University of Alberta

Move Groups as a General Enhancement for Monte Carlo Tree Search

by

Gabriel Van Eyck

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

©Gabriel Van Eyck
Spring 2014
Edmonton, Alberta

Permission is hereby granted to the University of Alberta Libraries to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only. Where the thesis is converted to, or otherwise made available in digital form, the University of Alberta will advise potential users of the thesis of these terms.

The author reserves all other publication and other rights in association with the copyright in the thesis and, except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatsoever without the author's prior written permission.

Abstract

Monte Carlo tree search (MCTS) combined with the upper confidence bounds applied to trees (UCT) algorithm has brought forth many advances in game related AI. This includes general game players and programs for specific games such as Amazons, Arimaa, and Go. However, there often is a need for further enhancements beyond this to make the strongest player for a given game. Move groups are investigated in detail to determine their effect on MCTS as a general enhancement. The structures of move groups that increase performance in an artificial game are analyzed then applied without knowledge of move values. We find that there is a speed increase in the tree while maintaining performance for a fixed number of simulations. Finally, move groups are applied to the game of Amazons successfully.

Acknowledgements

Due thanks I must profess

To him who more or less

  Stayed by my side

  To be my guide

Though often I caused distress.

Contents

1 Introduction **1**

2 Background **3**

2.1 Monte Carlo Tree Search 3

2.2 UCB and UCT 4

2.3 UCT in Games 6

2.4 Move Groups 6

2.5 Other Enhancements 8

3 Experiments with Nine Nodes **10**

3.1 Experimental Framework 10

3.2 An Easy Problem 11

3.3 A Hard Problem 18

3.4 Group Selection Rate Performance with Changing Difficulty 26

4 Using Move Groups with Knowledge **29**

4.1 Creating Groups in an Easy Problem 30

4.2 Creating Groups in a Hard Problem 32

4.3 Creating Groups in an Easy Problem with Prior Knowledge 33

4.4 Creating Groups in a Hard Problem with Prior Knowledge 34

5 Experiments with Twenty Seven Nodes **36**

5.1 Many Groups of Few Nodes 39

5.2 Few Groups of Many Nodes 40

5.3 Multiple Levels of Groups 40

6 Move Groups in the Game of Amazons **42**

6.1 Implementation Details 42

6.2 Experiments with Amazons 43

7 Conclusions and Future Work **45**

7.1 Research Questions Revisited 45

7.2 Other Future Work 47

7.2.1 Theoretical Approach for Move Groups 47

7.2.2 Additional Amazons Move Groups Experiments 47

7.2.3 Move Groups in Other Games 48

List of Tables

3.1 Group performance compared to no groups with the same parameters 15

4.1 Performance of different grouping methods with target structure 123 and best node 0.9. 31

4.2 Performance of different grouping methods with target structure 134 and best node 0.9. 31

4.3 Performance of different grouping methods with target structure 123 and best node 0.81. 32

4.4 Performance of different grouping methods with target structure 134 and best node 0.81. 33

4.5 Performance of using prior knowledge with target structure 123 and best node 0.9. 34

4.6 Performance of using prior knowledge with target structure 134 and best node 0.9. 34

4.7 Performance of using prior knowledge with the target structure being 123 and the best node being 0.81. 35

4.8 Performance of using prior knowledge with target structure 134 and best node 0.81. 35

5.1 Performance of selected groups with 27 nodes. 39

List of Figures

2.1 The Monte Carlo Tree Search process [3]. 4

2.2 The starting board for Amazons after one move has been played. 7

2.3 An example for move groups with five moves and two groups. 7

3.1 An example grouping of nine nodes. 12

3.2 Base case selection rate performance on nine nodes, valued 0.1 to 0.9. 14

3.3 The average value of groups vs. selection rate performance with the best node valued 0.9 and $C = 1$. 16

3.4 The average value of groups vs. selection rate performance with the best node valued 0.9 and $C = 5$. 17

3.5 Base case simple regret performance with nine nodes, valued 0.1 to 0.9. 19

3.6 The average value of groups vs. simple regret performance with the best node valued 0.9 and $C = 1$. 20

3.7 Cumulative regret performance of base case and groups after 512 simulations. 21

3.8 Cumulative regret performance of base case and groups after 8192 simulations. 21

3.9 Base case selection rate performance with nine nodes, valued 0.1 to 0.81. 22

3.10 The average value of groups vs. selection rate performance with the best node valued 0.81 and $C = 1$. 24

3.11 The average value of groups vs. selection rate performance with the best node valued 0.81 and $C = 5$. 25

3.12 The performance of a single group vs. the number of simulations with $C = 1$. 27

3.13 The performance of a single group vs. the number of simulations with $C = 5$ 28

5.1 Base case performance without groups with 27 nodes. 38

CHAPTER 1

Introduction

Monte Carlo Tree Search (MCTS) [3] has allowed for many recent advances in the field of Artificial Intelligence for games since the development of the Upper Confidence Bounds Applied to Trees algorithm (UCT) [18]. Game players for games such as Go [12], Amazons [17, 15], and Arimaa [19] have had their playing strength vastly increased by using this algorithm. However, a state of the art player does not consist only of an UCT implementation, but of other techniques as well. These techniques may include using prior knowledge [9], Rapid Action Value Estimation (RAVE) [11], simulation policies [21], opening books [16], progressive unpruning [4], and move groups [5]. Using all of these techniques effectively on a single game may prove difficult due to the nature of the game itself and the problems each of the techniques aim to solve. In the case of prior knowledge, simulation policies, and progressive unpruning, game specific knowledge is required to implement them. Opening books can be constructed for any game but do not improve the player throughout the game. RAVE can be applied to game players that use MCTS if there is a set of moves that is small enough, as in Go, and do not change drastically in value over the course of the game. Move groups can be applied to nearly any game, but the previous research has not investigated their effects in detail.

Given the number of techniques available for use when constructing a MCTS player, it is no surprise that there is much trial and error when testing the different techniques. If a general enhancement could be found for MCTS that does not involve game specific knowledge, then the playing strength of players in general would be improved. This brings us to move groups. Move groups simply group possible moves together at any stage in the game tree. So rather than choosing from all possible moves at a given game state, you first select a move group, then an actual move. This reduces the branching factor of the game tree at the cost of increasing the height. Most research on move groups up to this point has consisted

of implementing move groups for a specific game and examining the results [5, 19, 15]. This thesis aims to investigate the effects of move groups in detail in order to determine the best way to implement them in general.

I aim to answer the research questions detailed here:

- Does there exist a move group that performs better on average than using no groups, given the same number of simulations?
- Will a randomly selected move group perform better on average than no move group?
- What is the general structure of a move group that performs better on average than no move group?
- How do you implement move groups in MCTS to improve performance?

In this thesis, the content is structured as follows. Chapter 2 covers background information on the topic of move groups and games. Chapter 3 details the first set of experiments involving groups in an artificial game with nine nodes. Next, Chapter 4 investigates how to use prior knowledge in order to create move groups. After that, the game tree size is extended to 27 nodes with different group configurations in Chapter 5. Chapter 6 looks at experiments done in the game of Amazons with move groups. Lastly, conclusions and future work are discussed in Chapter 7. Portions of Chapter 3 and 4 have been published in a previous paper [23].

CHAPTER 2

Background

2.1 Monte Carlo Tree Search

Monte Carlo tree search (MCTS) is a best-first search technique that uses random simulations to gather statistics on a tree [3]. In games, it can be used to randomly simulate the possible games after a sequence of moves have been played. Since random games can be played quickly, after a number of games, a good move can be selected based on the accumulated statistics. As shown in Figure 2.1, there are four stages to one simulation. First, a node in the tree is selected. This selection process takes into account previously gathered statistics in order to balance exploration, *i.e.* the number of nodes simulated and the certainty of their evaluation, and exploitation, *i.e.* simulating the most promising nodes. Then, after reaching a leaf node, a potential move from that node is expanded. This expands the tree by one node for every simulation. Next, a random or pseudo-random game is played out from that point. Simulation policies that guide this game playout are relevant here since completely random playouts generally make for weaker players [21]. Finally, the win or loss is propagated back up the tree. Every node visited before the playout needs to be updated. When the simulations are finished or the budgeted time runs out, a node is selected as the action to play. The selection criteria are usually the most simulated node or the node with the highest value. This technique has been used successfully in many games, most notably Go [10]. Until the arrival of MCTS, Go programs were unable to compete with professional players. This changed when MCTS programs such as Fuego and MoGo dominated all classical programs and even defeated top rated professional players on small boards [7].

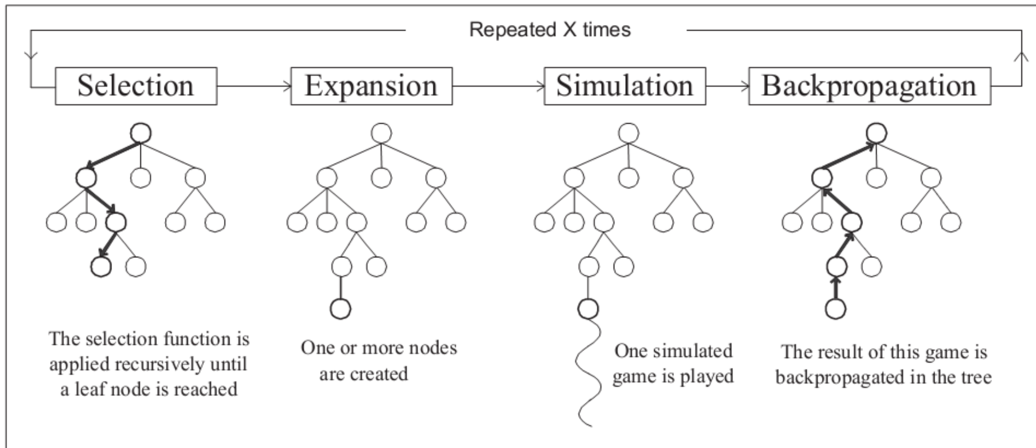

Figure 2.1: The Monte Carlo Tree Search process [3].

2.2 UCB and UCT

The basis for the experiments in this thesis is MCTS using the Upper Confidence Bounds applied to Trees (UCT) algorithm. UCT is an extension of the Upper Confidence Bounds (UCB) algorithm when it is applied to trees [18]. UCB provides a general method of balancing exploration and exploitation in the multi-armed bandit problem to achieve sub-linear regret [2]. The multi-armed bandit problem is where there is a number of slot machines, or bandits, and your goal is to play on those machines and maximize your profit. Said another way, you need to minimize your losses or regret, which is the total amount you have regretted picking the machines over all of your plays. So if the highest payoff machine is chosen, then the regret is zero and playing any other machines accumulates regret equal to the difference of their payoffs. Since the regret is sub-linear, as the number of plays goes to infinity, the average regret goes to zero. The UCT algorithm extends this method to trees and proves that sub-linear regret is maintained [18]. The selection method for UCT and UCB is the node or machine which maximizes the following formula at a given timestep:

$$\text{value}_j = \begin{cases} \bar{x}_j + C\sqrt{\frac{\ln(n)}{n_j}} & \text{if } n_j > 0 \\ FPU & \text{if } n_j = 0 \end{cases}$$

Here, $\bar{x}_j$ is the average reward from machine j, n_j is the number of times j has been played, and n is the total number of plays. C is the exploration constant. Greater values mean more exploration, which is $\sqrt{2}$ by default. The FPU, or First Play Urgency, value is determined by the first-play urgency policy and is ∞ by default. If the FPU value is ∞, then all nodes are selected once before any node is selected again. In some cases, this is undesirable when

you want to exploit promising nodes until they fall below a certain value, represented by the FPU. Extending this method to trees is simple in that the formula is repeatedly used to select children until a leaf is reached. After reaching a leaf, a node is expanded, and a simulation of the rest of the game is used to determine whether it was a win or a loss.

In pseudo-code, UCT looks like the following:

```
1  function performUCTSimulation(rootNode)
2    nodesVisited[0] := rootNode;
3    i := 0;
4    while (nodesVisited[i] is not leaf) do
5      nodesVisited[i + 1] := selectNode(nodesVisited[i]);
6      i := i + 1;
7    end while;
8    value := simulateGameFromNode(nodesVisited[i]);
9    for idx := 0 to i − 1 do
10     nodesVisited[idx].visits := nodesVisited[idx].visits + 1;
11     nodesVisited[idx].value := nodesVisited[idx].value + value;
12   end for;
13 end function;
14
15 function selectNode(parent)
16   if parent.visits = 0
17     return parent.children[randomInt(0, parent.children.size())];
18   end if;
19   bestValue := −1;
20   bestNode := null;
21   for i := 0 to parent.children.size() − 1 do
22     curNode = parent.children[i];
23     if curNode.visits = 0
24       value := FPU;
25     else
26       value := curNode.value/curNode.visits + C*sqrt(ln(parent.visits)/ln(
             curNode.visits));
27     end if;
28     if value > bestValue
29       bestNode := curNode;
30       bestValue := value;
31     end if;
32   end for;
33   return bestNode;
34 end function;
```

Listing 2.1: Pseudo-code for UCT on a game tree for one player.

After reaching a given number of simulations or after time runs out, a move must be selected. Common selection methods are selecting the node with the most simulations or selecting the node with the highest value. In order to measure the effectiveness of the algorithm, there

are three different metrics. The first is simply whether or not the best node was selected. Next is the cumulative regret of all simulations. That is the metric that UCT minimizes. Lastly, there is simple regret, which is the regret of the final move selection. All three of these metrics will be used in this thesis.

2.3 UCT in Games

Games where the UCT algorithm has been successfully used include Go, Arimaa, and Amazons. They each have different complexities that make them suited for using UCT. In Go, there is no good heuristic function known at this time. This has made alpha-beta players much weaker when compared to their MCTS counterparts. In terms of game complexity, on a 19 by 19 board, the average game length is about 250 moves with an average branching factor of more than 200 [1]. For Arimaa, there are good heuristic functions, but the game complexity makes it difficult to create an alpha-beta player that searches deeply. The average game length is 92 moves with an average branching factor of more than 17,000 [13]. Given the enormous branching factor, most computer players use a finely tuned MCTS in order to search deeper.

Amazons will be explained in more detail since it was used for the experiments described in Chapter 6. Standard Amazons is played on a 10 by 10 chess board. Each player has four queens that start on the edges of the board as shown in Figure 2.2. A player's move consists of moving a queen to an empty square, then firing an arrow to an empty square. Both moves can be any distance as long as they are a straight line orthogonally or diagonally. Queens and arrows cannot pass through or land on squares occupied by other queens or arrows. A player loses the game when there are no legal moves available. Strategically, the game is played as territory control since empty space that you control completely allows you to move in that area when you have no moves elsewhere. The average game length is 80 moves with an average branching factor of approximately 500 [14]. Unlike Go, however, the branching factor decreases exponentially rather than linearly. On the first move, there are 2176 possible moves [17]. This causes MCTS to have difficulty settling on moves at the beginning of the game.

