GAMES

Iterative Widening

Tristan Cazenave
Labo IA, Université Paris 8
2 rue de la Liberté, 93526 Saint Denis, France.
cazenave@ai.univ-paris8.fr

Abstract

We propose a method to gradually expand the moves to consider at the nodes of game search trees. The algorithm is an extension of Abstract Proof Search, an algorithm that solves more problem than basic Alpha-Beta search in less time and which is more reliable. Unlike other related algorithms, iterative widening adapts to the game via general game definition functions. In the game of Go, it can solve more problems than the original non widening algorithm in approximately half of the time, as shown by experimental results.

1 Introduction

We propose a method to gradually expand the moves to consider at the nodes of game search trees. The algorithm begins with an iterative deepening search using the minimal set of moves, and if the search does not succeed, iteratively widens the set of possible moves, performing a complete iterative deepening search after each widening. Iterative widening enables to reduce by almost a factor two the time used for solving capture problems in the game of Go and solves more problems than the original non-widening algorithm, as shown by experimental results.

The second section describes the search algorithm and compares it with related existing algorithms. The third section explains the game used to perform our experiments. The fourth section deals with theorem proving in Go. The fifth section gives hints on how to define and combine the gradually expanding sets of moves and defines some of these sets for the capture game in the game of Go. The sixth section details experimental results.

2 The Search Algorithm

2.1 The Basic Search Algorithm

In this subsection, we describe the initial non-widening algorithm.

We use Abstract Proof Search [Cazenave, 2000] to develop AND/OR proof trees for the game of Go. This is an iterative deepening Null Window Search [Marsland and Björnsson, 2000], that uses some game specific functions to efficiently prove theorems about goals in games. This search algorithm is much more efficient than a usual alpha-beta search, and scales well when given more resources.

We do not use forward pruning with null move search because we are looking for exact results, some of our experiments show that null move pruning can speed-up the algorithm with very little drawbacks. However, Abstract Proof Search often stops searching when alpha-beta continues searching: it stops searching at AND nodes when it cannot prove the goal can be reached in less than four plies after a null move. When the goal can be reached in four plies or less, it selects the only relevant moves that can prevent reaching the goal with another small five plies search. So Abstract Proof Search can be considered as performing a kind of forward pruning [Smith and Nau 1994] that improves the quality and the rapidity of the search instead of making it worse as with usual forward pruning.

We use iterative deepening [Korf, 1990], transposition tables, quiescence search, null-window search when not at the root and the history heuristic [Schaeffer, 1989]. We stop search early when the goal is reached. In the experiments of this paper, we stop search after the first winning move.

In our tests on the capture game, we also use incrementality so as not to recalculate all the abstract properties of the strings after each move. We keep track of the liberties of the strings, and of the adjacent strings of each string. Each intersection is associated to a bit in a bit array so as to optimize checking of liberties.

Transposition Tables are used to detect identical positions and return the associated value if the search depth of the stored position is greater than the depth of the node or if the value is +INFINITY or –INFINITY. Transpositions are also used to recall the best move from previous search in the position and try it first when searching deeper so as to maximize cut-off.

The History Heuristic is used to order the moves that are not given by the transposition table. When all the moves at a node have been tried, the move that returned the best value, or the one that caused a cut-off, is credited with 2^{Depth}. At each node, the moves are sorted according to their credit, and tried in this order.

In our experiments in Go, a Quiescence Search is performed at leaf nodes. The quiescence search alternatively calls two function QSCapture() that plays on the liberties of the string to capture if it has 2 liberties, and QSSave() that plays the liberty of the string to capture and the liberties of the adjacent strings in atari[1], if the string to capture is in

[1] atari means only one liberty left

atari. This ensures that the Quiescence search returns correct results on the capture status of the string and quickly reads simple and deep ladders.

Iterative deepening stops when a winning move is found or as soon as the maximum processing time has elapsed.

2.2 Iterative Widening

We now define how we have applied iterative widening to our search algorithm.

We define sets of abstract possible moves, that can be tried at the node of the search tree at a given widening threshold. Sets are numbered, the following set always contains the previous set. The algorithm tries the sets of moves in the same order as their numbers.

For example, if the sets of possible moves to be tried at different widening threshold are the sets S1, S2,…,Sn. We have $S1 \subset S2 \subset \ldots \subset Sn$. The algorithm begins with an Abstract Proof Search, trying the moves in the set S1. If this search fails, it then makes another search with the S2 set. And so on until all the possible sets have failed, or the allotted time has elapsed.

For each problem, two search trees are usually expanded. The first one with White playing first and the second one with Black playing first. However, we discard the search with Black trying to prevent the goal, if the search with White playing first does not find a winning move. Similarly, if the problem consists only in finding a winning move, the search to find the preventing move is not performed.

The iterative widening algorithm consists in calling first the iterative deepening search algorithm with the first move function that returns the moves of the first set. If the search does not succeed, it continues with the following sets until the search succeeds or the time has elapsed, or the search eventually fails with the ultimate set.

In our first experiments, when a search fails at a given widening threshold, the transposition table is reinitialized and a new search is performed with the next set if possible. Further experiments have shown that this can be improved when reusing the same transposition table for all the widening steps.

2.3 Related Work

After having designed the method, we found that it has links with Iterative Broadening [Ginsberg and Harvey, 1992]. This method is successful in constraint satisfaction search [Meseguer and Walsh, 1998]. However, Iterative broadening is not the same algorithm as ours because it sets an artificial breadth cutoff *c*, and backtracks at most *c* times at any node of the tree. It iteratively increases *c*, and information can be memorized for the next iteration. Experiments by Ginsberg and Harvey in applying Iterative Broadening to Chess gave disappointing results because the move ordering of current Chess program is already near the optimum. Another previous related approach was phased state space search [Marsland and Srimani, 1986], an hybrid algorithm between SSS* and Alpha-Beta that partitions the set of all immediate successors of MAX nodes into k groups, and limits the search to one partition per phase. The originality of our method is that it is more concerned with widening at AND nodes, and that it provides game independent games definition functions that enable large speed-ups, and adapt to the game and the position at hand to select the worthwhile moves, rather than setting an artificial breadth cut-off.

