Amazons Discover Monte–Carlo

Richard J. Lorentz

Department of Computer Science,
California State University,
Northridge CA 91330-8281, USA
lorentz@csun.edu

Abstract. Monte-Carlo algorithms and their UCT-like successors have recently shown remarkable promise for Go-playing programs. We apply some of these same algorithms to an Amazons-playing program. Our experiments suggest that a pure MC/UCT type program for playing Amazons has little promise, but by using strong evaluation functions we are able to create a hybrid MC/UCT program that is superior to both the basic MC/UCT program and the conventional minimax-based programs. The MC/UCT program is able to beat INVADER, a strong minimax program, over 80% of the time at tournament time controls.

1 Introduction

Amazons is a fairly new game, invented in 1988 by Walter Zamkauskas of Argentina and is a trademark of Ediciones de Mente. Amazons has attracted attention in the game-programming community because of its simple rules and its difficulty of play. On average there are more than 1,000 legal moves at any given position. Typically many hundreds of these legal moves seem quite reasonable [1]. A game lasts a maximum of 92 moves but is usually decided by about move 40, so games can be completed in a reasonable amount of time. Most people view Amazons as intermediate in difficulty between Chess and Go. Thus, Amazons provides a natural test bed for games research.

To date, most strong Amazons programs use traditional minimax-based algorithms. Two such programs are INVADER [1] and AMAZONG [12, 13]. However, Monte-Carlo (MC)-based algorithms have recently been used extremely effectively in Go programming [5, 9]. In fact, MC-based programs now completely dominate in the arena of 9×9 Go, and are beginning to do the same in 19×19 Go. It seems natural to test similar techniques in Amazons programming. Kloetzer, Iida, and Bouzy were the first to try this with their Amazons program CAMPYA [10]. We have modified INVADER, our Amazons playing program, to create INVADERMC, an MC-based Amazons program.

In the following we discuss the details of our approach. We explain how MC was added to INVADER and describe improvements to the basic MC algorithm that proved useful to INVADERMC. These include forward pruning and stopping MC simulations early. We then explain how UCT is added and discuss improvements that include finding the proper evaluation function, progressive widening, choosing the correct moment to stop a random simulation, and other tuning issues. We show the level of success we have achieved, and how the techniques we used in INVADERMC compare to those used in CAMPYA.

In the next section we review the rules of Amazons. We then summarize the main features of MC algorithms and briefly describe the UCT enhancement. Our emphasis throughout is on how these algorithms apply to Amazons programming. Section 3 provides all the details of our approach. We focus on the techniques that proved particularly effective for INVADERMC. We conclude in Sect. 4 with a summary of what we accomplished, a description of techniques that surprised us as not being particularly effective, and suggestions for further research.

2 The Game of Amazons and the Monte-Carlo Approach

This section provides basic background information. We explain the rules of Amazons and briefly describe the MC and UCT algorithms.

2.1 The Rules of Amazons

Amazons is usually played on a 10×10 board where each player is given 4 amazons that are initially placed as shown in Fig. 1(a). Although the game may be played with other board sizes and with different numbers and different initial placements of amazons, this version is considered standard and so will be the only version discussed here.

A move comprises two steps: first an amazon is moved like a chess queen through unoccupied squares to an unoccupied square. Then from this new location the amazon "throws an arrow", again like a chess queen, across unoccupied squares to an unoccupied square where it stays for the remainder of the game. The arrow now occupies that square, acting as a barrier through which neither amazons nor thrown arrows can pass. From the initial position shown in Fig. 1(a), one of White's 2,176 possible moves is to move the amazon from square D1 to square D8 and then throw an arrow to square B6, where the arrow must stay for the remainder of the game. This move is denoted D1-D8(B6) and is shown in Fig. 1(b). White moves first and passing is not allowed. The last player to move wins.