2.4 Move Groups

Introducing move groups into tree searches for games is very easy to imagine. Simply take all of the available moves at a position, then create groups of moves randomly or based on some characteristic. An example of this is shown in Figure 2.3. Multiple levels of groups

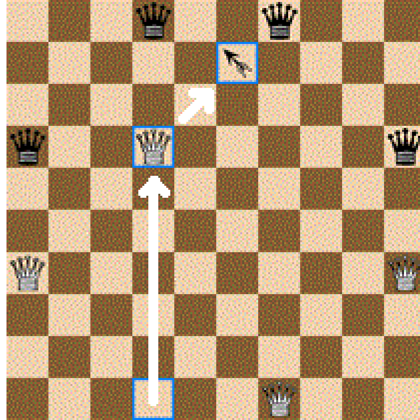

Figure 2.2: The starting board for Amazons after one move has been played.

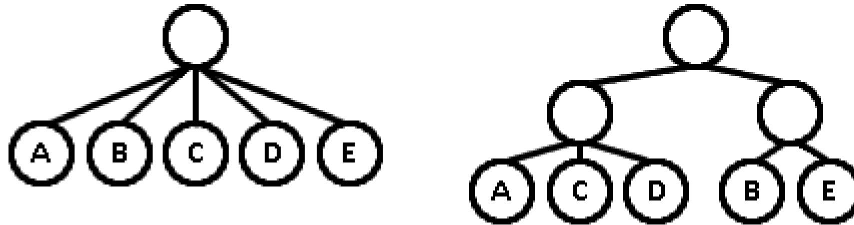

Figure 2.3: An example for move groups with five moves and two groups.

is also possible. As an example in Amazons, a possible grouping method is to group at the highest level by each distinct queen, then by each of that queen's moves, and finally by each of those moves' possible arrow shots. This is like breaking the move into stages where first a queen is selected, then that queen's destination square, and finally the arrow's destination square.

When the idea of move groups was applied to Go by Childs et al., they took an existing Go library, libEGO, and modified it to use move groups in the search [5]. Moves were grouped based on their Manhattan distance to the two previous moves. So there was a group consisting of moves exactly one space distant from the previous move, a group consisting of moves two spaces distant from the previous move, and so on. The same grouping was done again except with the distance being from the second previous move. Since each move was in both sets of groups, the groups were not disjoint. When playing against GnuGo, they saw a significant increase in playing strength over the unaltered library with the same time constraints. Saito et al. also applied move groups to Go [20]. They used three groups consisting of (1) moves within two spaces of the last move, (2) moves on the edge of the board, and (3) all other moves. With these groups, they observed an increase in performance

when playing against a version without groups. They also noted that the player seemed to shift focus between the groups when appropriate, *i.e.* the player stopped playing local moves when it was better to move somewhere else.

In Arimaa, move groups are essentially mandatory due to the large branching factor. A move in Arimaa actually consists of four partial moves. So, a player can move one piece four times, four pieces once each, or anything in between. Gerhard Trippen developed the Arimaa player Rat which is an alpha-beta player with a unique style move generation [22]. In Rat's case, move generation is restricted to certain categories such as attacks, retreats, and captures. Some moves are excluded entirely from move generation if they are not in a category. This process is repeated until all four moves for a turn have been evaluated. Tomáš Kozelek implemented a MCTS player for Arimaa called Akimot that uses a similar step based strategy for move generation [19]. In Akimot's case, all moves are considered, but duplicates played in a different order are considered to be the same move through use of a transposition table.

Finally, in the game of Amazons, most MCTS players use move groups. Richard Lorentz in his Amazons program Invader uses a two step move generation where first a queen destination square is chosen, then an arrow destination square [15]. Campya, another Amazons player, uses the same method [17]. Lorentz reports that after this change, the player was ten times faster due to the savings from generating moves. At the University of Alberta, we have an Amazons player, Arrow2, that uses the three step move generation previously mentioned. Experiments and results for our player are discussed in Chapter 6.

2.5 Other Enhancements

Rapid Action Value Estimation has been used in many Go programs to great success [11]. It offers bias towards moves being played again if they assisted in a win for a simulation at any time in the playout. This can be used in any game if that game allows for moves to be played out of order. This is most certainly the case in Go since stones can be placed in any order as long as the space is not occupied or otherwise illegal to place in. For games where this is not the case or where moves rely on other moves being played before them, such as moving a pawn twice in chess, it is possible to implement RAVE, but difficult and of lesser value.

Progressive unpruning is a technique developed by Chaslot et al. that has parallels to move groups [4]. Progressive unpruning begins its search with a subset of moves and includes more moves in the subset as time allows. The initial subset of moves is based on prior knowledge, game specific knowledge about the strength of different moves. This is similar

to a situation with move groups where all of the good moves start out grouped together with a number of visits and wins to encourage that group's selection. Then, as time allows, other moves outside the group are added to the good group. The issue in this case is that progressive unpruning requires a way to select the initial subset. This can be done by using prior knowledge in the form of heuristics or by using the value of nodes after a number of simulations.

CHAPTER 3

Experiments with Nine Nodes

3.1 Experimental Framework

The experimental framework developed needed to accomodate the goals as determined by the research questions. Therefore, a simple abstract game was created in order to simulate real game playing while still allowing detailed examination. The game operates as follows:

1. The entire game tree initially is the root node with N children.
2. Each child has a fixed probability, p, of paying off representing the probability of a given move in a real game tree of winning. Payoffs are constant: 1 with probability p and 0 with probability $1 - p$.
3. This flattened tree competes with trees that have move groups and therefore additional levels in the tree. No child is duplicated in the groups.
4. UCT is used to select the next node to simulate. The FPU used is ∞ which means that all nodes will be explored once before further simulations.

This simple artificial game allows for small changes to be analyzed in detail. It is, however, an approximation of MCTS. When a tree search is performed, the actual payoff probabilities of nodes in an actual game change over time as the tree is expanded. This aspect is simplified to be fixed probabilities in order to facilitate a more detailed analysis.

First of all, by having fixed probabilities for the payoffs, one distribution of payoffs can be completely analyzed. This first includes modifying the C value in order to determine what value performs the best with no groups. Then, when using groups, we can easily compare

the group performance to the base case with no groups. From that comparison, given a set of underlying payoff probabilities, we can say which groups perform better than the base case, and, among those groups, which groups performed the best. From that set of groups, we then can look at the underlying structure of the groups and generalize what makes those groups perform better.

In order to make sure all groups have been considered, we first start with a small number of nodes, nine to be exact, so that all possible groups can be considered. After testing all possible groups, we can conclusively determine exactly which ones perform best of all groups. This knowledge can then be extended to situations with more nodes where testing all possible groups is not feasible.

Since the underlying node probabilities are a large factor in the actual performance of groups, further experiments are carried out. By changing the payoff probability of the best node, we change the problem's difficulty, which is mostly determined by the difference in value between the best and second best node [2]. This allows for running the same analysis on problems of varying difficulty. Comparing the results for the best groups allows for more generalization of group structure for different problems.

Given these considerations, the experiments are performed in the following order. First is an easy problem where the best and second best node have a large difference in payoffs. Next is a hard problem where there is only a small difference between the two best nodes. Finally, the changes in group performance with small changes of problem difficulty are analyzed.

3.2 An Easy Problem

The initial experiment was measuring the performance of every possible combination of three groups of three nodes against the nine nodes without groups. The underlying payoffs used were $\{0.1, 0.2, 0.3, ..., 0.9\}$. An example of one possible group is shown in Figure 3.1. A single run of the experiment consisted of running 16,384 simulations. At simulation breakpoints of 2^n where $4 \leq n \leq 14$, performance was measured as if the run had stopped at that point. At each simulation breakpoint, the node selected is the one with the most simulations. This selection criteria was chosen since it is used in many MCTS players [15, 19]. All results were averaged over 5,000 runs. The performance metrics initially considered are selection rate of the best node, simple regret, and cumulative regret as described in Section 2.2. This experiment was repeated for different C values, consisting of 0.0, 0.1, 0.2, 0.5, 1.0, 2.0, 5.0, and 10.0, to determine how it influenced different groups.

Before going into the experiments, here is a short explanation of the performance metrics and their relation to actual games. Selection rate is similar to a situation where there may

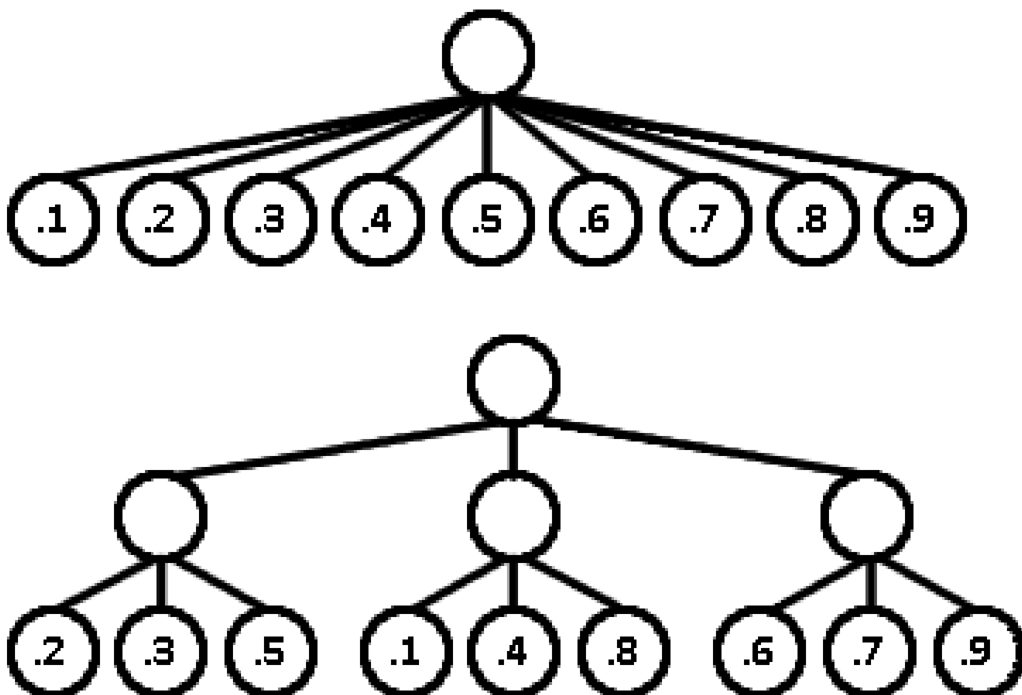

Figure 3.1: An example grouping of nine nodes. The base case tree is on top and the example grouping is below.

be many nodes that have a high value, but only one is a good move. Playing any move but that one would be a blunder. So selection rate penalizes any move that is not the best. Simple regret is similar to a situation where you only have to find a move that is close to optimal. Since there will only be a small amount of regret for choosing a close to optimal move, simple regret does not punish those choices as much. Cumulative regret is similar to a situation where the very act of simulating a move is costly. So cumulative regret punishes for any moves simulated that are not the best move.

For these experiments, it was noted that the groups tended to finish their simulations in approximately 80% of the time it took for the base case to do so. This can be attributed to the fact that the base case needed to determine the value, see line 26 of the UCT algorithm in Listing 2.1, of all nine nodes when choosing which node to simulate next. Groups only needed to find the value of six nodes, three for selecting which group to simulate and three for selecting which node in that group. This speed increase was seen in all following experiments as well.

The performance with 500 runs according to the selection rate of the best node is shown in Figure 3.2. For example, a 95% confidence interval for a selection rate of 0.5 is ± 0.0138 and ± 0.0122 for selection rates of 0.25 and 0.75. These confidence intervals hold for all further experiments unless otherwise noted. The best performing C values for the base case were 0.5 and 1.0, with 0.5 performing slightly better than 1.0 at simulation counts below 512 and slightly worse after that. To look at the effect of introducing move groups without changing any parameters, the performance of the base case was compared to all move groups with the same C value. The results are shown in Table 3.1. For very low simulation counts, move groups on average performed better. When the C value was low, up to the lowest optimal value of 0.5 for the base case, introducing move groups was beneficial except for a few select groups. For C values 1.0 and larger, introducing move groups on average saw a boost in performance. This seems to suggest that move groups fare better with higher C values.

When no longer fixing the C values for comparisons between the base case and groups, a different picture emerged. There were a total of 2240 group/C value pairs, from 280 groups and 8 C values. Of those pairs, there were 359 pairs that performed as well or better than the base case. The range of C values was from 0.5 to 5.0 inclusive with 1.0 and 2.0 being the most common. This was the initial place to begin looking for the best group structure. Three of these pairs selected at random that performed better than the base case are as follows:

- $C = 2.0$, { { 0.1, 0.2, 0.3 } { 0.5, 0.6, 0.7 } { 0.4, 0.8, 0.9 } }
- $C = 1.0$, { { 0.1, 0.5, 0.6 } { 0.2, 0.4, 0.7 } { 0.3, 0.8, 0.9 } }
- $C = 0.5$, { { 0.2, 0.3, 0.5 } { 0.1, 0.4, 0.8 } { 0.6, 0.7, 0.9 } }

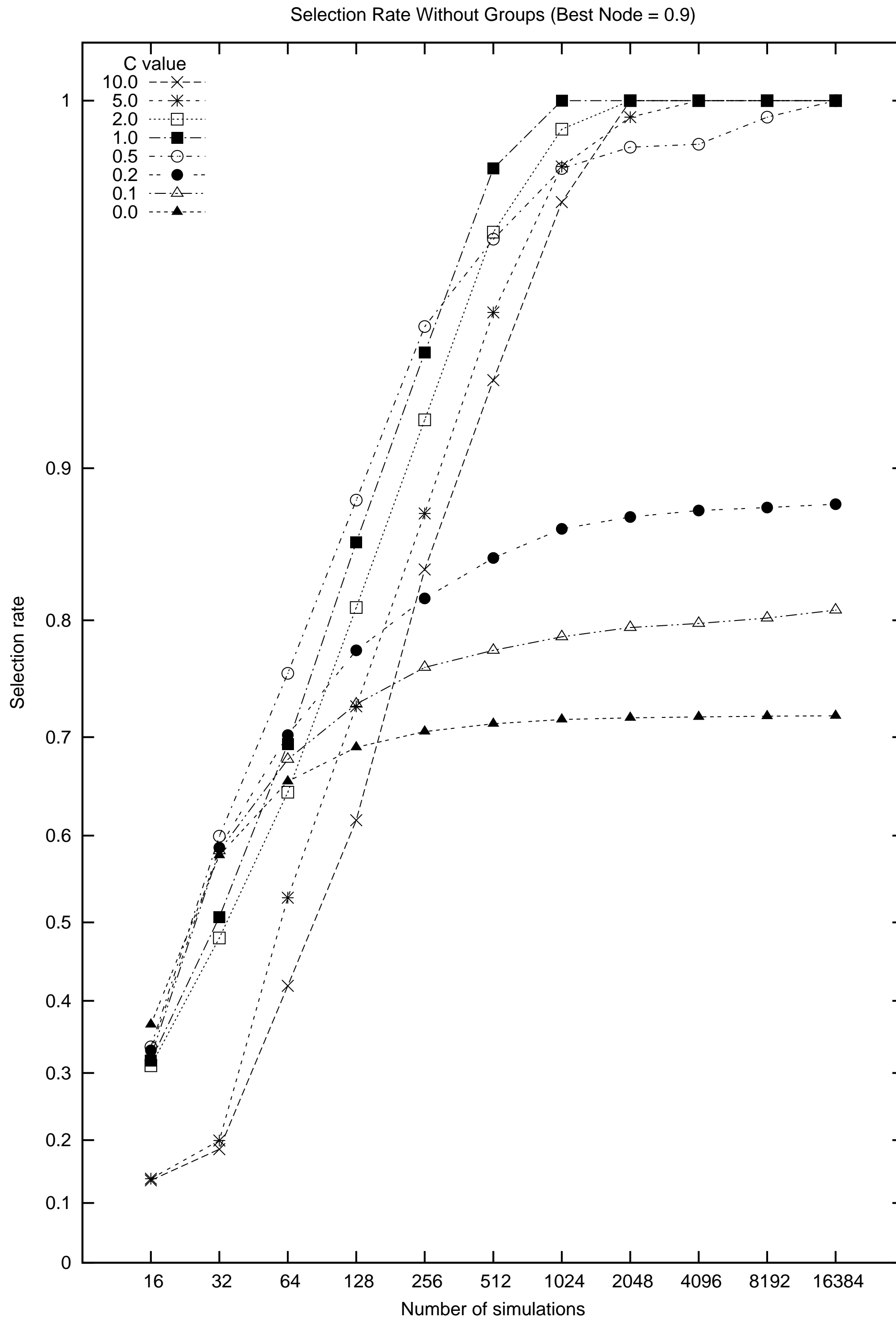

Figure 3.2: Base case performance, selection rate of the best node at various simulation counts with node payoffs ranging from 0.1 to 0.9.