3 The Capture Game

In our experiments, we mainly used the capture game to test the algorithm. The capture game is the most fundamental sub-game of Go. It is usually associated with deep and narrow search trees. It has strong relations with connections, eyes, life and death, safety of groups and many important Go concepts.

Figure 1 gives some examples of the capture game. The first example is called a geta, a white move at A captures the black stone marked with an x, it can be solved in 5 plies. The second example is an illustration of the capture game as a sub-game of the connection game, a white move at B captures the marked black stone and connects the two white strings, it requires 9 plies.

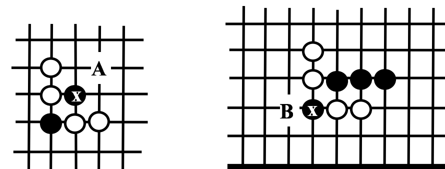

Figure 1. Examples of captures

4 Theorem Proving in Go

In this section, we show how to prove tactical goals in the game of Go. Our method has also proven useful in other games such as Phutball, Gomoku or Hex.

Let P be a position and m a move. Let play(P,m) be the function that returns the position after move m on position P. We can define games:

gi_k (P,W) = W can capture in k White moves if W plays first in position P and if perfect play and alternated moves are assumed:
$\exists$ move { P_1=play(P,move), $g_{k-1}(P_1,W)$ }.

ip_k (P,B) = gi_k (P,W).

S_k (P,B) is the set of all black moves that prevent W from capturing in k white moves in position P if B plays first when ip_k (P,B) is verified.

g_k (P,W) = $\exists$ m { m≤k, ip_m (P,B), $\forall$ move $\in$ S_m (P,B) { P_1=play(P,move), $\exists$ o {o≤k, $gi_o(P_1,W)$ }}}.

In some previous research [Cazenave, 1998], we have shown it is possible to generate programs for the game definition functions using the rules of the game defined in a logic language, and a metaprogramming system. The generated programs select the same moves as our search based game definition functions. They can be generated dynamically by safely generalizing trace of proofs on examples [Cazenave, 1996], or statically by specializing the definitions of the games functions on the rules of the game [Cazenave, 1998].

Recently, we defined an equivalent search based algorithm that selects the same moves using small game definition functions based on abstract properties of the games [Cazenave, 2000]. The algorithm is more concise and easier to program than our previous metaprogramming system. It needs to know the complete set of abstract moves that can change the outcome of a fixed depth small search. For example, in the leftmost diagram of figure 2, the black stone has only one liberty and can be captured in one white move at A, i.e. in a 1 ply search. The only abstract black moves to prevent the capture are the liberty of the stone and the liberties of its adjacent strings in atari. Based on the logic of our previous system, we have designed complete sets of abstract moves that can prevent one, three and five plies search.

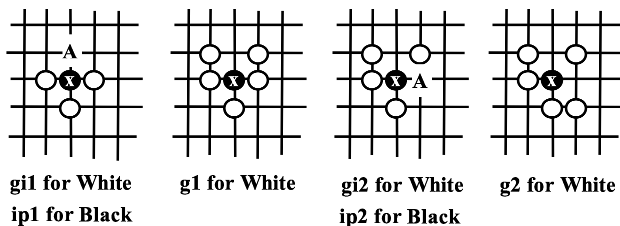

Figure 2. Examples of games

In the following we will give names to the different games, according to their possible outcomes. The names of the games are usually followed by a number that indicate the minimum number of white moves in order to reach the goal. A game that can be won if White moves first is called 'gi', a game where White wins unless Black plays first is called 'ip', it is the almost the same as 'gi' except that it is associated to black moves. A game that White can win even if Black plays first is called 'g'. A game is always associated to a player, the g and gi games are associated to the player that can reach the goal, the ip games are associated to the player that tries to prevent the opponent from reaching the goal. The gi and ip games are also associated to a set of moves. The ipn moves are the moves that prevent a string to be captured in n moves by the opponent. For example, the ip1 moves are the moves that may prevent a string in atari to be captured in one move (i. e. playing the liberty, or capturing an adjacent string).

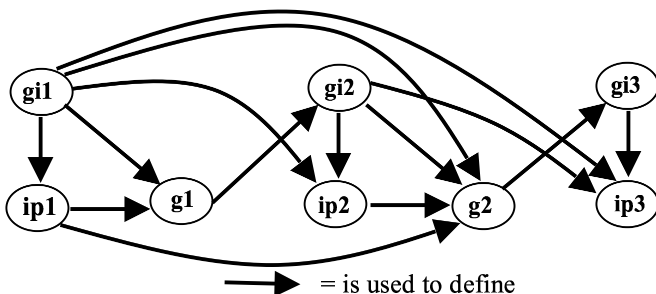

Figure 3. The dependencies between games

A *forced move* is a move associated to an ip game. For example, when the program checks whether a game is ip2, it begins with verifying that White can capture in two moves if he/she plays first (a gi2 game, associated to a three plies search). The forced ip2 moves are the black moves that prevent White from capturing the string in two moves once one of the Black ip2 moves has been played (we can say that the gi2 game has been invalidated by the black move, for example the move at A in the third diagram of figure 2 is an ip2 move for Black and a gi2 move for White).

Figure 3 gives the dependencies between games definitions. A game can be defined using the games for the lower number of plies, for example, the g1 game for White is defined as: the game is ip1 for Black, and all the forced black moves lead to a gi1 game for White after the black move (as in the second diagram of figure 2). So the g1 game is defined using the definitions of the gi1 and of the ip1 games, as it is shown in figure 3 where arrows go from the gi1 and ip1 games to the g1 game. Another example is the gi3 game for White: a white move leads to a g2 game for White. So the gi3 game depends on the g2 game only. In order to make things clear some examples of games are given in the figure 2.

The gi2 game relies on the g1 game as shown by the arrow between g1 and gi2 in the figure 3. A gi2 game can be tried if the string to capture has two liberties. For each of the two liberties, the program tries to fill the liberty, and verifies that the game is g1 after the liberty is filled, using the g1 game definition function.