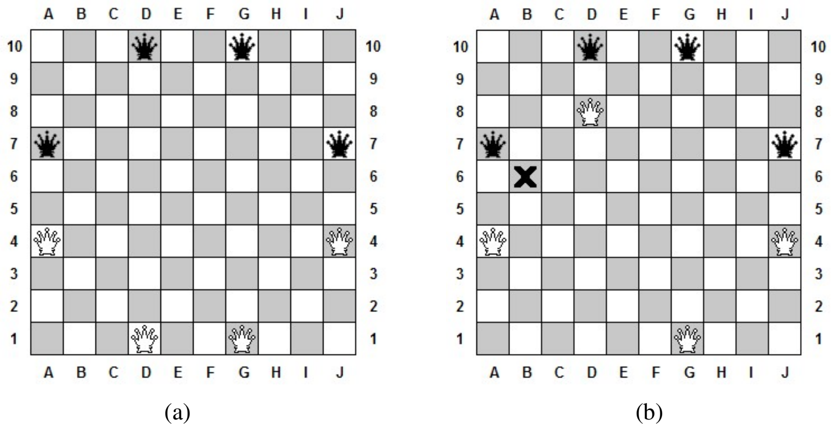

Fig. 1. The initial position and a typical first move

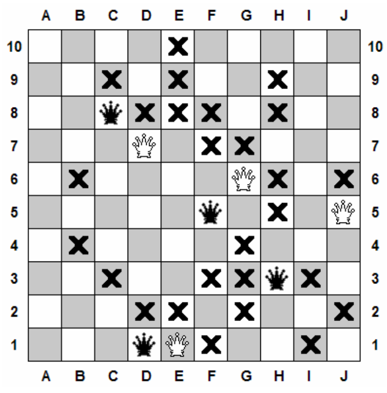

Fig. 2. MC algorithm failure

The concept of territory is important in Amazons. Territory concerns regions of the board that are under the complete control of one player, thus allowing that player to be the only player to be moving amazons and throwing arrows there. For example, in Fig. 2 below, if White moves J5-I5(J5) White will have complete control of the upper right region and this region will be considered White's territory. White can make 16 undisturbed moves there. The upper left side of the board in Fig. 2 appears likely to become eventually Black's territory and so we might say that Black has approximately seven squares of potential territory there. Obviously acquiring territory is an important aspect of the game since the side with the most territory should be able to make the last move and win the game.

2.2 Monte-Carlo Programming

The MC approach to game programming is well known and has been used in Go, for example, with varying degrees of success for more than 10 years [2]. The basic idea is that the best move is found by repeatedly sampling all possible moves either iteratively, random uniformly, or more cleverly. For each move sampled a random game is played to the end. This random game is sometimes referred to as a simulation. The wins and losses of the various simulations are tabulated and eventually the move that reports the highest winning percentage is played. In Amazons, as with Go, this simple algorithm can be surprisingly effective. If more random games are played, more information is acquired, so the speed of playing out random games is significant.

It is important to understand that a pure MC approach has a fatal flaw as can be seen in a game that was played by our first version of INVADERMC. The critical position is shown in Fig. 2 where INVADERMC is White.

From the position in Fig. 2, White made the move G6-C6(G6). This loses immediately because Black can now play H3-I4(I6) closing off White's potential territory in the upper right. But that is the only move that wins for Black, so White still wins the vast majority of the random games after making that faulty move. If instead White plays a move like J5-I5(J5), White will be winning, but the game is actually quite close. So White wins fewer of the random games and this winning move actually appears less desirable to INVADERMC.

2.3 The UCT Algorithm

One popular solution to this problem is referred to as UCT [11] and stands for "upper confidence bounds extended for trees". The idea is that instead of just keeping a list of candidate moves and running random games for each of them, we maintain a search tree where each node of the tree keeps track of the winning record for that particular position. We call this a UCT tree. Starting at the root of the UCT tree a path is found to a leaf node by proceeding down the tree, choosing nodes that have the highest expansion value. The method for calculating the expansion value is described below. A random simulation is then run from the position corresponding to this leaf node and this leaf may then be expanded, adding its children to the UCT tree. The decision to expand a leaf node is usually based on how many times that node was visited.