C val	16	32	64	128	256	512	1024	2048	4096	8192	16384
0	47.1	0.3	0	0	0	0	0	0	0	0	0
0.1	67.8	0.7	0	0	0	0	0	0	0	0	0
0.2	65.7	2.5	0	0	0	0	0	0	0	0	0
0.5	74.2	21.4	5.0	2.1	0.3	1.0	0.3	1.0	1.43	2.86	**5.0**
1.0	75.7	70.3	62.1	61.0	62.1	28.5	**48.9**	**81.0**	**92.8**	**95.3**	**98.9**
2.0	57.8	54.2	55.0	56.9	61.4	66.0	81.0	**100**	**100**	**100**	**100**
5.0	68.3	87.1	55.3	51.4	53.2	54.6	57.5	70.3	**98.9**	**100**	**100**
10.0	68.9	73.5	50.0	55.0	49.2	52.1	50.7	**57.5**	**66.0**	**83.5**	**100**

Table 3.1: Percentage of all groups that performed equally or better than the base case with the same parameters. Bold indicates where the base case selected the best node for all runs.

In general, the best structure was found in groups that had the best node paired with the second or third best node. Also, the average value of nodes in the best group was greater than the average value of other groups. When analyzing groups on an individual basis, 1.0 was the best performing C value on average. However, the number of simulations it took for a C value of 1.0 to perform better than 0.5 was much lower than the base case, 32 versus 512. Finally, in an effort to map group structure to performance, the average value of groups has been plotted against the performance of that group after 128 simulations with a C value of 1.0 in Figure 3.3. The trend there is obvious: better performance corresponds to the group with the best node having higher average value than the next group. However, there needs to be a balance between the groups in order to have the best performance. With the three best nodes grouped together, the performance was about the same as the base case, as seen by the cluster of triangles on the right side of the figure. With less disparate group averages and still having the best node in the highest value group, the performance increased. This confirms the group structure previously found: groups with the best node and second or third best node performed the best. Changing the C value to 5.0 as in Figure 3.4 gave a similar picture, but with a wider variance of performances. For a few groups, the performance increased, but for many groups, it was worse than with $C = 1.0$.

The next performance metric is simple regret. Base case performance is nearly the same as when using best node selection rate as seen in Figure 3.5 in terms of the best performing breakpoint/C value pairs. The exception to this is at the 16 and 32 simulation breakpoints, where very low C values, 0.0 to 0.2, perform just as well as 0.5 and 1.0. This makes sense since low exploration means that the first good nodes seen are heavily exploited. On average, this will generate a good move given a short time frame.

When looking for the best group structure using this performance metric, the same groups are performing better than the base case as before. Even the relation between average group values and group performance is preserved as shown in Figure 3.6. If the best node is in

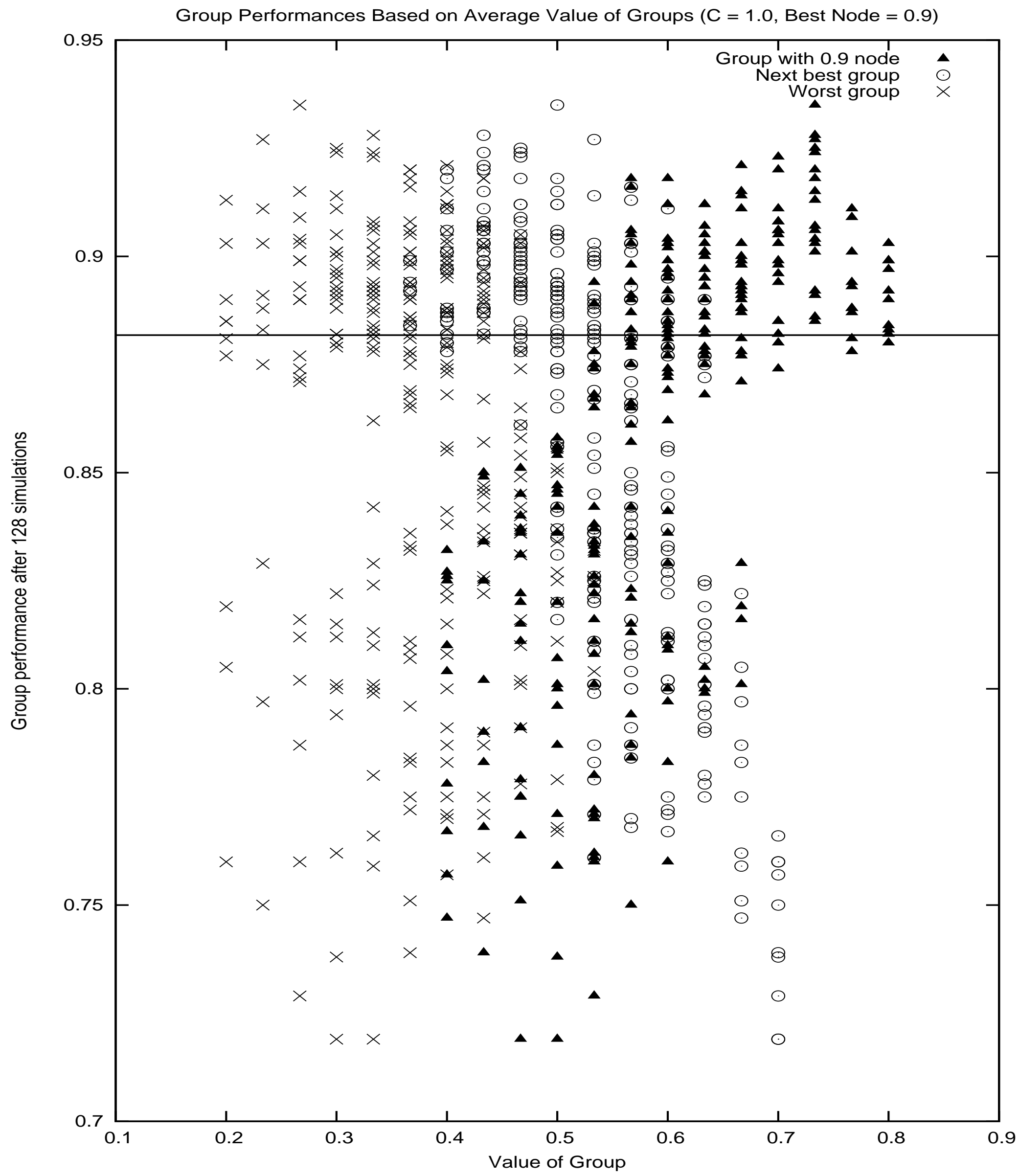

Figure 3.3: The average value of groups vs. performance in terms of selection rate after 128 simulations. Each group is one set of symbols. Node values range from 0.1 to 0.9 and C is 1. The horizontal line is the base case performance without groups.

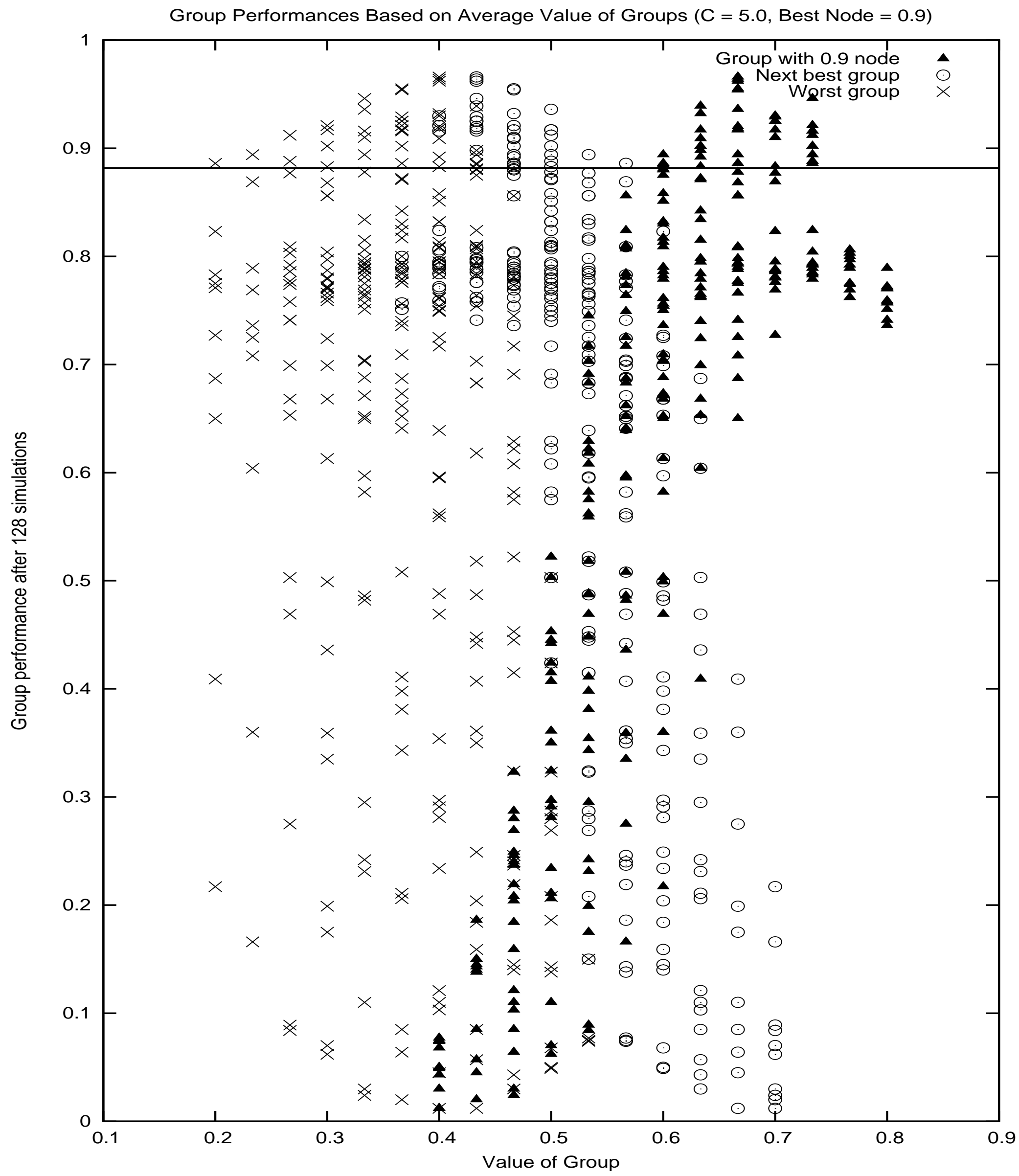

Figure 3.4: The average value of groups vs. performance in terms of selection rate after 128 simulations. Node values range from 0.1 to 0.9 and C is 5. The horizontal line is the base case performance without groups.

a group with a higher average value than the next group, then the group has less simple regret.

The last metric, cumulative regret, was found to be not as useful as the other two metrics. After 512 simulations, the cumulative regret for all groups and the base case is shown in Figure 3.7. There is not very much difference between the range of regret for groups and the base case at the same C value. Also, even though groups with C values of 1.0 and higher are performing better than the base case in terms of selection rate, a C value of 0.5 still has a much lower cumulative regret. This can be attributed to the fact that exploring more often results in finding the best node faster, but wastes simulations on nodes that are not the best node. The relative cumulative regrets stay the same even with more simulations as seen in Figure 3.8.

To summarize, with an easy problem of distinguishing the best node with a value 0.1 greater than the second best, there are groups that perform better than no groups at all. These groups were found to have the best node grouped with other high value nodes, typically the second or third best node. This caused the search to favor that high value group for more simulations and then the search only had to distinguish between the three nodes in that group. In terms of metrics, selection rate and simple regret were found to be useful for distinguishing good groups. Cumulative regret was found to be not useful since it measured performance of all simulations rather than the end result. Having analyzed this set of node payoff probabilities in detail, the next experiment analyzes the groups again with a different set of payoffs to look for any changes in group structure.

3.3 A Hard Problem

Given in the easy problem of Section 3.2 that the majority of groups and even the base case reached 100% selection rate quickly within 1024 simulations, a comparatively hard problem was chosen as the next experiment. The experiment was the same in every way as the previous one, with the exception that the best node now had a payoff probability of 0.81 instead of 0.9. So, the payoff probabilities were $\{ 0.1, 0.2, 0.3, ..., 0.8, 0.81 \}$. Since the values of the best node and second best node were much closer, finding the best move was much harder.

The base case's selection rate performance is shown in Figure 3.9. The optimal C values are the same as before, with 0.5 starting off better but later being surpassed by 1.0. Overall, the performance decreased when compared to the previous experiment.