The function defining the ip2 game and its associated moves is equivalent to finding the forced moves that prevent the string to be captured in 3 plies. It is checked at every AND nodes of the Abstract Proof Search tree provided the ip1 function has not been verified before.

The ip2 game definition function is defined using simple concepts and the functions corresponding to other games. Here again, as shown in the figure 3, the ip2 function relies on the functions defining the gi1 and gi2 games. The function adds the forced moves to prevent a 3 plies capture (the ip2 moves). The function begins with verifying that the string can be captured in two moves if White plays first, by calling the gi2 game definition function. If it is the case, the function finds the complete set of black moves that may change the issue of the gi2 game. Then, for each move of this set, it plays it and checks whether the game is not gi1 and not gi2 after the move. If it is the case, then the move has been successful in preventing the gi2 game, and is therefore an ip2 black move, so it adds the move to the set of forced ip2 moves.

The g2 game definition is a little more complex than the previous ones because there are two possibilities:

Either the function ip1 is verified, the black string can be captured in one move by White, so it has only one liberty. After playing on its liberty the string can still be captured in two white moves (the gi2 function applies).

Or the function ip2 is verified, but all the moves that could prevent the game to be gi2 do not work, so the ip2 function returns an empty set of forced moves. In that case, the game is won for White because none of the black moves to prevent gi2 works. The rightmost diagram of figure 2 is an

example of this kind of g2 game.

The gi3 and ip3 game definition functions use the same kind of definitions as the gi2 and ip2 functions.

At each node and at each depth of the Abstract Proof Search, the game definition functions are called, they are equivalent to the development of small search trees. So Abstract Proof Search is a search algorithm that can be considered as developing small specialized search trees at each node of its search tree. At OR nodes, the program first checks if the position is gi1, if it is not, it checks if it is gi2 (equivalent to a three plies deep search tree), and if it is not, it checks if it is gi3 (equivalent to a five plies deep search tree). As soon as one of the gi games is recognized, the program stops searching and returns Won. Otherwise it tries the OR node moves associated to the position. At AND nodes, the same thing is done for ip1, ip2 and ip3 games, if none of them is verified, the programs returns Lost, otherwise it tries the moves associated to the verified ip1, ip2 or ip3 game.

5 Designing the Gradual Sets of Moves

It is quite important to carefully choose the sets of moves. The first set is better if it contains the moves that are likely to reach the goal. Typically, the last set contains all the moves worth trying. We have separated the sets for the OR nodes and the AND nodes of the tree, as they have completely different properties.

We have defined two sets of moves at OR nodes: OR1 is constituted by the liberties of the string to capture only. OR2 is constituted by all the moves worth trying, including the liberties of the string to capture, the liberties of the liberties of the string to capture and the liberties of the adjacent strings that have less liberties than the string to capture.

Similarly, we have defined two sets of moves at AND nodes : AND1 is constituted by the ip1 and ip2 moves, AND2 is constituted by the ip1, ip2 and ip3 moves.

There are different orders in which the widening can be performed. Each order is called a widening strategy, each widening strategy has a number and a name:

1. OR2-2AND2-2: This is the original non-widening, iterative deepening Null Window Search algorithm. The OR2 set of moves is used at OR nodes, and the AND2 set of moves is used at AND nodes.
2. OR1-2AND2-2: The algorithm begins with the OR1 and AND2 sets of moves, and if the search fails, it searches again with the OR2 and AND2 sets of moves.
3. OR2-2AND1-2: The algorithm begins with the OR2 and AND1 sets of moves, and if the search fails, it searches again with the OR2 and AND2 sets of moves.
4. AND1-2OR1-2: The algorithm begins with the OR1 and AND1 sets of moves, and if the search fails, it searches again with the OR1 and AND2 sets of moves. If the search fails again, it searches again with the OR2 and AND2 sets of moves.
5. OR1-2AND1-2: The algorithm begins with the OR1 and AND1 sets of moves, and if the search fails, it searches again with the OR2 and AND1 sets of moves. If the search fails again, it searches again with the OR2 and AND2 sets of moves.
6. ORAND1-2: The algorithm begins with the OR1 and AND1 sets of moves, and if the search fails, it searches again with the OR2 and AND2 sets of moves.
7. OR1-2ANDOR1-2: The algorithm begins with the OR1 and AND1 sets of moves, and if the search fails, it searches again with the OR2 and AND1 sets of moves. If the search fails again, it searches again with the OR1 and AND2 sets of moves. If the search fails again, it eventually searches with the OR2 and AND2 sets of moves.
8. ORAND1-2AND1-2: The algorithm begins with the OR1 and AND1 sets of moves, and if the search fails, it searches again with the OR1 and AND2 sets of moves. If the search fails again, it searches again with the OR2 and AND1 sets of moves. If the search fails again, it eventually searches with the OR2 and AND2 sets of moves.
9. Brute Force : The algorithm always tries all the possible moves. It is intended to compare our selective search algorithm based on game definition with a brute force approach. Note that our quiescence search is responsible for most of the problems solved by the brute force approach.

6 Experimental Results

This section gives experimental results on a standard test set for capturing strings in Go: we call them ggv1 [Kano, 1985a], ggv2 [Kano, 1985b] and ggv3 [Kano, 1987]. These books are regarded by Go players as a well balanced collection of problems that cover the essential problems that arise in real games. The first book contains very simple beginner's problems, the second book requires more knowledge of the game, and the third book contains problems that average players can find interesting. We have selected all the problems involving a capture of a string, including semeai and some connection problems. There are 114 capture problems in ggv1, 144 in ggv2 and 75 in ggv3. Experiments were performed on a Pentium III 600 MHz microprocessor.

Problems are counted as solved only when the search returns Won (in some case it may be useful to consider the preventing moves associated to the Unknown value as they are not refuted due to a lack of search, but may be preventing moves, however the brute force algorithm returns Unknown for many bad moves, therefore they should not be considered as correct answers).

A maximum processing time is set for each search, as soon as the time has elapsed, the search is stopped, and the status is set to 0 (Unknown) on remaining leaves. Two searches are performed for each problem, one with Black playing first, the other with White playing first.