The expansion value of a node is equal to the winning percentage of all simulations that have passed through that node plus a bias value. The bias value allows nodes that have not been visited very often to obtain a share of the simulations and is calculated using the formula

$$k \cdot \sqrt{\frac{\log(parent\_count)}{node\_count}} \qquad (1)$$

In this formula *parent_count* corresponds to the number of simulations that have passed through the parent of the node, *node_count* is the number of simulations that have passed through the node, and *k* is a constant that is tuned according to the particular application. A small value for *k* means moves that have been performing well will continue to be expanded with high likelihood (exploitation) while a large value means more moves tend to be tried at each level (exploration) [8].

For example, when using UCT from the position in Fig. 2, the node in the tree corresponding to the move G6-C6(G6) will have child nodes corresponding to each of Black's follow-ups. Since the child node corresponding to the move H3-I4(I6) will win most of its simulations, that path down the tree will be the one taken most often, meaning that the top node G6-C6(G6) will start losing more games, eventually showing it to be undesirable. Meanwhile, because of the bias value, another move like J5-I5(J5), will eventually be explored frequently to obtain a higher winning percentage and UCT will continue to grow the tree and confirm that that move (or something similar) is the better move in this position.

In the next section we explain in detail how we incorporated the basic ideas of MC/UCT into INVADERMC and then made improvements to create a hybrid MC/UCT program. This hybrid version is considerably stronger than minimax-based INVADER.

3 The Hybrid MC/UCT Approach

INVADER, our strong minimax-based Amazons program, has competed in numerous Computer Olympiads and has finished second or third every time. We used INVADER as the basis for creating INVADERMC, our MC/UCT based program. We will summarize the various basic algorithms and modifications to these algorithms we used and how they affected playing quality to show how we ultimately created a strong MC/UCT-based hybrid program. Unless otherwise stated all tests were 100-game matches where each game of the match began with a different two-move opening. Games were played at tournament speeds, i.e., at the rate of 30 minutes for the entire game per player.

3.1 Adding MC to INVADER

Our experiments began with a pure MC-based version of INVADER. The idea was to see how strong our program could become by using only the MC-based idea, that is, by not building any trees as is required by more sophisticated techniques such as UCT. The most basic version simply repeatedly iterated over all legal moves, ran a random game from each such position, and kept track of the results. The main obstacle that needed to be overcome was to find a way to run random games quickly. There are usually over 1,000 legal moves in a typical Amazons position. So, finding a random move without bias among all the possibilities can be quite time consuming. Since an amazon's move is comprised of two parts, the move and the throw, we first select a random move uniformly among all the possible amazon moves and then from that move select a random throw. This is the same technique used in CAMPYA [10]. It does add a slight bias towards amazon moves that have fewer throws available, i.e., amazons that are in more restricted positions. But the cost of generating all possible legal moves slowed the random generation by more than a factor of 10 and so was deemed too slow for any further study.

There is an obvious way to speed this up even more. We can first randomly select one of the four amazons, then randomly select one of its moves, and finally select a throw from there. Moves produced using this method were of conspicuously lower quality and almost no test games were won using this approach. So, we immediately rejected this approach. All further tests were done using the two-step move generation process. This basic MC program did surprisingly well, winning 15% of the games in a test match against our min-max-based INVADER.

With MC-based programs the more simulations we can generate the more information we can gain. So, it is beneficial to be as efficient as possible. Subsections 3.2 and 3.5 deal with some other efficiency-related topics. However, we refrain from any discussion or tests concerning low-level efficiency concerns such as bit string board representations.

3.2 Stopping the MC Simulation Early

The basic MC program did quite well, but there are a number of things that can be done to improve the quality of its moves. The first has to do with choosing the correct time to end a simulation. Instead of waiting until a player loses because it has no legal moves we can stop a game as soon as all amazons have been completely isolated from its enemies. This gives a more precise and earlier determination but comes at a price of forcing us to check for this property during the simulations, thus slowing them down. It improved performance but suggested an even more important improvement.

The point of a random simulation is to provide statistical evidence about the strength of a move. Instead of simulating to the end of the game we can invoke an evaluation function earlier to help us determine whether the simulation is leading to a winning or a losing position. INVADER has a fairly strong evaluation function. So, it is easy to test this premise by using it in InvaderMC. Note that this approach is never used in MC Go programming because evaluation functions are notoriously poor in Go. In Go the random simulations to the end of the game provide better information than an evaluation function.