Group performance was a lot more varied this time around. Most of the groups that performed better than the base case for the last experiment did so again. However, there was

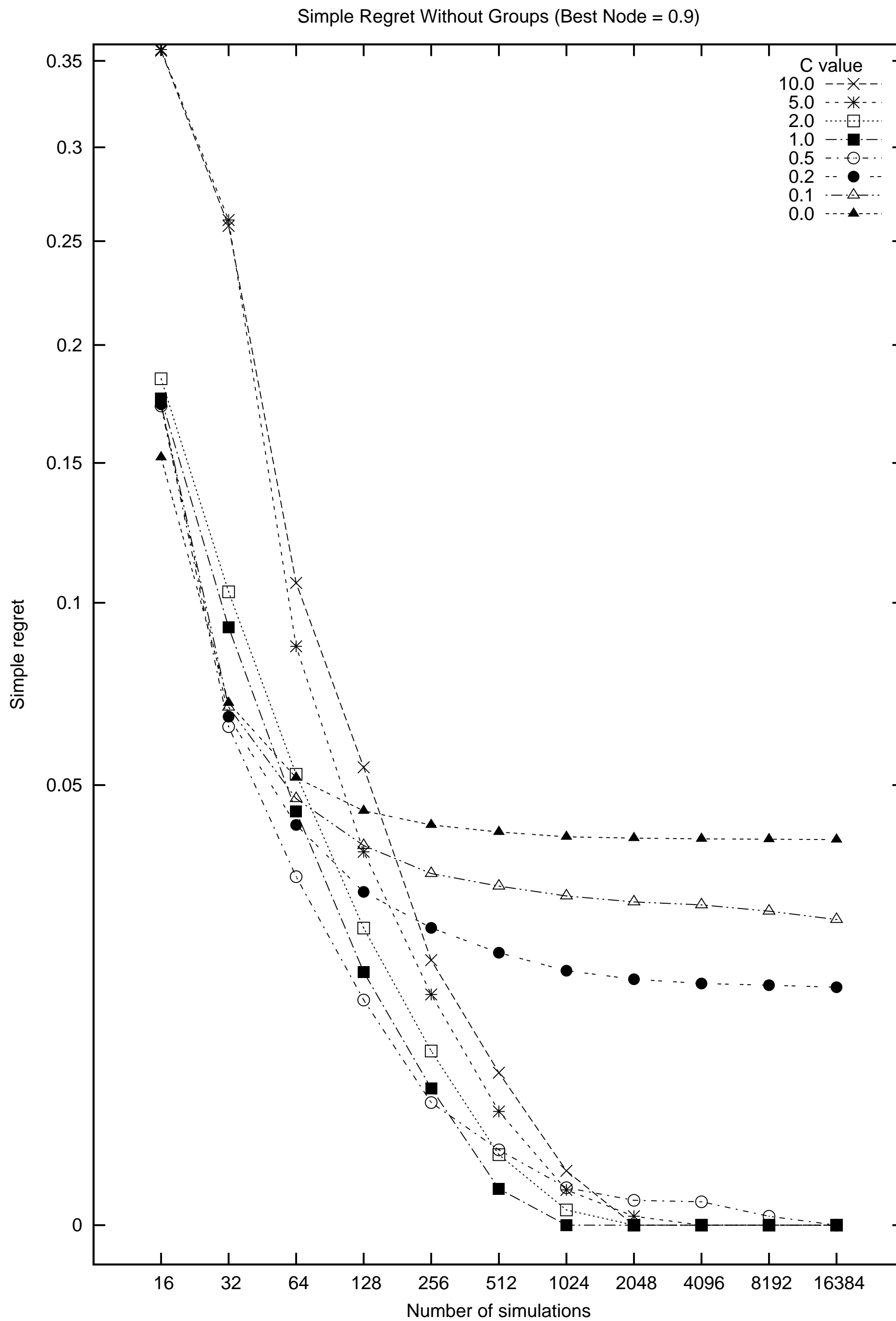

Figure 3.5: Base case performance with no groups, simple regret at various simulation counts with node payoffs ranging from 0.1 to 0.9. Less regret is better performance.

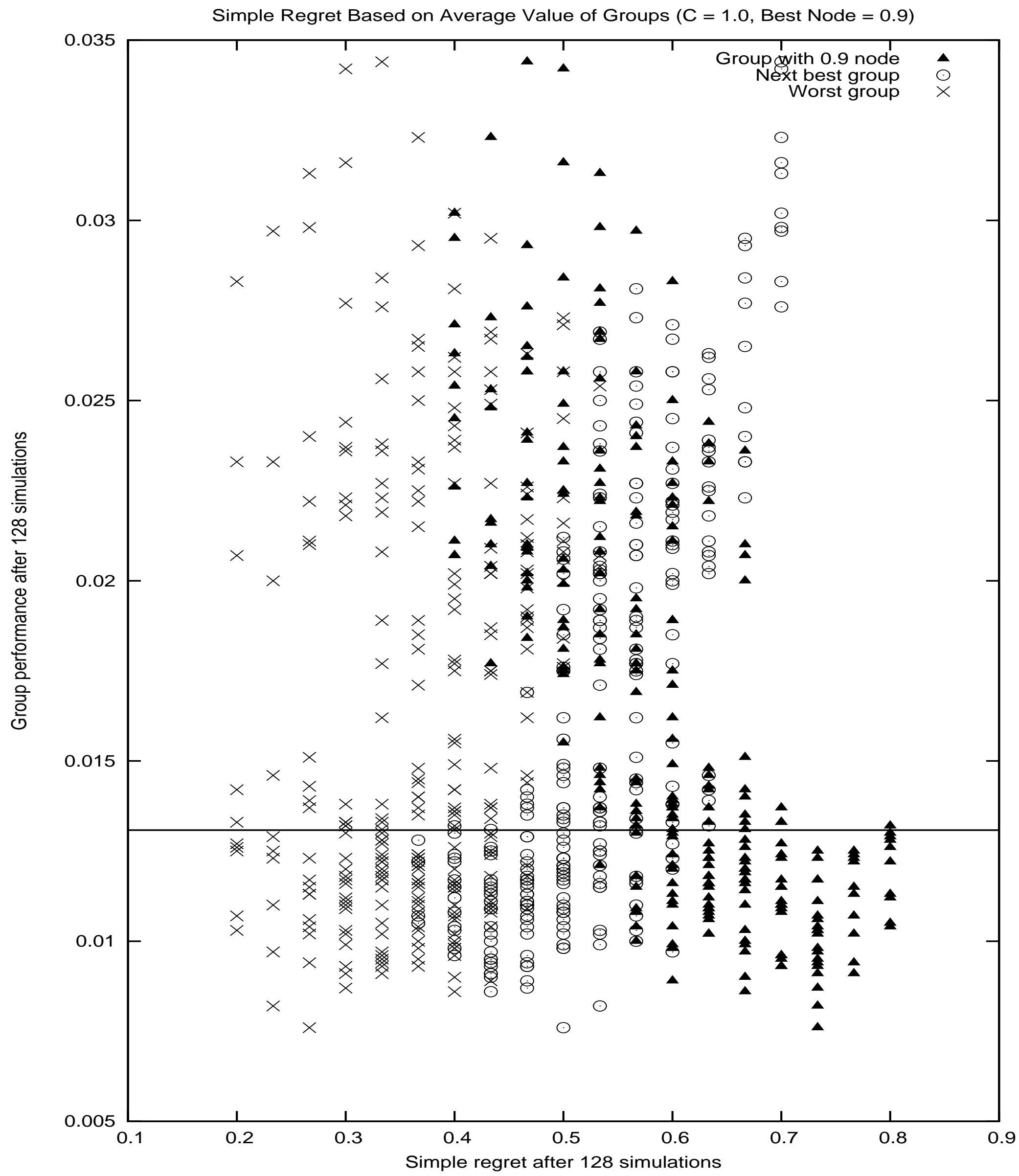

Figure 3.6: The average value of groups vs. performance in terms of simple regret after 128 simulations. Node values range from 0.1 to 0.9 and C is 1. The horizontal line is the base case performance without groups. Less regret is better performance.

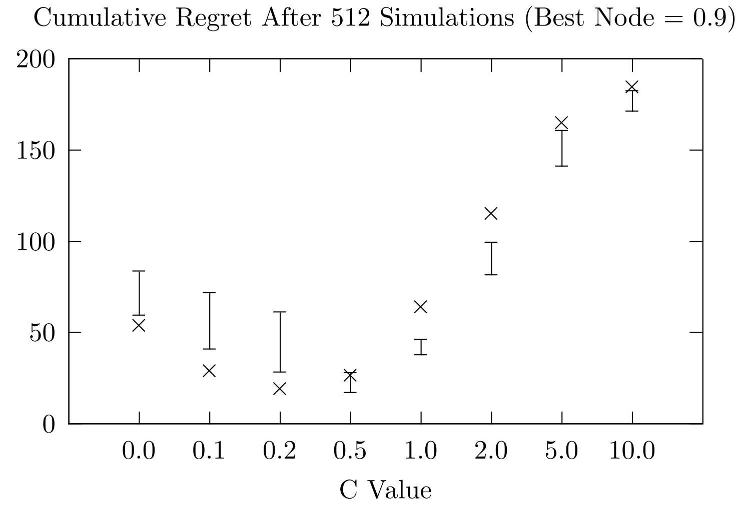

Figure 3.7: Cumulative regret of groups, range denoted by bars, and the base case, denoted by an x, after 512 simulations with payoff probabilities ranging from 0.1 to 0.9.

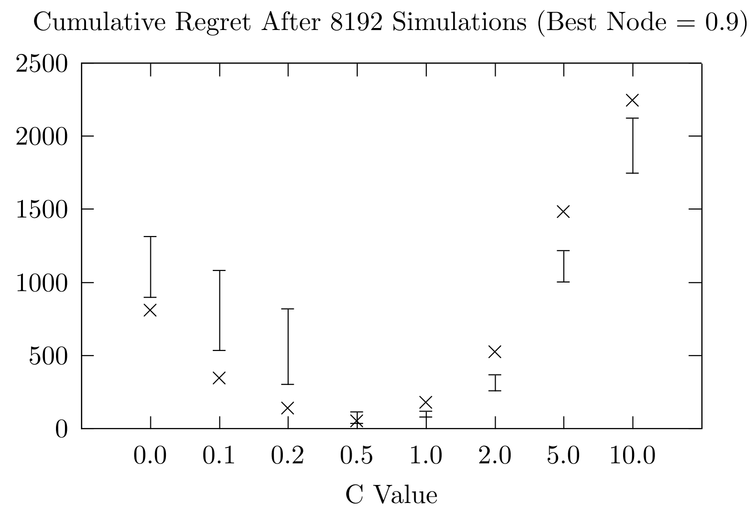

Figure 3.8: Cumulative regret of groups, range denoted by bars, and the base case, denoted by an x, after 8192 simulations with payoff probabilities ranging from 0.1 to 0.9.

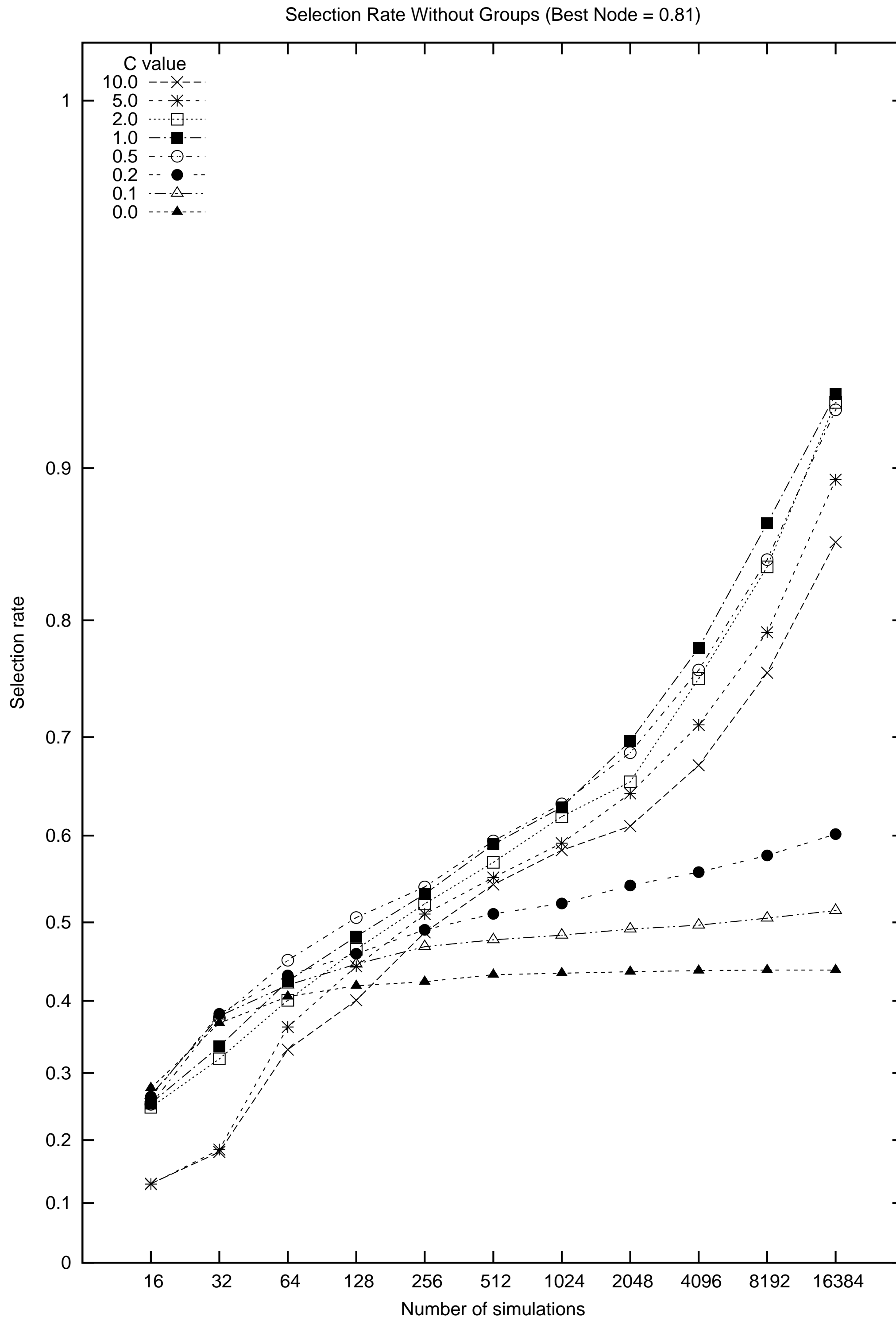

Figure 3.9: Base case performance, selection rate of the best node at various simulation counts with node payoffs ranging from 0.1 to 0.81.

some unexpected behavior. First, groups with exceptionally high C values of 5 or 10 did incredibly well in some cases. As an example, the group of {{ 0.1, 0.4, 0.8 } { 0.2, 0.3, 0.7 } { 0.5, 0.6, 0.81 }} with a C value of 10 started off with a selection rate of 0.2068 after 16 simulations but achieved a 100% selection rate after 1024 simulations. However, there were cases of groups experiencing a "switching problem" after a large number of simulations. Usually, this was the case when the best node and second best node were in the same group. During the simulations, that group was singled out very quickly since it had a high average value compared to the other two groups. However, due to the close values of the best and second best nodes, they accrued similar numbers of simulations. So when the time came to select a node, on average, both nodes had similar numbers of simulations. Therefore, the selection rate of the best node decreased due to there not being enough difference in their values. A more reliable structure was sought after due to this problem. The most reliable structure found was one that had { 0.6, 0.7, 0.81 } or { 0.5, 0.6, 0.81 } as a group. This group avoided the switching problem since the second best node was not in the same group as the best node. Also, the best node's group was heavily favored due to the fact that it had two other high value nodes in the group.

In general, similar structures as in the previous experiment performed well. After 128 simulations, there was no switching problem yet and we got Figure 3.10 and 3.11. Both are the groups average values and performance after 128 simulations, with Figure 3.10 having $C = 1$ and Figure 3.11 having $C = 5$. The distribution is very similar to before, even when changing the C value. The only noticeable difference is in terms of the structure of the absolute best groups. With $C = 1$, the best groups had the best node in a group with average value of about 0.7. Changing the C value to 5 increased group performance when the best group had an average value of about 0.63. The change in performance this time was more pronounced. As an example, the group consisting of {{ 0.1, 0.4, 0.7 } { 0.2, 0.3, 0.8 } { 0.5, 0.6, 0.81 }} increased from a selection rate of 0.6056 at $C = 1$ to 0.846 at $C = 5$. The change in performance ranged from -0.4072 to +0.2458 when changing C from 1 to 5.