For each book and each maximum processing time per search, a table gives the widening strategy used, the time in seconds used to search all the problems, the number of

nodes including leaf nodes, the minimum number of nodes (when the best move is always tried first at each node), and the percentage of solved problems.

	Time	Nodes	Minimum	Solved
1	0.63s	4404	4017	99.12%
2	0.54s	2123	1857	99.12%
3	0.45s	4268	3844	99.12%
4	0.65s	2484	2299	99.12%
5	0.70s	2494	2279	99.12%
6	0.48s	2036	1848	99.12%
7	0.80s	2430	2257	99.12%
8	0.79s	2582	2404	99.12%
9	26.40	890256	761161	78.07%

Table 1. Results for ggv1, Search Time < 1 second.

	Time	Nodes	Minimum	Solved
1	11.39s	50968	41751	88.19%
2	9.68s	36619	29909	89.58%
3	7.44s	49304	41479	89.58%
4	10.25s	38476	32132	88.89%
5	8.52s	45365	37788	89.58%
6	7.84s	32102	26588	89.58%
7	9.19s	46859	38785	88.89%
8	10.40s	46540	38244	88.19%
9	102.09s	4533217	4275396	30.56%

Table 2. Results for ggv2, Search Time < 1 second.

	Time	Nodes	Minimum	Solved
1	11.86s	52173	42140	80.00%
2	8.73s	32837	27412	84.00%
3	6.38s	37953	32335	84.00%
4	9.11s	33860	28402	84.00%
5	6.79s	34706	29401	84.00%
6	7.38s	30783	24931	84.00%
7	7.39s	34140	30159	84.00%
8	8.28s	37723	33325	84.00%
9	53.28s	2259720	2140103	29.33%

Table 3. Results for ggv3, Search Time < 1 second.

	Time	Nodes	Minimum	Solved
1	4.40s	22782	20634	75.00%
2	3.63s	13297	11745	83.33%
3	2.92s	25107	22355	85.42%
4	3.75s	12834	11834	84.03%
5	3.41s	16183	14681	85.42%
6	3.02s	13234	11770	84.03%
7	3.40s	15988	14467	86.81%
8	3.58s	14190	12973	85.42%
9	11.20s	488576	465145	25.00%

Table 4. Results for ggv2, Search Time < 0.1 second.

We have also tested the algorithm with a lower maximum processing time, closer to current limitations of Go programs, the one second limit may be interesting for programs that spend more time on tactical analysis than on global search. It can also figure the possible improvements due to search in the near future as computers get faster. Each search was stopped as soon as it took more than 100 ms. The results for ggv1 are not given as they are similar to table one, except for the brute force algorithm that solves 74.56% of the problems in 3.21s and 142276 nodes. The results on more complex problems is different, as shown in tables four and five.

	Time	Nodes	Minimum	Solved
1	3.00s	13005	11474	65.33%
2	2.78s	10776	10050	69.33%
3	2.04s	18817	17207	78.67%
4	2.78s	11681	10862	69.33%
5	2.39s	14023	12909	77.33%
6	2.39s	9626	8858	73.33%
7	2.55s	13280	12328	74.67%
8	2.99s	13439	12730	69.33%
9	5.87s	313009	292004	24.00%

Table 5. Results for ggv3, Search Time < 0.1 second.

The brute force approach is clearly much worse than all other strategies. Almost all the problems it can solve are solved by our quiescence search. Considering than the problems in our test set are setup on small boards (9x9 or 13x13 boards), it would be even much worse in a full Go playing program (19x19 board).

Many widening strategies gives speed-ups compared to the original algorithm, except for some very simple problems of volume one where some widening strategies slightly increase the solving time which is more than compensated by more difficult problems. All the widening strategies both decrease the time to solve problems and increase the percentage of solved problems on average. They do not only reduce significantly the computation time, they also solve more problem than the original non iterative widening algorithm (OR2-2AND2-2).

In the original non widening algorithm, the liberties of the string are tried first, and the order of the moves at each node is the same as in the iterative widening algorithm, therefore the observed speed-ups are due to the iterative widening, not to another factor such as move ordering. Moreover, the number of nodes is close enough to the minimum number of nodes to consider that the move ordering is not bad. And even with the perfect move ordering, we can see that iterative widening is still clearly better than the non widening algorithm.

The OR2-2AND1-2 strategy is quite efficient and it is domain independent. There is no widening at OR nodes, and the widening at AND nodes is performed only by the selection of some specified game definitions functions. As game definition functions can be defined similarly for many games, this widening strategy is both effective and general.

We can also observe in table four and five that the OR2-2AND1-2 strategy searches more nodes that the non

widening algorithm, but only takes two third of its time and solves significantly more problems. This apparent anomaly may be due to the cost of the ip3 game definition function, that is higher that the cost of the ip1 and ip2 functions. Therefore searching less nodes using the ip3 function can take more time than searching more nodes only using the ip2 function. In other experiments [Cazenave, 2000], we have already shown that the non widening algorithm based on the ip3 function solves more problems in much less time than the same selective Alpha-Beta Null-Window Search algorithm that uses the same set of moves, without performing the ip3 tests.

We performed tests in another game to assess the generality of iterative widening. We chose Phutball [Conway & al., 1982], as it also falls in the class of games that benefit from theorem proving and selectivity. Other games in this class are Gomoku, mate search in Chess and Hex for example. Generally these games have a high branching factor and a simple and well defined goal. Phutball is played on a Go board, a black stone represents the ball, and players are represented by white stones. A player tries to put the ball on the first line of its opponent. A move consists in moving the ball by jumping over a player, or in putting a new stone on an empty intersection. In Phutball, using iterative widening with the OR2-2AND1-2 combination enabled a speed-up by a factor greater than two.

In these experiments, the transposition table was completely initialized before each widening. It would be more clever to keep the same transposition table, and to put a flag on the transpositions, memorizing the widening step of the transposed board, in order to reuse the information from the previous and narrower search so as to save computation time. Another optimization is to reuse the stored score at OR nodes to update alpha. The results of the experiments that use the enhanced transposition tables, performing the two previously mentioned optimizations, are given in table 6. It appears that it enables to solve roughly 1% more problems in 5/6th of the time. The OR2-2AND1-2 (3rd) widening strategy is used.

