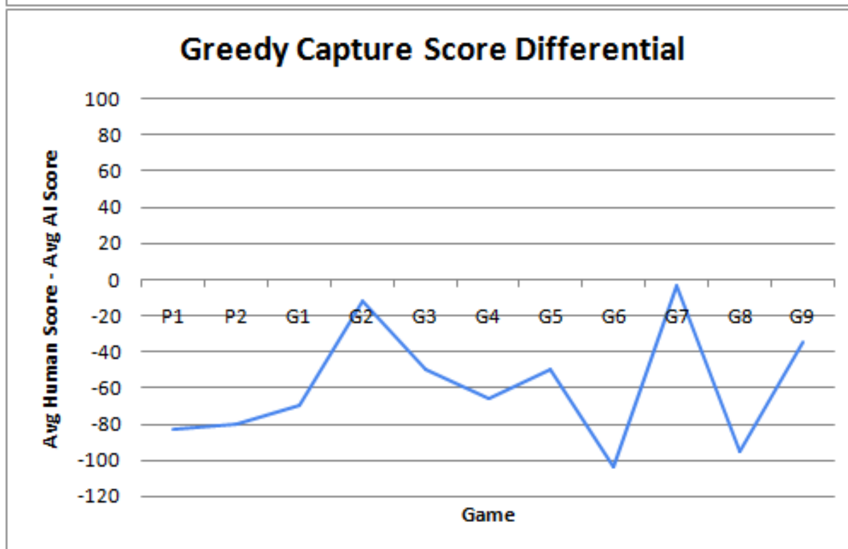
Appendix B

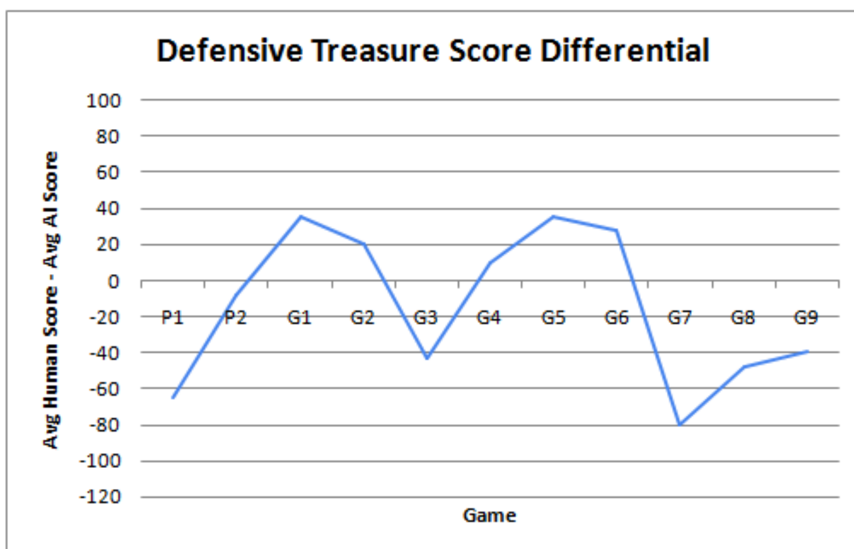
Score Differentials by Static Opponent

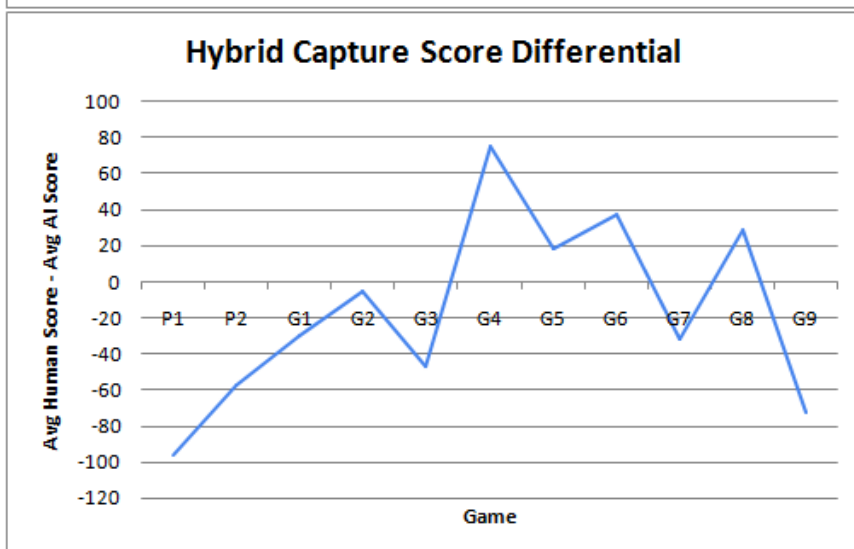
Appendix B








Defensive Treasure Score Differential
100
80
60
40
20
0
-20
-40
-60
-80
-100
-120
Avg Human Score - Avg AI Score
P1
P2
G1
G2
G3
G4
G5
G6
G7
G8
G9
Game


Hybrid Capture Score Differential
100
80
60
40
20
0
-20
-40
-60
-80
-100
-120
Avg Human Score - Avg AI Score
P1
P2
G1
G2
G3
G4
G5
G6
G7
G8
G9
Game