When using the evaluation function we have a choice of recording either the presumed margin of victory, or simply who won. We choose the latter. In other words, only the sign of the evaluation is actually used at the end of the simulation. Though Campya programmers claim to find advantage in using the margin of victory [10], so far we have been unable to improve on using the simple win/loss results.

But when is the correct time to invoke the evaluation function? The two obvious choices are: (1) after a certain stage of the game (e.g., after move number 30) has been reached; or (2) after a certain number of random moves have been played. Early experiments indicated that choice (2) produces a better InvaderMC, winning more than twice as many games against Invader than various versions trying the other approach. Applying the evaluation after 6 random moves appears best. Waiting until 10 moves caused InvaderMC to win 10% fewer games and waiting until 14 moves dropped the win rate about 30%. Also, applying the evaluation after only 4 moves had a similar negative effect, dropping the win rate 25%. Campya [10] does something similar, though it is not clear exactly how deep into the simulation they invoke their evaluation function.

3.3 Forward Pruning with MC

A second possible use for the evaluation function is to do (unsafe) forward pruning. Rather than waste time and space building the UCT tree for all legal moves we can use the evaluation to select the moves that are most likely to be good. With some tuning we found that selecting the top 400 moves for MC simulation seemed to give good results. More detailed tuning in the UCT case also found 400 to be a good choice. For example, our optimal UCT program that keeps the top 400 moves wins 80% of its games against normal Invader. Instead, keeping the top 800 moves produces a slightly lower win rate, but when keeping 1,200 moves the win rate drops to 57%. Also, keeping only the top 200 moves shows a similar drop, winning only 61% of its games.

Extending this forward pruning idea, even among these 400 moves we select, we prefer gaining information about moves that are likely to be good and only occasionally simulate moves that appear to be weak, giving them a chance to prove themselves if necessary. We implemented this technique by essentially doing a depth-1 UCT search. We used the UCT bias values to determine which move we would simulate, but we never extended the search tree beyond its initial depth of 1.

Using these algorithms InvaderMC played at nearly the level of Invader, winning 40% of its games. However, as pointed out in Sect. 2, there is a limit to how well we can expect a pure MC program to play.

3.4 Adding UCT

UCT extends the MC algorithm by creating a search tree of moves so that nodes (moves) that are performing well create child nodes beneath them and then the more promising children are given more simulations. Where MC tends to concentrate on promising moves (nodes), UCT concentrates on promising variations (tree paths). The basic algorithms for UCT are straightforward but a great deal of care is needed to make the best use of UCT.

As mentioned in Subsection 2.3, it is important to choose the proper exploitation/ exploration constant *k*. Experimental results indicate that in the case of INVADERMC, choosing the constant so that the UCT tree created ultimately ends up with a maximum depth of around 8 (assuming normal tournament time controls of 30 minutes per game) seems optimal. Setting $k = .35$ achieves this kind of result.

Attempts at deeper exploitation miss too many important refutations while more exploration does not give sufficient time for the best moves to reveal themselves. We compared various choices of *k* using our best INVADERMC which wins 80% of its games against INVADER. If we divide *k* by 2 the UCT tree now grows to depths of around 10 or 11, but INVADERMC only wins 63% of its games. If we multiply *k* by 2 the UCT depths tend to be around 6 but we obtain results very similar to those using the original *k*. This version won 78% of its games. However, if we multiply *k* by 4 we obtain a drastic change. This version has UCT tree depths of around 4 or 5 and only wins 25% of its games against INVADER.

By way of comparison, minimax-based INVADER is usually not able to play beyond depth 3 in the early stages of the game and can sometimes reach depth 8 or 9, but only at the latest stages. Minimax INVADER is, of course, implemented with most of the usual improvements including alpha-beta pruning, the killer heuristic, etc. It also prunes all but the best 400 moves at each node of the minimax tree. Since it is not clear what the relationship is between UCT depths and minimax depths this comparison is presented simply as a point of interest.