Book	MaxTime	Time	Nodes	Minimum	Solved
ggv1	1s	0.30s	4469	4342	99.12%
ggv2	1s	7.48s	63266	58024	88.89%
ggv3	1s	5.10s	43166	38909	84.00%
ggv1	0.1s	0.30s	4469	4342	99.12%
ggv2	0.1s	2.41s	29038	27159	86.81%
ggv3	0.1s	1.74s	21822	20225	80.00%

Table 6. OR2-2AND1-2 with an enhanced transposition table

7 Conclusion

Gradually widening the sets of moves in some complex game search trees enables to reduce significantly the search time when performing an iterative deepening Null Window Search. It also appears that more problems can be solved by using this technique. A general widening strategy relevant to many complex games such as Go, Phutball, Hex and Chess has been experimentally proven useful. It consists in iteratively increasing the order of the game definitions functions used to select forced moves at the AND nodes of the search trees. Results are slightly better when reusing transposition table information from the previous and less wide search.

Acknowledgements

Thank you to J. Méhat and J.-P. Vesinet for proof reading.

References

[Cazenave, 1996] Cazenave T.: *Système d'Apprentissage par Auto-Observation. Application au Jeu de Go*. Ph.D. diss., Université Paris 6. 1996.

[Cazenave, 1998] Cazenave T.: *Metaprogramming Forced Moves*. Proceedings ECAI98 pp. 645-649, Brigthon, 1998.

[Cazenave, 2000] Cazenave T.: *Abstract Proof Search*. Proceedings of CG2000. Published in LNCS 2001.

[Conway et al., 1982] Conway J., Berlekamp E., Guy R.: *Winning ways*, Tome 1 and 2, Academic Press, 1982.

[Ginsberg and Harvey, 1992] Ginsberg M. L., Harvey W. D.: *Iterative Broadening*. Artificial Intelligence 55 (2-3), pp. 367-383. 1992.

[Kano, 1985a] Kano Y.: *Graded Go Problems For Beginners. Volume One.* The Nihon Ki-in. 1985.

[Kano, 1985b] Kano Y.: *Graded Go Problems For Beginners. Volume Two.* The Nihon Ki-in. 1985.

[Kano, 1987] Kano Y.: *Graded Go Problems For Beginners. Volume Three.* The Nihon Ki-in. 1987.

[Korf, 1990] Korf R. : *Depth-first iterative-deepening : An optimal admissible tree search*. Artificial Intelligence 27, N°1, pp 97-109, North-Holland 1990

[Marsland and Srimani, 1986] Marsland T. A., Srimani N.: *Phased State Space Search*. ACM/IEEE Fall Joint Computer Conference, Dallas, Nov. 1986, pp 514-518.

[Marsland and Björnsson, 2000] Marsland T. A., Björnsson Y.: *From Minimax to Manhattan*. Games in AI Research, pp. 5-17. Edited by H.J. van den Herik and H. Iida, Universiteit Maastricht. ISBN 90-621-6416-1. 2000.

[Meseguer and Walsh, 1998] Meseguer P., Walsh T. : *Interleaved and Discrepancy Based Search.* Proceedings ECAI98 (ed. H. Prade). John Wiley & Sons Ltd., Chichester, England. ISBN 0-471-98431-0. 1998.

[Schaeffer, 1989] Schaeffer J.: *The History Heuristic and Alpha-Beta Search Enhancements in Practice.* IEEE Transactions on Pattern Analysis and Machine Intelligence 11, N° 11, pp. 1203-1212, 1989.

[Smith and Nau, 1994] S.J.J. Smith and D.S. Nau. *An Analysis of Forward Pruning.* In *AAAI-94*, 1994.

Temporal Difference Learning Applied to a High-Performance Game-Playing Program

Jonathan Schaeffer, Markian Hlynka
{jonathan, markian}@cs.ualberta.ca
Department of Computing Science
University of Alberta
Edmonton, Canada T6G 2H1

Vili Jussila
vili@cc.hut.fi
Laboratory of Computational Engineering
Helsinki University of Technology
Helsinki, Finland

Abstract

The temporal difference (TD) learning algorithm offers the hope that the arduous task of manually tuning the evaluation function weights of game-playing programs can be automated. With one exception (*TD-Gammon*), TD learning has not been demonstrated to be effective in a high-performance, world class game-playing program. Further, there has been doubt expressed by game-program developers that learned weights could compete with the best hand-tuned weights. *Chinook* is the World Man-Machine Checkers Champion. Its weights were manually tuned over 5 years. This paper shows that TD learning is capable of competing with the best human effort.

1 Introduction

The most time-consuming aspect of building a high-performance game-playing program is the design, implementation and tuning of the evaluation function. Designing the knowledge-based features in the evaluation function and implementing them in a fast, efficient manner remains a difficult task for humans, although there have been some limited successes at automating this task [Buro, 1995; van Rijswijck, 2001; Fawcett and Utgoff, 1992]. Historically, tuning the evaluation function—adjusting the weight (importance) of each feature contributing to the evaluation—has been a tedious, manual task. There have been numerous attempts to automate this (for example, [van der Muelen, 1989; Anantharaman, 1991]), but none of these techniques achieved the requisite high performance. Buro has achieved impressive results using linear regression in his Othello program [Buro, 2001], but it is not clear that similar techniques will work for a broader class of games.

Temporal difference (TD) learning has emerged as a powerful reinforcement learning technique for incrementally tuning parameters [Sutton and Barto, 1998]. Tesauro applied TD learning to tune the weights of a neural net, in the process building a world class backgammon program (*TD-Gammon*) [Tesauro, 1995]. For several years, this remained an isolated success story in the games literature, as the conditions in backgammon that appeared to favor TD learning did not exist in other high profile games, such as chess. In 1997, the TDLeaf algorithm was introduced (TD learning applied to minimax search) [Beal, 1997] and it achieved some success with chess (*KnightCap* [Baxter *et al.*, 1998a; 1998b; 2000]).