To summarize the results on group structures, the best performing group structures occurred when the target node was in a group that had higher average value compared to other groups. The part of the structure that was most important was that the best node was in a group with other high value nodes. At the extreme of that, grouping the best nodes together in a single group achieved similar performance to the base case performance with no groups. Increasing the value of the other two groups while leaving the best node's group with the highest average value had the best performance. Increasing the C value beyond what is ideal for basic UCT widened the range of performance. In the case of good group structures, performance increased; in the case of bad group structures, performance decreased greatly. This effect was more pronounced when the problem became more difficult, where more than a 0.2 selection rate increase was observed when increasing the C value.

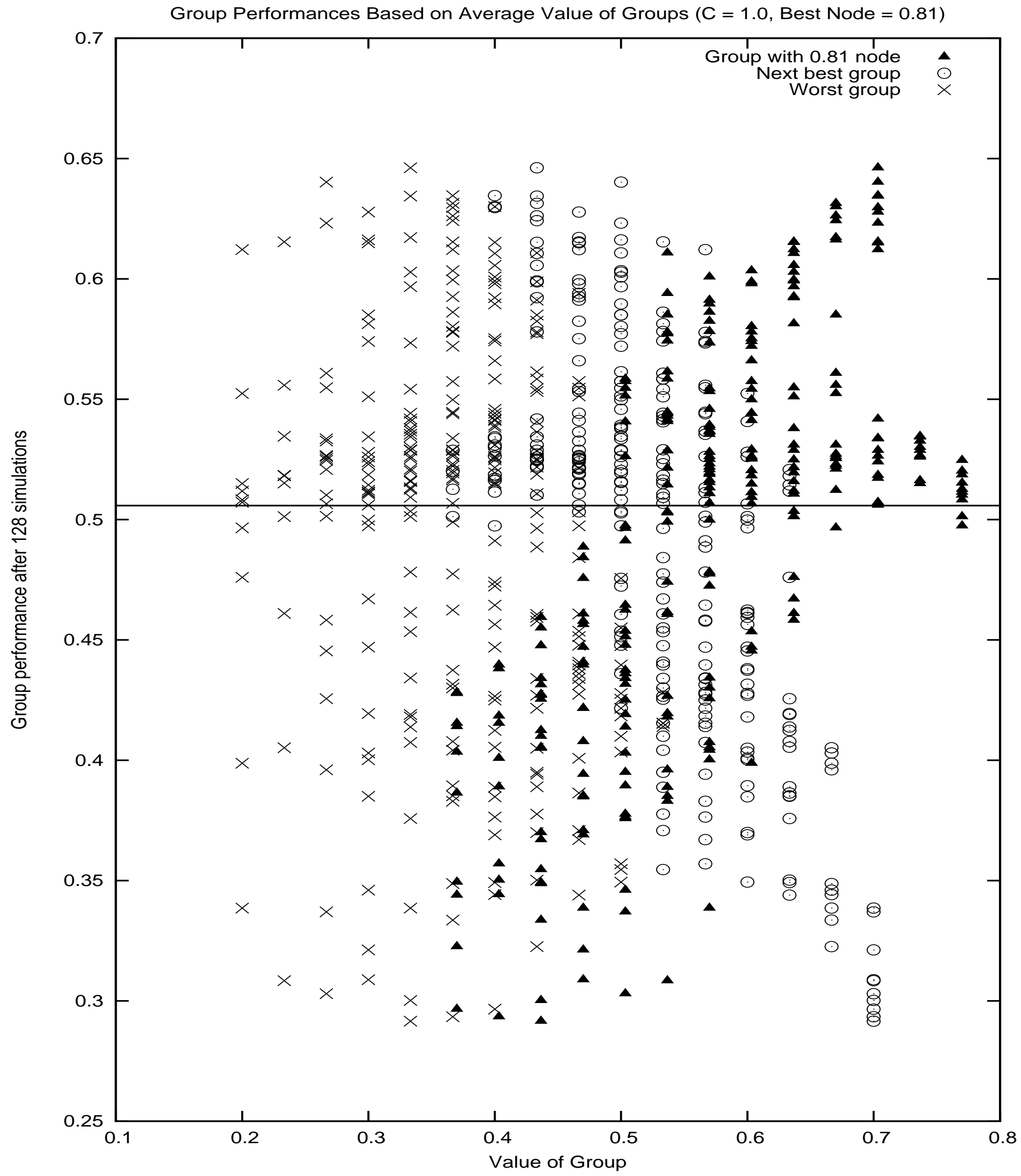

Figure 3.10: The average value of groups vs. performance in terms of simple regret after 128 simulations. Node values range from 0.1 to 0.81 and C is 1. The horizontal line is the base case performance without groups.

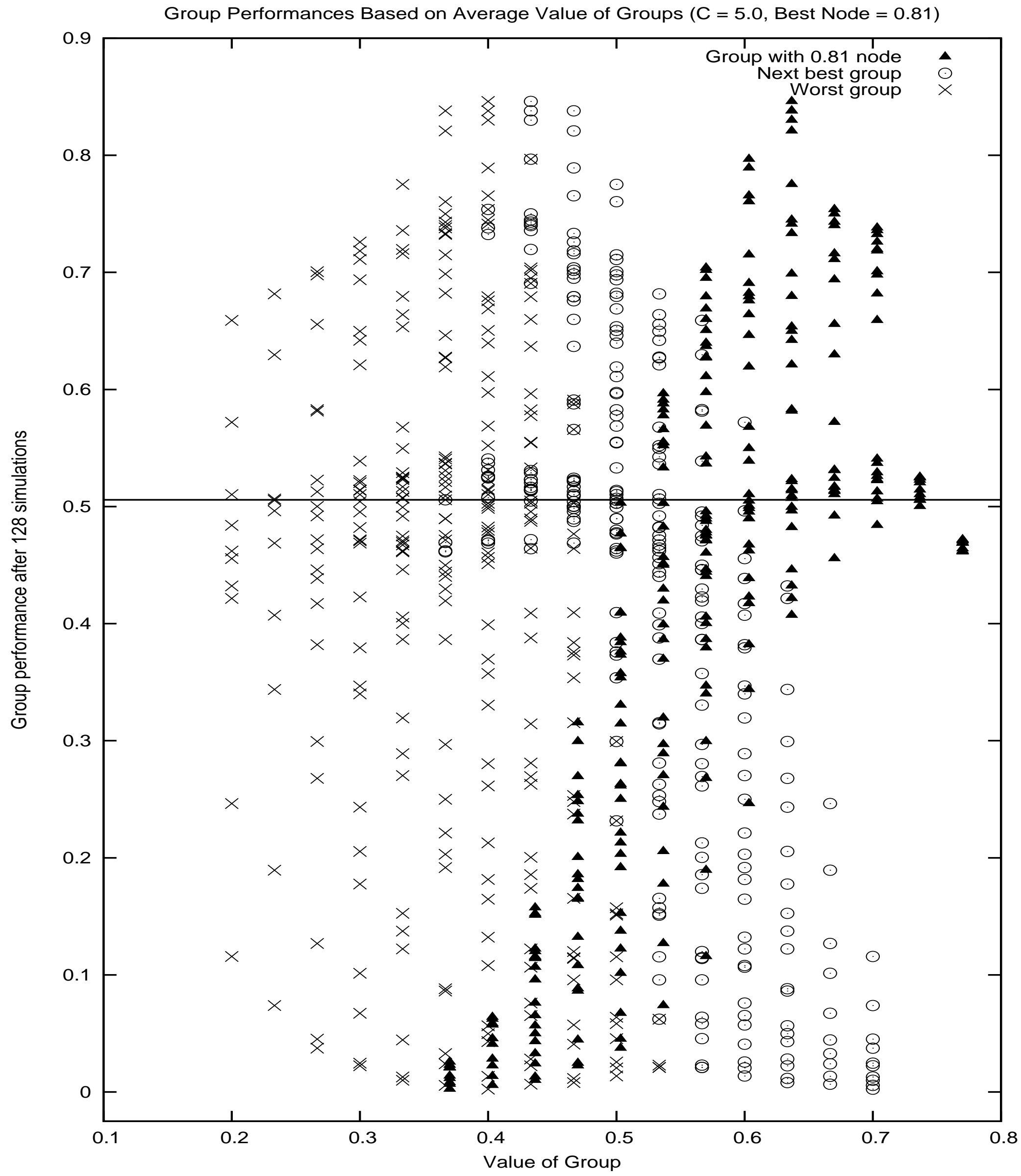

Figure 3.11: The average value of groups vs. performance in terms of simple regret after 128 simulations. Node values range from 0.1 to 0.81 and C is 5. The horizontal line is the base case performance without groups.

3.4 Group Selection Rate Performance with Changing Difficulty

In order to get an idea of how problem difficulty and C value affected performance, one group that performed well overall was selected to be analyzed further. The group was {{ 0.2, 0.3, 0.7 } { 0.1, 0.4, 0.8 } { 0.5, 0.6, 0.81 }}. First, Figure 3.12 shows the group's performance with $C = 1$ over the number of simulations with multiple lines for different problem difficulties. Figure 3.13 shows the same but with $C = 5$. It is important to note how both values for C have issues with differentiating the best node from the second best node, *i.e.* the switching problem, as the problem difficulty increases. With a hard problem and a C value of 1, the performance decreases around 1024 simulations due to the difficulty of distinguishing between the two best nodes. With a C value of 5, this happens much later in the simulations with a more pronounced effect. In the case of the best node having a value of 0.81, the performance drops as seen in Figure 3.13.

The groups that handled the switching problem best were those that had the best node with other high value nodes and the other nodes split between the other two groups equally in terms of value. However, recreating this exact structure in an actual game would prove to be difficult. So in order to determine what group structures should be mimicked, the next chapter explores creating structures without exact information about the underlying node payoffs.

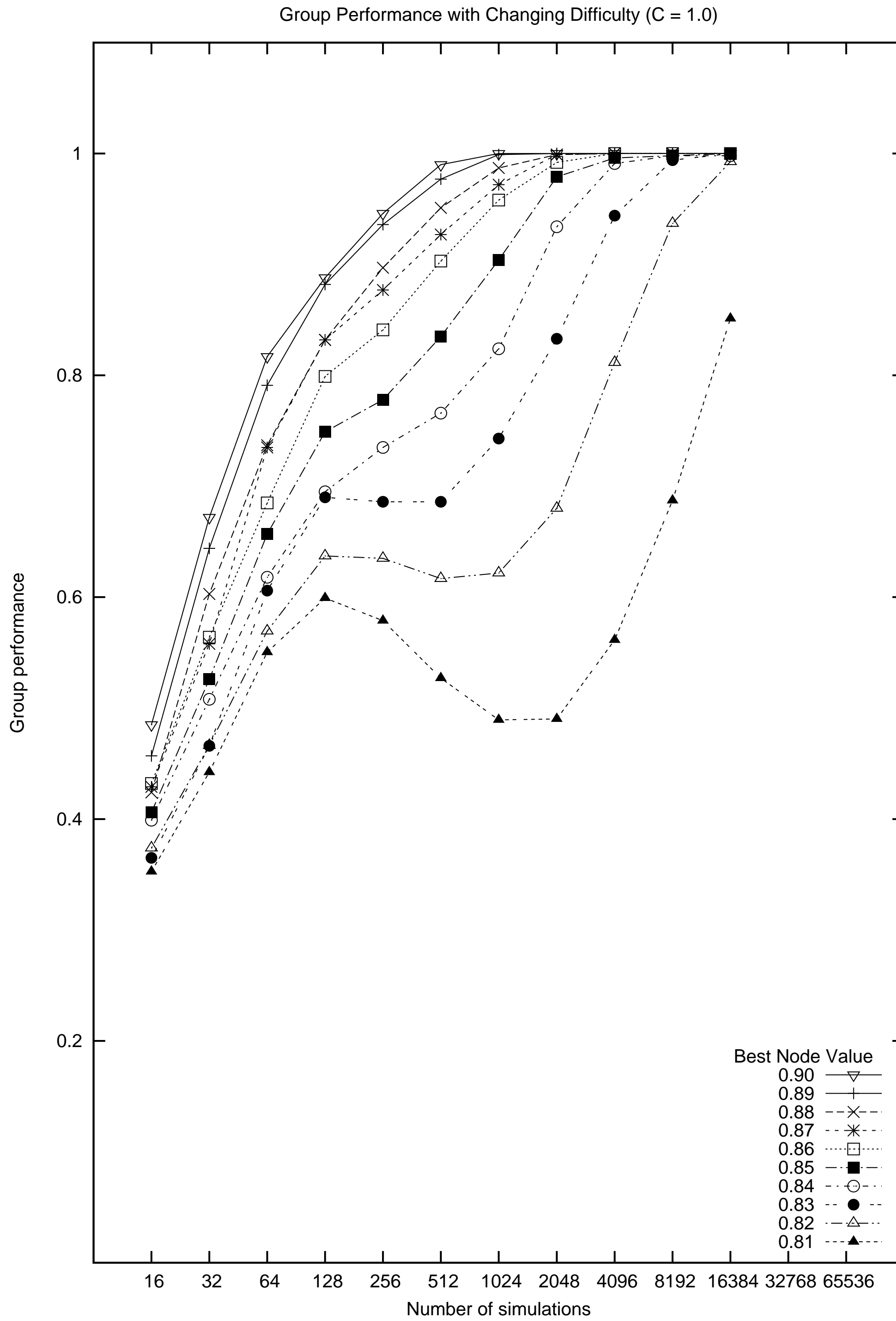

Figure 3.12: The performance of a single group vs. the number of simulations with $C = 1$. Multiple lines show different problem difficulties. The group is {{ 0.2, 0.3, 0.7 } { 0.1, 0.4, 0.8 } { 0.5, 0.6, 0.81 }}.