Also mentioned in Subsection 2.3, we need to decide when to expand leaf nodes in the tree. The idea to not necessarily expand a leaf the first time it is visited was first mentioned by Coulom in a slightly different context and is frequently used in MC/UCT Go programming [5]. There are a number of reasons why it is not prudent to expand immediately in the case of Amazons. First, to expand a node takes a great deal of time since all possible moves need to be generated and all of these moves need to then be evaluated. Experiments and intuition suggest this time might be better spent doing more simulations. A second reason is that even though we employ forward pruning and expand leaf nodes to have only 400 children, 400 is sufficiently large to cause the UCT tree to grow quite fast and can quickly exhaust available memory. Third, a single random Amazons simulation does not necessarily provide that much information about the likelihood that a given node is favorable for White or Black. We need to gather more data. Again, experiments with our best version of INVADERMC indicate that a leaf should not be expanded until it has been visited around 40 times. Dropping that value to 20 reduces the win rate by a small amount. Dropping it to 5 produces a program whose win rate is only 65% compared with 80% for our best INVADERMC. Raising the value to 120 has a similar negative effect.

By adding these basic UCT features to INVADERMC we have created a program that outperforms INVADER. This version of INVADERMC is able to beat INVADER approximately 60% of the time. We now describe further enhancements that move us closer to our best version that achieves a win rate of 80% against INVADER.

3.5 Choosing the Proper Evaluation Function

As is the case with most game-playing algorithms, the speed of the evaluation function is critical. Not only does an evaluation take place with every simulation, but every time a UCT tree node is expanded there will typically be on the order of 1,000 evaluations to perform corresponding to the average number of legal moves from a position. All of these nodes need evaluating so that we can select the top 400. There is an obvious tradeoff here: the faster the evaluation, the less information it provides so the more simulations we will need. But a faster evaluation also allows more simulations in the same amount of time. Our experience shows that very fast yet naïve evaluations allow significantly more simulations, but the lost information makes for a poorer player. Nevertheless, we were able to achieve some success by creating less precise but faster evaluation functions.

INVADER uses some quite complicated calculations to estimate accurately the territories of the players. One of the most time-consuming aspects involves doing a flood fill type calculation in each area of potential territory. We then rate the value of this potential territory based on factors such as the calculated size and the number of external access points to this potential territory.

By being less careful about the exact sizes and likelihoods of the players acquiring territory, but still taking territory into account on a rougher scale, we are able to evaluate a board in about two thirds of the time the normal evaluation takes. We do this by simply saying that a group of contiguous squares constitutes potential territory if every square in the group requires fewer queen moves to reach a friendly amazon than it takes to reach an enemy one. With this added speed INVADERMC is able to create a larger UCT tree while still evaluating reasonably accurately. INVADERMC using this faster evaluation function improves considerably, winning 67% of its test games against INVADER. It is worth noting that using the new evaluation function in the normal INVADER program makes INVADER weaker. It only won 30% of the games in a test against INVADER with its original evaluation. Also, our best INVADERMC program wins 89% of its games against INVADER using the new function. So the new evaluation benefits INVADERMC but harms INVADER.

3.6 Evaluation Parity Effect

Our simulations make a fixed number of random moves, usually 6, before the evaluation function is invoked. Since these simulations can commence from any leaf in the UCT tree, depending where we are in the tree, the simulation might terminate with either White or Black to play. This parity problem is particularly problematic with INVADERMC because its evaluation values swing quite widely according to whose turn it is to move. This ultimately causes the values that propagate up the UCT tree to be unnecessarily undependable. A useful fix is to terminate all random simulations with the same player to move. This means that, for example, when we are doing 6-move simulations we actually do 6 and 5-move simulations, depending on where in the tree we are relative to the starting position. This enhancement produces a significant improvement in the playing strength of INVADERMC. Our best INVADERMC without this enhancement only wins 73% of its games.