In none of the above cases has it been possible to compare the performance of TD learning to that of the best-tuned human weights. *TD-Gammon* learned through self-play; a human-tuned version of the program does not exist. *KnightCap* learned through playing speed chess against humans on the Internet. A human-tuned version of the program does exist, but both it and the TD version of the program are far below grandmaster level in strength. Also, tuning for speed chess is not necessarily representative of what needs to be learned for tournament chess (where the search depths are greater). In all the examples of TD learning applied to games, there has been a nagging question: Can TD-learned weights be successful in strong (world-championship-calibre) game-playing programs? For challenging games, such as chess, game developers have expressed doubt that tuned weights would be sufficient to achieve the highest levels of performance.

Chinook is the World Man-Machine Checkers Champion [Schaeffer, 1997]. Its evaluation function weights were tuned manually over a period of 5 years. They were extensively tested both in self-play games and in hundreds of games against top human players (including playing 96 games for the World Checkers Championship). This paper investigates whether the tuning of evaluation function weights in *Chinook* can be replaced by TDLeaf learning. The experimental data indicates that the answer is "yes", as well as giving new insights into TD learning in game-playing programs. This is the first known attempt to conduct a detailed study that compares hand-tuned and TD-trained weights in an established high-performance game program.

2 Temporal Difference Learning

Temporal difference learning is an unsupervised reinforcement learning algorithm [Sutton and Barto, 1998]. It learns from experience without a model of the en-

vironment's dynamics, and updates its estimates based on other, as yet unconfirmed, estimates. Thus, TD can learn without waiting for a final outcome on a given task; it evaluates the sub-steps between evaluations.

The TD(λ) algorithm can be succinctly expressed as follows [Sutton and Barto, 1998]. Given a series of predictions, $P_0...P_{t+1}$ (search results from a game in this context), then the weights in the evaluation function can be modified as follows:

$$\Delta w_t = \alpha(P_{t+1} - P_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k \quad (1)$$

The change in weights (Δw_t) depends upon the predictions (P_k) and the gradient of the predicted value of the kth state with respect to the weights (∇P_k).

The λ term is a decay-rate parameter. It determines the extent to which learning is affected by subsequent states. A λ of zero is equivalent to learning only from the next state. A λ of 1 indicates learning only from the final reinforcement signal; in the case of a game, the final won/lost assessment. α is a step size parameter: the proportion of adjustment to allow on each iteration. Thus, the λ parameter determines whether the algorithm is applying short or long range prediction, while α determines how quickly this learning takes place.

TD(λ) is a proven algorithm for reinforcement leaning. One of its important advantages is that it can be computed incrementally. However, to apply it to problems utilizing search, some refinements are required. The TDLeaf algorithm is essentially TD learning applied to minimax search. TDLeaf was originally implemented by Beal and Smith [Beal, 1997], though not under the name TDLeaf (which is attributed to [Baxter *et al.*, 1998a; 1998b]). The crux of the algorithm is *not* to use the position at the root of the search tree to tune the search. Instead, tuning takes place using the position of the leaf node at the end of the principal variation of the search. The principal variation is the line of best play; the position at the end of this line of play has had its value backed-up to the root of the search.

TDLeaf was implemented in the chess program *KnightCap* [Baxter *et al.*, 1998a; 1998b]. Baxter *et. al* report that the program's chess rating rose from 1650 to 2150 in three days (308 games). While this sounds impressive, there are a few caveats that need to be mentioned. First, the results were achieved at speed chess; there is no indication that these results will apply to (slower) over-the-board chess. Second, the learning plateaued well before achieving a high level of play. Finally, despite the early promise of TDLeaf, no one has demonstrated that it can out-perform the best set of human-tuned weights. Many researchers active in the computer games community (including the first author) have publicly doubted that TD learning is capable of achieving the high level of performance required in a game-playing program.

3 Training

Chinook's evaluation function is the linear combination of 23 knowledge-based features for each of 4 game phases. Two features cannot be modified (the value of a checker and the value of a king) because of some search code dependencies. Hence, a total of 84 parameters need to be tuned.[1]

Chinook supports an integer evaluation function and integer weights. TD learning is inherently a real-number task. Thus, *Chinook* was modified to accept floating point values. However, the final position evaluation would be converted to an integer, allowing these changes to be restricted to the evaluation function.

Chinook and the TD learning (TDL) were kept as separate programs which communicated with each other using text files. The file *Chinook* reads in includes information about the opening sequence, search depth, number of turns to play, and the weights for both sides. After a game finishes, *Chinook* outputs a file containing the result of the game and evaluations for each weight component. TDL uses this file to adjust the weights and then starts a new game with the revised weights. During this process TDL also saves information about the progress of the learning, such as the results of each played game and the value of the weights after each game. Because not all weights are updated every turn, we also record the frequency at which weights are modified to discover if some game situations happen so seldomly that the corresponding weight does not get much training.

The TDLeaf algorithm in TDL operates on pairs of moves. The weights of move i are updated based in part upon the evaluation at move $i+1$. Not all pairs of moves were candidates for TDLeaf updating. Capture moves are forced in checkers, so if only one move is legal in a position, no updating would occur. Also, occasionally *Chinook*'s search algorithm was incapable of recovering the principal variation as far as the leaf node. In this case, TDLeaf could not be applied.

The training routine was as follows:

1. *Chinook* is used to play two weight files against each other.
2. TDL modifies one or both of the weight files based on the game played.
3. This procedure is iterated until learning is seen to plateau (typically before 10,000 iterations).

To prevent the programs from playing the same moves in every game, an opening book was used which included the standard 144 checkers openings, each 3 ply long. The learning rate α was chosen to be 0.01 and the TD parameter λ was set to 0.95. These values were chosen based on the *KnightCap* experience, but finding the best settings remains an open question.

Several different approaches were attempted for learning. Each experiment involved starting with all weights set to zero, train using TD learning, and then evaluate

[1] Note that the 21 tunable features are each the result of a function that itself may contain many parameters. These lower-level parameters are not addressed in this paper.

the learned weights by using them in a match against the tournament version of *Chinook*.