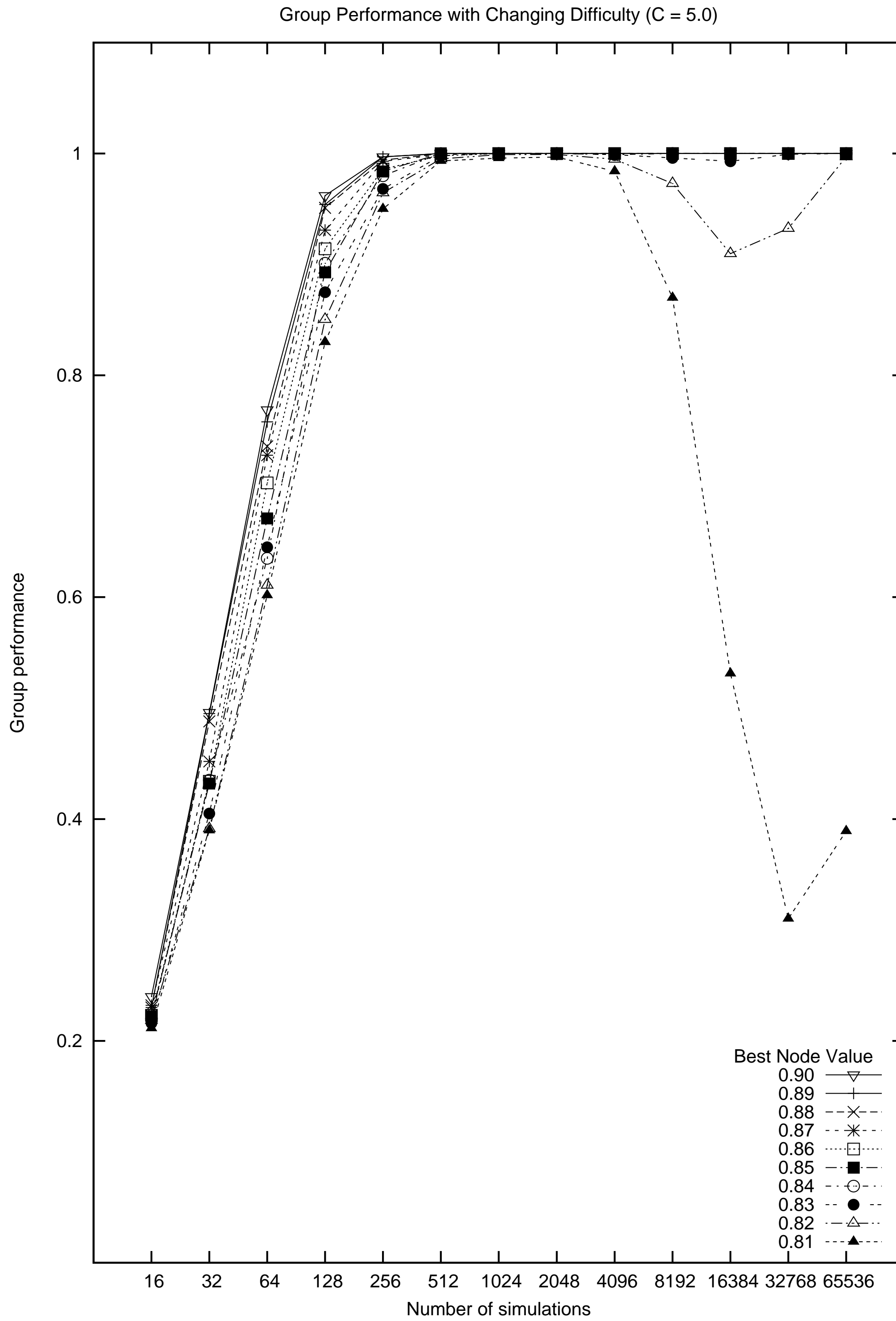

Figure 3.13: The performance of a single group vs. the number of simulations with $C = 5$. Multiple lines show different problem difficulties. The group is {{ 0.2, 0.3, 0.7 } { 0.1, 0.4, 0.8 } { 0.5, 0.6, 0.81 }}.

CHAPTER 4

Using Move Groups with Knowledge

As shown in the previous chapter, we can create a grouping of nodes that performs better than without any groups if we know the exact value of the nodes. However, the exact value of nodes is unknown in a real game. So, the application of the structures found requires some additional experimentation on a more realistic problem. Since the structures rely on the relative value of the nodes, we need a way of calculating the value of the nodes without knowing their true value. Two methods of doing this are described and experimented with in this chapter. The first method uses the value of the nodes after a number of simulations. The second method is to use game specific knowledge in the form of a heuristic function to approximate the value of nodes. It should be noted that the first method can be used even if there is no heuristic function available.

By testing both of these methods with the group structures found, we can determine the effect of groups with imperfect knowledge. Two group structures will be tested. These structures were selected on the basis of selection rate of the best node. In regards to the other performance criteria, simple regret had the same best performing structures as selection rate and cumulative regret does not measure relevant performance to the problem. Thus, only selection rate was the chosen metric for these experiments. The first group structure used was grouping the three best nodes together, referred to later as the 123 structure. This structure had a selection rate of 88.3% to 90.3% after 128 simulations when the best node's payoff probability was 0.9. For comparison, the base case with no groups had a selection rate of 88.2%. The second group structure is grouping the best node with the third and fourth best nodes, referred to later as the 134 structure. This structure had a selection rate of 91.3% to 93.5% with the same parameters as for the other structure. The first structure is more reliable since it only requires knowing which three nodes are better than the rest. The second structure is more difficult since the second best node must be

separated from the group or the performance decreases.

4.1 Creating Groups in an Easy Problem

The first experiment for creating groups without heuristic knowledge was based off of the easy problem introduced in Section 3.2. All parameters are the same except groups are not initially present; they are only created after a number of simulations. Three sets of experiments were done with groups created once after 64, 128, or 512 simulations. Another three sets of experiments were with groups created after 50, 100, or 200 simulations and recreated every 50, 100, or 200 simulations after that. The last experiment tested groups that were created and recreated every 2^n simulations starting at 16. Creating the groups is a simple process. Simply take the current mean value of the nodes based on their previous simulations and create a group that matches the target structure. Only the main group matters, the 123 or 134 group, due to the small range of performance from those structures regardless of how the other nodes were arranged. These ranges are detailed at the beginning of this chapter. For these experiments, the target structure was created, then the other 6 nodes were randomly arranged into two groups of 3.

Starting with the 123 structure, the results in terms of selection rate for the seven grouping methods along with the base case with no groups are listed in Table 4.1. Significant differences were determined by a 99% confidence interval around the base performance. With 5000 trials, these intervals are $\{ .6 \pm 0.0069, .747 \pm .0061, .88 \pm .0046, .957 \pm .0029, .996 \pm .0009, 1 \pm 0 \}$. The after 64 and every 50 methods generally performed worse than the base case. The after 128, every 100, every 200, and every 2^n methods started off performing better then performed as well as the base case except for a decrease in performance at 512 or 1024 simulations. The after 512 method matched performance of the base case.

The experiment was repeated using the 134 structure as the target. The results for this experiment are listed in Table 4.2. This time the only significant increase in performance was for the 2^n method at 32 simulations. Besides that, all methods generally performed significantly worse until 1024 simulations, except the 2^n method which performed worse until 2048 simulations. So for an easy problem, creating the 123 structure had results with comparable performance as without groups, with the added bonus of a speed increase for tree traversal. All grouping methods performed fairly close to the base case with the exception of the after 64 method. The 134 structure did not have as good performance as the 123 structure.

Type	32	64	128	256	512	1024	2048	4096	8192	16384
Base	.600	.747	.880	.957	.996	1.00	1.00	1.00	1.00	1.00
After 64	–	–	*.871*	.957	*.992*	1.00	1.00	1.00	1.00	1.00
After 128	–	–	–	**.968**	.995	*.999*	1.00	1.00	1.00	1.00
After 512	–	–	–	–	–	1.00	1.00	1.00	1.00	1.00
Every 50	–	.747	.877	.958	.995	*.999*	1.00	1.00	1.00	1.00
Every 100	–	–	**.886**	**.969**	.995	1.00	1.00	1.00	1.00	1.00
Every 200	–	–	–	**.961**	.996	*.999*	1.00	1.00	1.00	1.00
Every 2^n	**.618**	.746	.863	.954	*.993*	*.999*	1.00	1.00	1.00	1.00

Table 4.1: Performance of different grouping methods with target structure 123 and best node 0.9. Dashes indicate points where the groups were not yet created. Bold indicates a significant increase in performance and italics indicates a significant decrease.

Type	32	64	128	256	512	1024	2048	4096	8192	16384
Base	.594	.747	.874	.961	.996	1.00	1.00	1.00	1.00	1.00
After 64	–	–	*.851*	*.951*	*.991*	1.00	1.00	1.00	1.00	1.00
After 128	–	–	–	*.950*	.995	1.00	1.00	1.00	1.00	1.00
After 512	–	–	–	–	–	1.00	1.00	1.00	1.00	1.00
Every 50	–	.746	*.841*	*.949*	*.992*	1.00	1.00	1.00	1.00	1.00
Every 100	–	–	.872	*.943*	*.994*	1.00	1.00	1.00	1.00	1.00
Every 200	–	–	–	*.952*	*.992*	1.00	1.00	1.00	1.00	1.00
Every 2^n	**.611**	*.711*	*.834*	*.949*	*.991*	*.999*	1.00	1.00	1.00	1.00

Table 4.2: Performance of different grouping methods with target structure 134 and best node 0.9. Dashes indicate points where the groups were not yet created. Bold indicates a significant increase in performance and italics indicates a significant decrease.

Type	32	64	128	256	512	1024	2048	4096	8192	16384
Base	.385	.460	.511	.552	.594	.643	.703	.775	.865	.945
After 64	–	–	.512	.553	**.604**	.639	.697	.771	*.843*	*.925*
After 128	–	–	–	**.562**	.596	*.630*	.699	*.760*	*.852*	*.936*
After 512	–	–	–	–	–	.644	*.694*	**.782**	.865	*.940*
Every 50	–	.465	.504	.558	.596	*.626*	.697	*.764*	*.853*	*.938*
Every 100	–	–	*.502*	.546	*.585*	*.631*	.697	.775	*.856*	.942
Every 200	–	–	–	.555	.594	*.631*	*.693*	.770	*.857*	*.939*
Every 2^n	.388	*.446*	*.500*	*.543*	.592	.638	.701	.774	*.859*	*.941*

Table 4.3: Performance of different grouping methods with target structure 123 and best node 0.81. Dashes indicate points where the groups were not yet created. Bold indicates a significant increase in performance and italics indicates a significant decrease.

4.2 Creating Groups in a Hard Problem

For the next experiment, we repeated the previous experiment but with the hard problem described in Section 3.3. The results for the 123 structure on this problem are shown in Table 4.3. A 99% confidence interval for the base case is $\{$ $.385 \pm .0069$, $.460 \pm .007$, $.511 \pm .0071$, $.552 \pm .007$, $.594 \pm .0069$, $.643 \pm .0068$, $.703 \pm .0065$, $.775 \pm .0059$, $.865 \pm .0048$, $.945 \pm .0032$ $\}$. Overall, performance for the grouping methods on the 123 structure shifted back and forth between no significant performance change and a significant performance decrease. Only the after n methods had a significant performance increase at any point. Using the 134 group structure as the target gave the results shown in Table 4.4. For this experiment, all grouping methods performed significantly worse after 2048 simulations. The only significant performance increases occurred for the every 50, every 100, and every 2^n methods at the first simulation breakpoint after their introduction.

In general, the 123 structure performed much better than the 134 structure especially at simulation counts 2048 and higher. This can be attributed to the difference between the target group and the group actually created. A 134 structure performed very well in general, but a 234 structure, which could be easily created by chance for the hard problem, had poorer performance. Among the grouping methods themselves, the differences were minor. Choosing a grouping method therefore depends on how it will be used. As an example, the every 50 method performed consistently well with only a few performance drops. If grouping as soon as possible is desired, then the every 2^n method performed better at low simulation counts, but had more performance drops at higher simulations.

Type	32	64	128	256	512	1024	2048	4096	8192	16384
Base	.390	.450	.501	.548	.585	.634	.695	.774	.857	.942
After 64	–	–	*.488*	*.537*	.578	.633	*.680*	*.738*	*.818*	*.905*
After 128	–	–	–	.545	.585	.631	*.677*	*.727*	*.796*	*.888*
After 512	–	–	–	–	–	*.626*	*.678*	*.716*	*.789*	*.880*
Every 50	–	**.460**	.493	.548	.592	*.626*	*.671*	*.709*	*.799*	*.895*
Every 100	–	–	**.511**	.549	.590	.632	*.679*	*.708*	*.791*	*.900*
Every 200	–	–	–	.546	.583	.628	*.668*	*.703*	*.799*	*.899*
Every 2^n	**.435**	.452	.495	.543	*.572*	*.620*	*.675*	*.710*	*.786*	*.870*

Table 4.4: Performance of different grouping methods with target structure 134 and best node 0.81. Dashes indicate points where the groups were not yet created. Bold indicates a significant increase in performance and italics indicates a significant decrease.

4.3 Creating Groups in an Easy Problem with Prior Knowledge

The next set of experiments explored using prior knowledge in the form of a heuristic function to create groups. The heuristic function used was the true value of the node plus a uniform random value in $[-N, N]$ which represents noise. Five values were used for the range of this threshold, $N \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$. The experiments were structured as follows. First, a heuristic value for each of the nodes was used to rank them relative to each other. Then, given this ranking, a target group structure was created. The target structures were the same as in the previous experiment, the 123 and 134 structures. Finally, perform the simulations, and track selection rate of the best node at the same simulation breakpoints used in the other experiments. This was repeated 5,000 times for each value of N.

Results for the easy problem with the best node valued at 0.9 for the 123 structure are in Table 4.5, and the results for the 134 structure are in Table 4.6. A 99% confidence interval was used again and is similar to the one used in Section 4.1. For both target structures there is a significant increase in performance at 16 simulations. Then, for the 123 structure, for $N \leq 0.3$, performance is equal to or significantly better than the base case. For other N, performance is mostly significantly worse until 1024 simulations. For the 134 structure, performance is worse overall with only $N = 0.1$ performing as well or better than the base case for the large majority of simulations. With $N \geq 0.2$, the groups performed worst for the majority of simulation breakpoints between 32 and 512.

The results here show that even a relatively inaccurate heuristic function can be used to match nodes to a group structure and achieve good performance. The groups outperform the no group base case with small simulation counts or low noise. However, again, these

Type	16	32	64	128	256	512	1024	2048	4096	8192	16384
Base	.368	.599	.757	.882	.962	.997	1.00	1.00	1.00	1.00	1.00
$N = .1$	**.443**	.605	**.764**	.885	**.968**	.997	1.00	1.00	1.00	1.00	1.00
$N = .2$	**.454**	**.615**	**.765**	**.891**	**.971**	.998	1.00	1.00	1.00	1.00	1.00
$N = .3$	**.434**	.605	.763	.883	**.972**	.997	1.00	1.00	1.00	1.00	1.00
$N = .4$	**.423**	*.577*	*.725*	*.870*	.959	.996	1.00	1.00	1.00	1.00	1.00
$N = .5$	**.410**	*.584*	*.737*	*.876*	**.966**	*.995*	1.00	1.00	1.00	1.00	1.00

Table 4.5: Performance of using prior knowledge with target structure 123 and best node 0.9. Bold indicates a significant increase in performance and italics indicates a significant decrease.

Type	16	32	64	128	256	512	1024	2048	4096	8192	16384
Base	.368	.599	.757	.882	.962	.997	1.00	1.00	1.00	1.00	1.00
$N = .1$	**.441**	.604	**.772**	**.894**	**.966**	*.991*	1.00	1.00	1.00	1.00	1.00
$N = .2$	**.417**	*.567*	*.725*	*.872*	.959	*.993*	1.00	1.00	1.00	1.00	1.00
$N = .3$	**.408**	*.558*	*.740*	*.872*	**.966**	*.993*	1.00	1.00	1.00	1.00	1.00
$N = .4$	**.396**	*.562*	*.720*	*.865*	*.954*	*.991*	1.00	1.00	1.00	1.00	1.00
$N = .5$	**.401**	*.550*	*.733*	*.872*	*.957*	*.994*	1.00	1.00	1.00	1.00	1.00

Table 4.6: Performance of using prior knowledge with target structure 134 and best node 0.9. Bold indicates a significant increase in performance and italics indicates a significant decrease.

results do not consider the speed increase from using groups in the tree. There were significant differences between the target structures in these experiments with the 123 structure performing much better than the 134 structure in many cases such as the 128 and 256 simulation breakpoints.