3.7 Progressive Widening

The final important enhancement has many names. We choose the term "progressive widening." The basic idea is to give UCT a little assistance by first considering only the "best" children of a node and then gradually adding the other children until at some point all children of a node are actually being examined by the UCT algorithm [3, 4, 6]. There are many ways to implement this, some quite sophisticated. We have implemented a very basic version. We see how many times a node has been visited and if this number is small only some of the children will be examined. As the node is visited more often, we gradually add more child nodes to be considered by UCT until we eventually end up considering all 400 children when the node has been visited sufficiently often. To give a sense of the actual values, on our machine INVADERMC usually constructs a UCT tree of about 1,500 nodes under normal time controls. Until a node has been visited 1,000 times (recall that leaves do not expand until they have been visited 40 times) we only consider 5 children. We increase the number of children by increments of 5 for every 1,000 visits and all 400 nodes are not considered until the node has been visited 80,000 times. Without this improvement our best INVADERMC wins fewer than 70% of its games against INVADER.

3.8 Minor Tuning

There are two other minor tuning issues that are interrelated and deserve mention. Though individually they do not contribute significantly to the strength of the program, combined they do seem to have a small positive effect. In most games, including Amazons, it is important to get off to a good start early. Therefore, many game programs, especially MC based programs, allocate more time for moves in the early stages of the game. INVADERMC does this in a primitive, but still useful way. Given the time control, we calculate how much time is available per move and then allow triple this base time for the first 10 moves, double for the next 10 moves, 1.5 times the base time for the next 10, the base time for the next 10 moves, and then allocate the remaining time to the remaining moves. However, the time allocated to these final moves is so small that often the UCT algorithm does not have time to build a sufficiently big tree to find a decent move. So, for these last moves we switch over to our basic minimax engine.

INVADERMC configured with all of the above features is significantly stronger than basic INVADER. After playing 400 games between the two, INVADERMC wins about 80% of the games.

4 Summary of Our Findings and Future Work

We have found that by adding a hybrid UCT structure to our Amazons playing program, INVADER, the new program, INVADERMC, is significantly stronger and is able to beat INVADER 80% of the time. It is not a pure UCT approach, because we prune (in an unsafe manner) child nodes and we run the random games for only about 6 moves. But our use of progressive widening and our tuning of the UCT depth and node visits per leaf expansion variables is standard when using UCT. Figure 3 below

Pure MC	INVADER	15%
MC with eval. func. based pruning	INVADER	40%
Basic hybrid UCT	INVADER	60%
Basic hybrid with improved evaluation	INVADER	67%
Hybrid UCT with all improvements	INVADER	80%
Hybrid UCT w/o parity adjustment	INVADER	73%
Hybrid UCT w/o progressive widening	INVADER	70%
INVADER with new evaluation	INVADER	38%
Hybrid UCT with all improvements	INVADER with new evaluation	89%

Fig. 3. Summary of test results

summarizes the test results where all tests were done with at least 100 games and the total time per game per player was set to 30 minutes. The first program's winning percentage over the second is shown in the third column. The first five rows show how INVADERMC performs against minimax INVADER as algorithmic features are added. The next two rows show the effect of removing one of the later enhancements. The last two rows demonstrate that the new evaluation function actually hurts the minimax INVADER, though it was a significant help for INVADERMC.

We tried a number of other techniques that seemed promising on paper but when implemented did not deliver. After UCT appeared on the Go scene and proved useful, the next big step was the use of "heavy playouts", that is, guiding the random simulations so that more natural looking random games are played [7]. All efforts to incorporate heavy playouts in INVADERMC have so far been disappointing, resulting in weaker play. Since this has had such a significant positive impact on Go-playing programs we feel that additional work in this area is critical.

INVADER has a strong and presumably reasonably accurate evaluation function. So, it seemed natural to try to eliminate completely the random simulations and simply use the evaluation function to judge the probability of winning from a given position. We were unsuccessful with this approach, suggesting that the sign of the evaluation does a good job of suggesting who is winning but converting the value of the evaluation into a winning probability is quite difficult.

It has been suggested that having the UCT tree maximize the scores rather than the winning percentages might be advantageous [10]. In Go programming, it has been shown that this is not the case [9]. Likewise, our experience with INVADERMC indicates it is best to maximize the winning percentages.