The first approach involved training the weights by playing against tournament *Chinook* (*teacher learning*). The goal was to determine how effective the learning was given the benefit of a high-performance teacher. The second set of tests involved self-play (*self-play learning*). Here the goal was to see if the learning could boot-strap itself to achieve high performance. In both cases, separate experiments were performed using 5, 9, and 13-ply searches, generating separate weights for the black and white sides.

Examining the output of a training session shows that the performance of the learned weights against tournament *Chinook* rapidly improves at the beginning of the session due to the poor starting values. After this initial period, the rate of improvement slows until at roughly 4,000 games a stable state is reached.[2] In the experiments, only 84 weights had to be learned. In contrast, *KnightCap* had to learn 1,500 parameters in its first set of experiments. This was later expanded to 6,000 parameters [Baxter *et al.*, 1998a]. The small number of parameters used in *Chinook* accounts for the relatively fast learning phase.

4 Results

Trained weight sets were tested against the tournament version of *Chinook*. Evaluation consisted of a 288-game match (each program playing both sides of the 144 openings). All versions of the program used *Chinook's* 6-piece endgame databases. Tournament *Chinook* has no knowledge of how to play simplified endgame positions because it assumes that the database will always be used. Using the databases had the benefit of speeding up the experiments since, once a position with 6 or fewer pieces was reached, the databases would give the final result of the game, thereby ending the game.

4.1 Baseline

How important are the evaluation function weights? The obvious way to answer this question is to set all the weights to zero and see how the program performs. In effect, this "zero knowledge" program uses only material for its evaluation. The result of the match is not surprising: a 34–254 game loss to tournament *Chinook* with 15-ply searches (the endgame database knowledge salvaged many draws). Since both programs used the same search depth, the quality of the knowledge is solely responsible for the match score. As an additional data point, all the weights were set to one. Now the program "knows" how to evaluate a position, but it does not understand the relative importance of each feature. Having some knowledge is obviously beneficial, as this program loses by a smaller margin (an average score of 94.5–193.5).

[2] *KnightCap* required fewer training games, but its performance levels off at a playing strength that is considerably below world-championship caliber.

4.2 Teacher Learning

Figures 1a, 1b, and 1c shows the performance of white and black weights that were trained using 5, 9, and 13-ply searches, respectively. The x-axis shows the search depth used for the evaluation, and the y-axis shows the number of wins minus losses from the learning program's point of view.

The 5-ply-trained weight set does well against *Chinook* when playing games with a search depth of 5 ply, but performance quickly tapers off as the programs play games using larger search depths (Figure 1a). A similar pattern is seen with the 9-ply-trained weights (Figure 1b). The experiment shows the learned weights defeating *Chinook* in matches up to 9-ply, but tapering off with deeper searches. Whereas with 5-ply searches, the results of training using the white positions dominates those for the black positions, with 9-ply searches the difference between the two sets of weights essentially disappears.

For the 13-ply results (Figure 1c), the data is not as clear. As before performance seems strong around the training search depth (13-ply) and there is the suggestion that it is beginning to taper off for deeper searches (it would take several weeks to get the 17-ply data). Unlike the previous graphs, the performance of the weights using search depths shallower than the training depth are poorer. However, the difference between the 7-ply and 13-ply results in Figure 1c represents only a 7% improvement, well within the statistical variability expected.

The graphs reveal an important insight for anyone using TD learning in game-playing programs: the weights must be trained using the depths of search expected to be seen in practice. Deeper searches provide a more accurate approximation of the root position's true value for the TD algorithm to learn. This suggests that the *KnightCap* weights that were obtained using speed chess will not perform well in slower tournament chess (similarly, [Anantharaman, 1991] needs deeper searches to be effective). In effect, there is no free lunch; you can't use shallow search results to approximate deep results.

We experimented with creating separate weight sets for playing white and black. The purpose was to see if the specialization of the weight sets could lead to better play, given that white generally has an opening advantage. Surprisingly, using the black weights only when playing black, and the white weights only when playing white, does not seem to be statistically significantly better in our experiments. After the opening phase of the game, the resulting types of positions seen are similar for white and black, resulting in a similar learning experience. This is more pronounced with deeper searches (since the search can see "beyond" the opening) than it is with shallower searches. This would account for the large difference between the white and black performance in Figure 1a.

The previous experiments have not been entirely fair. Both the training and evaluation was done using the same 144 starting positions. The good results for the learned weights might be a consequence of the program

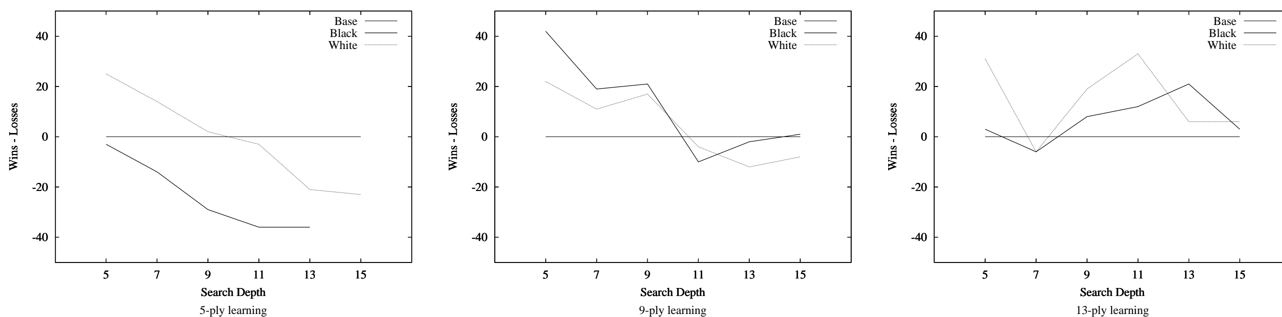

Figure 1: Teacher learning: a) 5-ply, b) 9-ply and c) 13-ply.

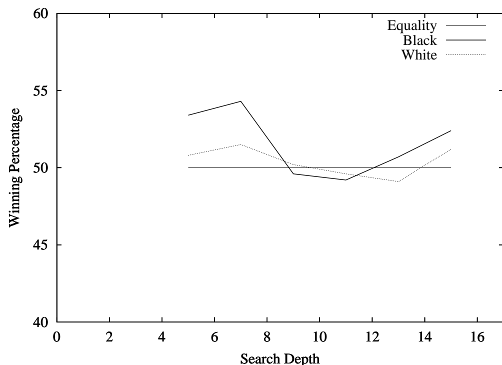

Figure 2: 786 game matches using 13-ply learning.