4.4 Creating Groups in a Hard Problem with Prior Knowledge

The experiments from the previous section were repeated except with the best node having a value of 0.81. The results for the 123 structure are in Table 4.7. Again, a 99% confidence interval was used and is similar to the one used in Section 4.2. For this problem, at 16 simulations, all values of N performed better than the base case. From 32 to 1024 simulations, the groups generally performed as well as or better than the base case. For greater number of simulations, performance decreased starting with high noise heuristic functions. At 2048 simulations with $N \in \{0.2, 0.4, 0.5\}$, 4096 simulations with $N \geq 0.2$, 8192 simulations with $N \geq 0.1$, and 16384 simulations with $N \geq 0.2$, performance decreased relative to the base case with no groups. Higher noise was correlated with worse performance at any simulation

Type	16	32	64	128	256	512	1024	2048	4096	8192	16384
Base	.278	.383	.453	.506	.542	.594	.634	.696	.778	.868	.936
$N = .1$	**.352**	**.411**	.452	.510	**.560**	.590	.641	.698	.772	*.860*	**.944**
$N = .2$	**.332**	**.395**	**.461**	.512	**.553**	.589	.630	*.683*	*.764*	*.845*	*.931*
$N = .3$	**.314**	.390	.451	.505	.547	.597	.640	.694	*.755*	*.841*	*.923*
$N = .4$	**.318**	.381	.446	.512	.543	*.582*	.629	*.685*	*.749*	*.831*	*.916*
$N = .5$	**.308**	.377	*.444*	.502	**.553**	*.585*	.631	*.674*	*.742*	*.830*	*.916*

Table 4.7: Performance of using prior knowledge with the target structure being 123 and the best node being 0.81. Bold indicates a significant increase in performance and italics indicates a significant decrease.

Type	16	32	64	128	256	512	1024	2048	4096	8192	16384
Base	.278	.383	.453	.506	.542	.594	.634	.696	.778	.868	.936
$N = .1$	**.308**	.377	*.445*	*.493*	.542	*.583*	*.611*	*.678*	*.708*	*.762*	*.901*
$N = .2$	**.307**	.387	.453	*.496*	.541	*.585*	.633	.692	*.735*	*.811*	*.892*
$N = .3$	**.308**	*.372*	*.443*	.502	**.551**	.596	.637	*.683*	*.744*	*.811*	*.899*
$N = .4$	**.300**	*.373*	*.445*	*.494*	.547	*.586*	*.618*	*.676*	*.745*	*.825*	*.914*
$N = .5$	**.297**	*.374*	*.444*	*.497*	.541	*.584*	*.620*	*.677*	*.746*	*.832*	*.918*

Table 4.8: Performance of using prior knowledge with target structure 134 and best node 0.81. Bold indicates a significant increase in performance and italics indicates a significant decrease.

count.

The results for the 134 structure are in Table 4.8. Overall, performance was equal to or significantly worse than the base case, except with 16 simulations or 256 simulations and $N = 0.3$. At no point did the 134 structure perform significantly better than the 123 structure and there are many cases where it performed significantly worse. As an example, for $N = 0.1$ at 1024 and above, the 134 structur ealways performed worse than the 123 structure.

These results agree with the results previously found regarding the relative strength of the 123 structure and 134 structure. Both are usable, but the 123 structure is more reliable since there is additional room for error. There is no difference in performance between a 123 and a 213 structure, but there is a large difference between a 134 and a 234 structure. Given that using even a very noisy heuristic function to group was able to achieve performance close to without groups promises that only a loose ordering of nodes is needed. There are performance boosts in terms of tree traversal to be had if one can order the nodes for simply grouping the best nodes together.

CHAPTER 5

Experiments with Twenty Seven Nodes

The previous chapters focused on examining the effects of groups and how to use them with a small number of nodes. However, there is much more possible variety in group structure than simply N groups of N nodes. This chapter examines different types of groups: many groups of few nodes, few groups of many nodes, and additional levels of groups. To be exact, with 27 nodes, the group formats tested are 9 groups of 3 nodes each, 3 groups of 9 nodes, and 3 groups of 3 groups of 3 nodes. This additional variety of groups will allow for connections to be made between the conclusions made for the simple 9 node case and this more diverse 27 node case. Exhaustive groups could not be tested this time since the number of possible groups is very large.

The values used for the nodes was { 0.12, 0.15, 0.18, ..., 0.84, 0.87, 0.9 }. Compared to the previous experiments, this is a hard problem because there is only a 0.03 difference in value between the best node and the second best node, and there are three times as many nodes to search. For each category of groups, 100 were randomly generated. Then each of these groups were simulated along with a base case with no groups for 65,536 simulations. Simulation breakpoints were used again, this time checking performance in terms of best node selection rate at 2^n for $5 \leq n \leq 16$. The set of C values tested is the same as before, { 0.0, 0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0 }. These simulations were repeated 5,000 times. The performance of the base case with no groups is shown in Figure 5.1. Compared to the 9 node experiments in Chapter 3, this is more difficult than the hard problem in Section 3.3 simply due to the number of nodes. Performance is roughly the same as the hard problem if given 2.5 times as many simulations.

A selection of 5 groups for each group format are in Table 5.1 and will be referenced in the later sections. The node arrangements for the groups are detailed below. The numbers used to identify the nodes correspond to their value with 0 being the worst node 0.12 and

26 being the best node 0.9. The best four nodes are bolded.

1. $C = 1.0$, { 10, 17, 20 } { 16, **25**, **26** } { 1, 15, 18 } { 5, 13, **23** } { 0, 4, 14 } { 7, 11, **24** } { 12, 19, 21 } { 2, 8, 9 } { 3, 6, 22 }
2. $C = 1.0$, { 8, 10, **24** } { 0, 4, 18 } { 6, 9, 22 } { 5, **23**, **26** } { 13, 19, 20 } { 2, 14, 17 } { 12, 16, 21 } { 1, 3, 7 } { 11, 15, **25** }
3. $C = 10.0$, { 10, 12, **25** } { 13, 14, **23** } { 17, 19, 21 } { 0, 8, 16 } { 1, 3, 20 } { 15, **24**, **26** } { 4, 9, 18 } { 5, 6, 11 } { 2, 7, 22 }
4. $C = 5.0$, { 2, 3, **25** } { 10, 13, 21 } { 12, 16, **24** } { 17, 18, 22 } { 4, 14, 20 } { 1, 5, 7 } { 6, 8, 9 } { 11, 15, **26** } { 0, 19, **23** }
5. $C = 5.0$, { 2, 7, 16 } { 1, 10, **24** } { 12, 15, 22 } { 3, 8, **26** } { 13, 17, 20 } { 5, 6, **25** } { 18, 19, 21 } { 9, 11, **23** } { 0, 4, 14 }
6. $C = 1.0$, { 0, 5, 6, 9, 11, 12, 15, 18, 22 } { 2, 10, 13, 16, 17, 19, **24**, **25**, **26** } { 1, 3, 4, 7, 8, 14, 20, 21, **23** }
7. $C = 1.0$, { 3, 4, 5, 7, 9, 15, 18, 21, **26** } { 0, 1, 13, 14, 16, 17, 19, 20, 22 } { 2, 6, 8, 10, 11, 12, **23**, **24**, **25** }
8. $C = 2.0$, { 0, 2, 3, 7, 11, 12, 13, 15, 17 } { 1, 4, 6, 9, 19, 21, 22, **24**, **26** } { 5, 8, 10, 14, 16, 18, 20, **23**, **25** }
9. $C = 0.5$, { 3, 4, 5, 7, 9, 15, 18, 21, **26** } { 0, 1, 13, 14, 16, 17, 19, 20, 22 } { 2, 6, 8, 10, 11, 12, **23**, **24**, **25** }
10. $C = 2.0$, { 1, 5, 14, 16, 21, 22, **23**, **24**, **25** } { 2, 3, 6, 10, 12, 13, 15, 18, 20 } { 0, 4, 7, 8, 9, 11, 17, 19, **26** }
11. $C = 1.0$, {{ 6, 9, 10 } { 7, 8, 12 } { 2, 13, 17 }} {{ 3, 15, **23** } { **24**, **25**, **26** } { 0, 4, 21 }} {{ 1, 14, 18 } { 11, 16, 22 } { 5, 19, 20 }}
12. $C = 2.0$, {{ 9, 17, **25** } { 3, 4, 18 } { 0, 14, **23** }} {{ 7, 12, **24** } { 5, 20, 21 } { 19, 22, **26** }} {{ 1, 13, 16 } { 2, 10, 11 } { 6, 8, 15 }}
13. $C = 1.0$, {{ 3, 17, **26** } { 5, 18, **23** } { 2, 16, 19 }} {{ 8, 10, 15 } { 9, 13, **24** } { 4, 11, 14 }} {{ 0, 12, 21 } { 7, 20, 22 } { 1, 6, **25** }}
14. $C = 0.5$, {{ 5, 22, **25** } { 4, 8, 12 } { 6, 16, 19 }} {{ 7, 20, **23** } { 0, 3, 14 } { 1, 9, 18 }} {{ 2, 15, **24** } { 10, 11, 17 } { 13, 21, **26** }}
15. $C = 5.0$, {{ 1, 2, **26** } { 5, 7, 8 } { 4, 12, **24** }} {{ 9, 19, 21 } { 3, **23**, **25** } { 11, 16, 20 }} {{ 0, 6, 13 } { 10, 14, 17 } { 15, 18, 22 }}

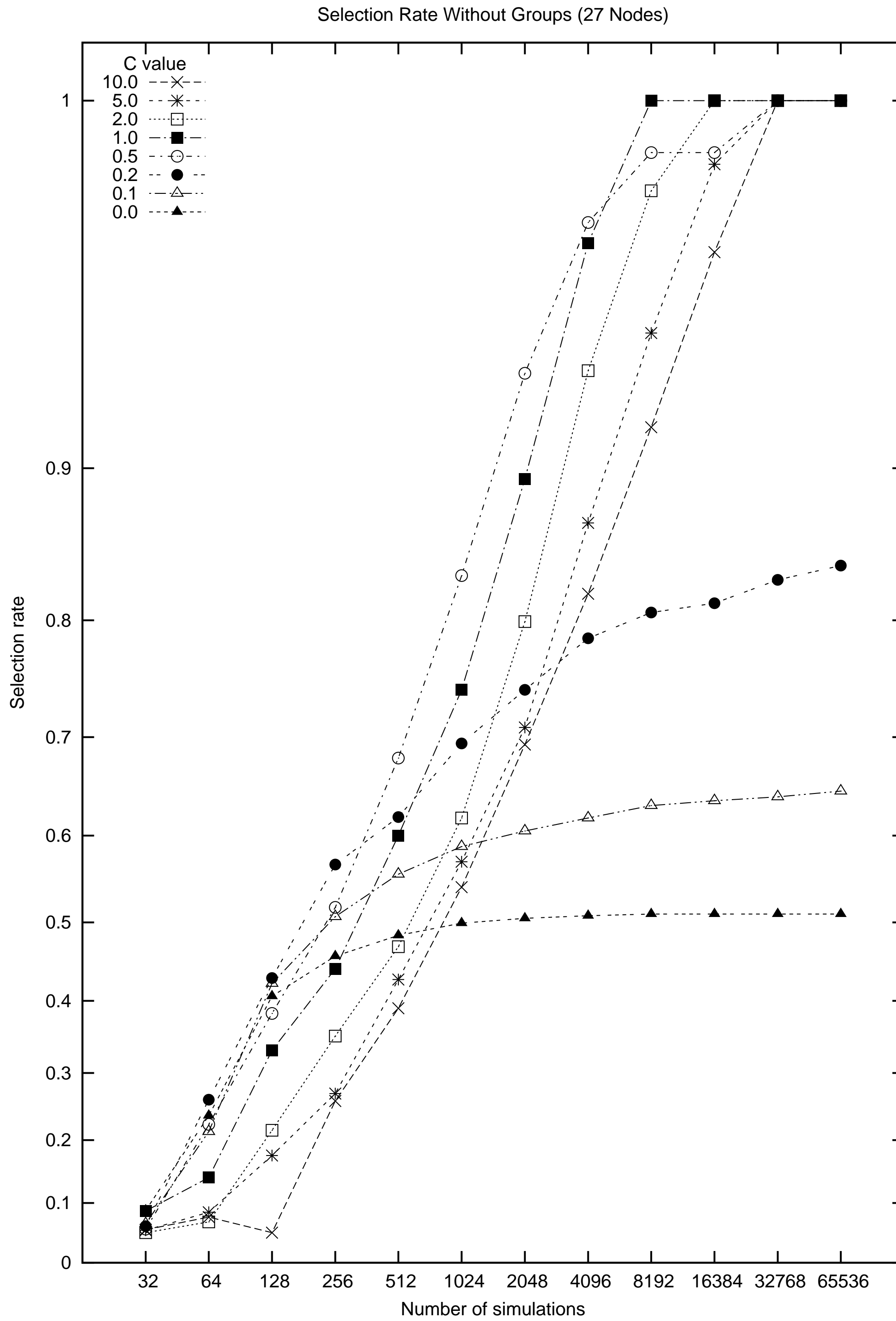

Figure 5.1: Base case performance without groups for 27 nodes.

ID	32	64	128	256	512	1024	2048	4096	8192	16384	32768
Base	.088	.261	.430	.568	.680	.833	.945	.989	1.00	1.00	1.00
1	.175	.299	.447	.589	.710	.847	.940	.996	1.00	1.00	1.00
2	.093	.259	.441	.662	.806	.902	.943	.986	1.00	1.00	1.00
3	.064	.174	.269	.389	.590	.761	.901	.965	.999	1.00	1.00
4	.061	.090	.151	.212	.272	.300	.478	.827	.997	.987	1.00
5	.048	.042	.037	.014	.008	.010	.027	.255	.977	.999	1.00
6	.144	.284	.437	.564	.718	.853	.957	.992	1.00	1.00	1.00
7	.118	.274	.415	.582	.737	.873	.955	.999	.999	1.00	1.00
8	.078	.135	.224	.350	.576	.826	.968	.998	1.00	1.00	1.00
9	.050	.124	.187	.236	.297	.349	.427	.515	.582	.655	.739
10	.036	.043	.026	.005	.000	.000	.000	.008	.176	.563	.834
11	.227	.334	.469	.615	.741	.864	.949	.997	1.00	1.00	1.00
12	.143	.300	.515	.774	.928	.985	.988	.993	.982	.993	1.00
13	.103	.242	.375	.520	.634	.741	.809	.864	.918	.952	.979
14	.218	.325	.396	.455	.494	.541	.604	.636	.670	.695	.740
15	.021	.005	.004	.000	.000	.000	.000	.000	.042	.998	1.00

Table 5.1: Performance of selected groups with 27 nodes. Group details are listed in Chapter 5.

5.1 Many Groups of Few Nodes

The first group category to be examined is 9 groups of 3 nodes. Structure of the groups that performed well is the focus since it can be compared to the structures found for the 9 nodes case in Chapter 3. There did exist groups that performed better than the base case. An example is the structure of group ID 1. The best node was grouped with the second best node and the 11th best node. That group therefore had a much higher average value than all other groups. In terms of actual performance, the group was significantly better than the base case until 2048 simulations where the performance evened out.