We still find it mysterious that random simulations of length 6 work so well. Adding to the mystery is the fact that a change of 2 in either direction immediately hurts the winning percentage by at least 5%. We need to understand better the mechanisms involved. We also still hope to find a way to convert or modify the evaluation function so that it can better express the actual probability of a win. An effective way of doing this will surely produce a much stronger program.

One idea to tweak the current program that might produce an improvement has to do with the UCT constant *k* that dictates the depth of the growing tree. It might prove beneficial to modify *k* as the game proceeds. For example, since later in the game there are usually fewer potential good moves we might want to change the constant to allow deeper trees to grow at the expense of possibly missing an unlikely looking move that ultimately turns out to be strong.

It is well known that self-testing is not the ideal way to test a game-playing program. However, the dearth of Amazons-playing programs, combined with the near nonexistence of publicly available programs, and the fact that the one accessible strong program known to us (AMAZONG) cannot be easily configured to play long test sets with INVADERMC make this approach necessary for now. As a kind of sanity check we registered on a game-playing Web site and entered an Amazons ladder. All moves made were those suggested by INVADERMC after approximately 10 minutes of thinking. Under this configuration INVADERMC went undefeated with a score of 9 and 0, and eventually landed at the top of the ladder. This result at least suggests that we are not going down the completely wrong path. Time constraints have prevented us from playing many games against AMAZONG (fast games are too much of a disadvantage for MC/UCT based programs), yet the few games we have played have produced a winning percentage for INVADERMC.

Acknowledgments. We want to thank the anonymous referees for their careful, thoughtful, and thorough comments and for pointing out our too frequent lapses. This paper is greatly improved as a result. We would also especially like to thank HLL for all the guidance, support, and inspiration she provided over the years.

References

1. Avetisyan, H., Lorentz, R.: Selective Search in an Amazons Program. In: Schaeffer, J., Müller, M., Björnsson, Y. (eds.) CG 2002. LNCS, vol. 2883, pp. 123–141. Springer, Heidelberg (2003)
2. Brügmann, B.: Monte Carlo Go. Technical report, Physics Department, Syracuse University (1993)
3. Cazenave, T.: Iterative widening. In: In 17th International Joint Conference on Artificial Intelligence (IJCAI 2001), pp. 523–528 (2001)
4. Chaslot, G.M.J.-B., Winands, M.H.M., Uiterwijk, J.W.H.M., van den Herik, H.J., Bouzy, B.: Progressive strategies for Monte-Carlo Tree Search. New Mathematics and Natural Computation 4(3), 343–357 (2008)
5. Coulom, R.: Efficient selectivity and back-up operators in Monte-Carlo Tree Search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M(J.) (eds.) CG 2006. LNCS, vol. 4630. Springer, Heidelberg (2007)
6. Coulom, R.: Computing "Elo Ratings" of Move Patterns in the Game of Go. ICGA Journal 30(4), 198–208 (2007)
7. Drake, P., Uurtamo, S.: Move Ordering vs Heavy Playouts: Where Should Heuristics Be Applied in Monte Carlo Go? In: Proceedings of the 3rd North American Game-On Conference (2007)
8. Gelly, S., Wang, Y.: Exploration exploitation in Go: UCT for Monte-Carlo Go. In: Twentieth Annual Conference on Neural Information Processing Systems (2006)
9. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT for Monte-Carlo Go. Technical Report 6062, INRIA (2006)
10. Kloetzer, J., Iida, H., Bouzy, B.: The Monte-Carlo approach in Amazons. In: van den Herik, H.J., Uiterwijk, J.W.H.M., Winands, M.H.M., Schadd, M.P.D. (eds.) Computer Games Workshop 2007, Amsterdam, The Netherlands, pp. 185–192 (2007)

11. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
12. Lieberum, J.: An Evaluation Function for the Game of Amazons. Theoretical Computer Science 349(22), 230–244 (2005)
13. Lorentz, R.: First-time Entry AMAZONG wins Amazons Tournament. ICGA Journal 25(3), 182–184 (2002)