In terms of structure of the groups that did well, all of the groups that performed significantly better than the base case had the best node grouped with the second, third, or fourth best node. The third node in that group did not matter as much. Group IDs 2 and 3 illustrate this, however, group 3 had a C value that was too high and decreased performance. Beyond the group containing the best node, performance only decreased significantly when there was a group that had higher average value than the main group. This structure is very similar to that of the 9 node case; the best node group has a high average value due to other high value nodes in it. So the grouping strategy remains to be grouping the high value nodes together.

The other groups selected, group IDs 4 and 5, did poorly for two simple reasons. First, their

C values were too high; the same groups with a lower C value performed better. Second, the best node is not grouped with any other high value nodes making it difficult to distinguish the best node group at the root level.

5.2 Few Groups of Many Nodes

The next group category is 3 groups of 9 nodes. Again, there are groups that perform significantly better than the base case. When comparing the best groups in this category, *e.g.* group IDs 6 and 7, to the previous category's best groups, the performance is about equal. The significant differences are minor, sometimes favoring one group category for lower simulation counts and the other for higher simulation counts. Even using a C value of 1 for the best groups remains the same. Group ID 7 is an example of this with the best node, second best node, and fourth best node in a single group performing very well with when $C = 1.0$.

Looking at the common structure between the best groups, an obvious pattern emerges. All of the best performing groups have at least three of the five best nodes, including the best node, in their group. This is similar to the previous structure where there is a clear bias in the best performing structures to put many of the best nodes in the same group in order to encourage more simulations of that group. In regards to why group IDs 8, 9, and 10 performed poorer, there are similar reasons as for the other group category. Group ID 8 split up the best four nodes among two groups making it more difficult to distinguish those two groups. Group IDs 9 and 10 had the second, third, and fourth best nodes in a group other than the best node group making it difficult to find the best node in the first place.

5.3 Multiple Levels of Groups

The last group category is 3 groups of 3 groups of 3 nodes. This category can be seen as a hybrid of the previous two. With many groups of few nodes, adding a level breaks down the "many" groups into another layer of groups. With few groups of many nodes, simply divide the "many" nodes into another set of groups. In terms of performance, this category of groups had twice as many good performing groups as either of the other two categories. This may have been due to random chance, but is interesting to note. Examples of good groups are group IDs 11 and 12.

Once again, a similar structure emerges when looking at the best groups. The best groups have many high value nodes when compared to the other groups. In fact, the structure looks like the previous two combined. At the highest level in these groups, there are 3 groups

of 9 nodes and the best groups have many high value nodes grouped with the best node. Looking at the lowest level, you have 9 groups of 3 nodes, and you see the best node paired with another high value node. Taking group ID 11 as an example, it has nodes 23, 24, 25, and 26 all in the same top level group. And at the bottom level, it has nodes 24, 25, and 26 grouped together. Group IDs 13, 14, and 15 all performed worse because their nodes were split up between groups, as well as a less than ideal C value in the case of 14 and 15.

In summary, the general group structure that performs the best is one that groups the best nodes together at every level of groups. So it appears that you can simply group high value nodes together based on a heuristic or a number of simulations and achieve good performance. If these larger groups follow the same pattern as the groups for the 9 node experiments with prior knowledge, then performance will be close to the base case without groups. That, of course, does not take the speed increase into consideration! With 9 groups of 3 nodes and 3 groups of 9 nodes, the experiments finished in approximately 56% of the time it took for the base case with no groups. With 3 groups of 3 groups of 3 nodes, the experiments finished in approximately 50% of the time.

CHAPTER 6

Move Groups in the Game of Amazons

At the University of Alberta, there is an Amazons player called Arrow2. It is a MCTS player that also uses move groups. The player itself is based on the Fuego framework, which is a general framework for creating Monte-Carlo players [7]. Several experiments in self-play were carried out in order to determine the effects of move groups on performance. These experiments serve as the proof of concept for using move groups to increase performance.

6.1 Implementation Details

The basic Amazons player without move groups that was used as the opponent operated as follows:

- The core playing engine is a standard MCTS player with UCT.
- All possible moves are expanded when a new node is being added to the game tree.
- Simulations are random, but do not simulate a full game:
    1. Five or six random moves are played from a leaf node, ending in a position with black to play.
    2. A heuristic function that estimates territory control is used to determine the winner of that position.

The implementation was modified for move groups in the following way:

- Instead of taking a full move, a player's move was broken into three stages:

1. Select a queen to move.
2. The opponent passes. The pass move was necessary because Fuego assumes alternating play.
3. Select a destination square for the selected queen.
4. The opponent passes again.
5. Select an arrow destination based on the queen move.

- Disabled a time control optimization for when the current best move can no longer be surpassed in value. This was required since Fuego only considered the partial move rather than the entire three-part move.

These changes were relatively simple to implement within the Fuego framework. Several performance enhancements could be made beyond the implementation described. First of all, removing the need for the pass moves would cut down the size of the tree as well as tree traversal time. Second, the time control optimization could be re-enabled by forcing Fuego to look deeper when checking for early stops. The following experiments were performed without either of these enhancements.

6.2 Experiments with Amazons

The initial set of experiments tested Arrow2 without groups against a version that used groups on a fixed time per move. The two settings used were 5 seconds per move and 30 seconds per move. First player alternated for 100 games so each version played 50 games as each color. Players were given a single 2.5GHz core and enough memory for 5 million nodes in the game tree.

With 5 seconds per move, Arrow2 with groups won 69 of 100 games. With 30 seconds per move, Arrow2 with groups won 76 of 100 games. There were several notable differences in performance for individual moves. The most drastic difference was on the first move. Given that the first move has 2176 possibilities, this caused an enormous game tree after only a few moves. With 30 seconds time, Arrow2 without groups needed to prune the game tree approximately 60 times on the first move. It also only reached a maximum depth of 4 moves in the tree. On the other hand, Arrow2 with groups never once needed to prune the tree and reached a depth of 6 full moves. The difference in number of games simulated per second on the first move was also enormous. Without groups there were approximately 15,000 simulations per second, with groups there were approximately 175,000 simulations per second. Over the course of the game, the player without groups had an average of

approximately 100,000 simulations per second and the player with groups had an average of approximately 200,000 simulations per second.

The second set of experiments tested the same two players this time with a fixed number of simulations per move. The simulation limits used were 100,000 and 500,000. There was no significant difference between the players under these conditions. Arrow2 with groups won 54 of 100 games with 100,000 simulations and 52 of 100 games with 500,000 simulations.

In the case of Arrow2, the immense savings in move generation and tree traversal time from using groups caused a great increase in performance. A 76% win rate translates to approximately a 200 elo gain. The groups themselves were not the cause of any performance increase as shown by the fixed simulation experiments, but they were not designed to. The groups were designed primarily to save in move generation so that the large branching factor would not be an issue. Given the results shown here, other game players that have expensive move generation due to a large branching factor can also be enhanced simply by using move groups.

CHAPTER 7

Conclusions and Future Work

In this final chapter, all of the conclusions made in previous chapters are presented as a whole. This begins with a revisiting of the research questions by directly answering them. Any potential future work related to these conclusions is described as well. Lastly, more areas for future work are discussed.

7.1 Research Questions Revisited

The original questions that guided this research are now able to be answered in the following manner:

Does there exist a move group that performs better on average than using no groups, given the same number of simulations?

Chapters 3 and 5 showed this to be empirically true for 9 nodes and 27 nodes respectively. In the case of nine nodes, given the underlying values of the nodes, there were many groups that performed better than without groups. Groups were also able to use a wide range of C values and still achieve better performance. This result likely holds for any number of nodes, but would require additional confirmation. There is also the possibility of more good groups that have "unbalanced" groups, that is, groups that do not have the same number of nodes.

Will a randomly selected move group perform better on average than no move group?

There is a small chance that a randomly selected move group will perform better than no groups. In Chapter 3, the chance that a randomly selected move group would perform better

than no move group was less than 20%. As shown in Section 2.2, when the best node was 0.9, only 359 out of 2240 group and C value pairs performed strictly better than the base case. This percentage decreased even more when the problem became harder with a best node of 0.81. So, random groups are not the way to go about increasing search performance. There is a possibility that random unbalanced groups have a greater chance of performing better than no groups, but this requires further work.

What is the general structure of a move group that performs better on average than no move group?

This was investigated in Chapters 3-5. Chapter 3 found that the best node should be grouped with other high value nodes. In the 123 structure or 134 structure, groups achieved equal or better performance than the base case with no groups. These group structures were then tested in Chapter 4 in a scenario where perfect information about the underlying node values was unknown. With no information about the node values, groups could be introduced based on past simulation statistics to achieve similar performance to the base case. Overall, the 123 structure was more consistent since there was less chance it could accidentally create an undesirable structure. This was not the case with the 134 structure which could create the poorly performing 234 structure by chance.

Lastly, in Chapter 5, the number of nodes was increased to 27 in order to see if the general structure remained the same. All three group categories tested agreed that a similar structure worked. With 9 groups of 3 nodes, the best node needed to be in the same group as at least one other high value node for good performance. With 3 groups of 9 nodes, the best node needed to be paired with several other high value nodes. With 3 groups of 3 groups of 3 nodes, the best node needed to be paired with another high value node in its individual group and the mid level group needed to have more high value nodes.

This distribution of nodes causes many simulations to filter down to the specific group that contains the best node, due to the higher than average value of this group. Then the search can focus on distinguishing between the few nodes in that group rather than all possible nodes. There may be better performance to be achieved with unbalanced groups, however. An example is to have the best 3 of 9 nodes in a single group and the other 6 in their own group. The performance of such groups remains as future work.

How do you implement move groups in MCTS to improve performance?

Chapter 4 investigate this question in an artificial game. Two methods of using move groups were tested in that chapter. First, assume that there is no prior knowledge regarding the value of the nodes. In that case, an approximate value of nodes can be taken from the mean value of the nodes after a number of simulations. This value can then be used to create a

good group structure. The 123 structure performed well for this experiment only performing worse in a few places. However, this does not take into account the tree traversal speed up that using groups allows. All experiments that used groups finished in approximately 80% of the time it took for the experiments with no groups to finish. This speed up comes from the reduced number of node value calculations, see line 26 of the UCT algorithm in Listing 2.1. Using prior knowledge was the second method of testing move groups in an artificial game. Prior knowledge in the form of a noisy heuristic function was used to build move groups from the start. Again, the 123 structure performed better than the 134 structure. Compared to the zero knowledge case, prior knowledge performed better until the noise got very high. This can be seen in Table 4.3 and Table 4.7.

Using move groups in the game of Amazons was the topic of Chapter 6. For this game, move groups were used simply as a way to reduce the branching factor and cost of move generation. There was a significant speed increase observed when breaking moves into three stages which translated to a 200 elo gain. At the beginning of the game, more than 10 times as many games were able to be simulated and this decreased to twice as many simulations over the course of the game. With the same number of simulations, there was no performance change which suggests that the arbitrary move stages created were a reasonable structure already. More experiments could be done in order to determine if that is actually the case.

7.2 Other Future Work

7.2.1 Theoretical Approach for Move Groups

All conclusions made were based on empirical evidence, so a theoretical approach for move groups is currently unexplored. Such a theoretical basis may be able to give exact detail on the performance of move groups before implementing them. Also, with a theoretical basis in a simple artificial game, it stands to reason that this result would be able to be extended to more complicated games in a similar manner as UCT was an extension of UCB.

7.2.2 Additional Amazons Move Groups Experiments

Already mentioned is the possibility of analyzing the group structure created by the move stages explained. This would allow for a comparison to the performance of move structures already found. Also, the number of stages used in moves could be increased to see if the additional tree traversal savings increase performance. An example of a larger number of move stages would be first choosing a queen, then choosing a move direction, then choosing a

destination square in that direction, and finally choosing an arrow direction and destination square.

7.2.3 Move Groups in Other Games

There are still many other games where move groups have not been experimented with. As an example, Kriegspiel or general game playing have had MCTS players created for them, but no move groups [6, 8]. It would be interesting to test the effects of introducing move groups or using prior knowledge for move groups in a game where there is not a move generation speed up.

Bibliography

[1] Victor Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, Maastricht, 1994.

[2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multi-armed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

[3] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-Carlo tree search: A new framework for game AI. In *Artificial Intelligence and Interactive Digital Entertainment*, pages 216–217, 2008.

[4] Guillaume Chaslot, Mark H.M. Winands, H. Jaap van den Herik, Jos W.H.M. Uiterwijk, and Bruno Bouzy. Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 4(3):343–357, 2008.

[5] Benjamin E. Childs, James H. Brodeur, and Levente Kocsis. Transpositions and move groups in Monte Carlo tree search. In *Computational Intelligence and Games*, pages 389–395, 2008.

[6] Paolo Ciancarini and Gian Piero Favini. Monte Carlo tree search techniques in the game of Kriegspiel. In *Proceedings of the 21st international joint conference on Artifical intelligence*, pages 474–479, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[7] Markus Enzenberger, Martin Müller, Broderick Arneson, and Richard Segal. Fuego - an open-source framework for board games and Go engine based on Monte Carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270, 2010.

[8] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *Proceedings of the 23rd national conference on Artificial intelligence*, volume 1 of *Association for the Advancement of Artificial Intelligence*, pages 259–264, 2008.

[9] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *International Conference on Machine Learning*, pages 273–280, 2007.

[10] Sylvain Gelly and David Silver. Achieving master level play in 9 x 9 computer Go. In *Association for the Advancement of Artificial Intelligence*, pages 1537–1540, 2008.

[11] Sylvain Gelly and David Silver. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

[12] Sylvain Gelly and Yizao Wang. Exploration exploitation in go: UCT for Monte-Carlo Go. In *Neural Information Processing Systems*, 2006.

[13] Brian Haskin. A look at the Arimaa branching factor. http://arimaa.janzert.com/bf_study/.

[14] P.P.L.M. Hensgens. A knowledge-based approach of the game of Amazons. Master's thesis, Universiteit Maastricht, Maastricht, 2001.

[15] Richard J. Lorentz. Amazons discover Monte-Carlo. In *Computers and Games*, pages 13–24, 2008.

[16] Julien Kloetzer. Monte-Carlo opening books for Amazons. In *Computers and Games*, pages 124–135, 2010.

[17] Julien Kloetzer, Hiroyuki Iida, and Bruno Bouzy. The Monte-Carlo approach in Amazons. In *Computer Games Workshop*, pages 185–192, 2007.

[18] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning*, pages 282–293, 2006.

[19] Tomáš Kozelek. Methods of MCTS and the game Arimaa. Master's thesis, Charles University, Prague, 2009.

[20] Jahn-Takeshi Saito, Mark H.M. Winands, Jos W.H.M. Uiterwijk, and H. Jaap van den Herik. Grouping nodes for Monte-Carlo tree search. In *Computer Games Workshop 2007*, pages 276–283, 2007.

[21] David Silver and Gerald Tesauro. Monte-Carlo simulation balancing. In *International Conference on Machine Learning*, pages 945–952, 2009.

[22] Gerhard Trippen. Plans, patterns, and move categories guiding a highly selective search. In *Advances in Computer Games*, pages 111–122, 2009.

[23] Gabriel Van Eyck and Martin Müller. Revisiting move groups in Monte Carlo tree search. In *Advances in Computer Games*, Lecture Notes in Computer Science, pages 13–23. 2012.