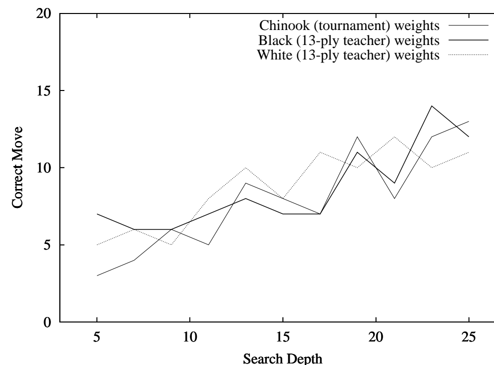

Figure 3: *Chinook* test set data.

being trained to play the same opening positions that are used in the evaluation. To get another indicator of the performance of the trained data, a second experiment was performed. From a collection of games played by former world champion Marion Tinsley, all positions 8-ply into the games were extracted (393 positions). These positions were used as the openings for a 786-game match between the learned weights and tournament *Chinook*.

Figure 2 shows the results for both the 13-ply white and black trained weight sets, expressed as the percentage of total points scored (over 786 games). At deeper search depths, the tuned weights perform slightly better than the hand-tuned weights, although the difference is not statistically significant. Given the match length, it is safe to say that the performance of the TD weights is comparable to that of the best hand-tuned effort.

4.3 Self-Play Learning

In this set of experiments, the program learned through self-play without the benefit of having a strong opponent to train against. All the self-play results analyzed to date are consistent with that seen in the previous section, with the exception that the training takes longer to plateau. The 13-ply-trained black weight sets scored 50.2% of the points in a 786-game match (using 15-ply searches) with tournament *Chinook*, while the white weights scored 48.3%.

The self-play data strongly indicates that a good teacher is not needed for the program to learn a set of evaluation function weights that achieves world-championship-calibre performance. This is wonderful news for game-program developers, as it suggests that manual weight tuning may be a thing of the past.

The *KnightCap* self-play results are not as good as those reported here. This is likely a consequence of the number of parameters being tuned; fewer parameters are easier to fit.

4.4 Additional Data

There is a test set of 19 positions (taken from *Chinook* games) that have proven to be particularly difficult for the program to solve. In these positions, the opponents (mostly humans players) demonstrated profound insights into the game that *Chinook*, at the time the game was played, could not match. None of the positions is easily resolved by search; the quality of the knowledge is the critical factor. Most of these positions were the motivation for adding additional features to the evaluation function and/or making major changes to the feature weights. During the development of the program, these positions were often used to benchmark the program.

Chinook has been tested on these positions using three weight sets: original, white teacher training at 13-ply, and black teacher training at 13-ply. The results are shown in Figure 3. For each of the positions, the program versions searched 5-ply to 25-ply deep (in increments of 2 ply). The figure records which search depths produced the correct solution to the positions. Note the general trend that increased search depth results in more frequent correct solutions. However, in most of the positions, the programs get the correct answer at the end

of an iteration, only to switch to a different move on the next iteration. All versions tested were indecisive in their move choice for most of the positions (a further indication that the positions are indeed still very hard for *Chinook*). Both TD weight sets perform comparably to the original weight set in *Chinook*. There is nothing to suggest that one weight set is significantly better than the others.

4.5 Comments

One must caution that most of the experimental results have been obtained from machine-versus-machine games.[3] The results may be different in machine-versus-human play. Unfortunately, with *Chinook* retired and the program significantly stronger than all human players, there are no opportunities to evaluate just how good the weights are in play against humans.

Although TD learning promises to reduce the effort to build a high-performance game-playing program, deciding on the evaluation function features still remains largely a manual chore. Some of the features in *Chinook*'s evaluation function came as the result of extensive human analysis of the program's play to identify deficiencies in the program's knowledge. Once a new feature was added to the program, then the manual tuning would begin again. TD learning makes this a less painful process. The human identifies and adds the new knowledge; the program learns the new weight set.

5 Examining the Weights

Table 1 shows *Chinook*'s original weights and those learned from the white positions with 13-ply searches. Not unexpectedly, there are some major differences:

1. Several of the features occur rarely in certain phases of the game and, hence, the computer-generated weights may be off (or irrelevant) because of insufficient training. For example, "free king", "king center" and "loose checker" are mainly endgame features. Over 8,533 games (a total of 238,0403 learning updates), in phase 1 these features occurred only 177, 16, and 184 times, respectively.

2. "Value of move" is a small bonus given to the side whose turn it is to move. In most positions, from a human's point of view, having the right to move is a small advantage. From the computer's point of view, value of move is just a constant added to the evaluation function. The negative value for this weight suggests that, in general, the evaluation scores obtained using trained weights are a bit high, and this feature is being used to make a small linear adjustment to the value to get a better fit.

3. The mobility terms are the most important part of *Chinook's* evaluation function, after material balance. The computer-generated weights are comparable to the human weights in that they generally have the same sign and similar magnitudes.

4. The terms "frozen", "dog hole", "loose men", "d2e7", and "free king" were late additions to the evaluation function. These terms were added to address problems that arose in play against human players. Both human and machine weights are affected by the infrequency with which these features occur. It is also likely that these features are not as common in machine-versus-machine play as they are in human-versus-machine play.

5. The biggest surprise is the difference in value for "trapped kings" (kings that are immobile in corners and cannot be freed). This is a symptom of the above problem. Against computers, some humans play for a trapped king since, historically, that was a major weakness in computer play (and, indeed, was a problem with early versions of *Chinook*). The evaluation function detects this situation and penalizes it heavily. However, since the training is from self-play, *Chinook* never plays to "dupe" *Chinook* into trapping its king. Consequently, the TD-learning infrequently sees this feature arising and, when it does, it is usually not a position where this is the decisive factor.

The lesson here is that play against human players is necessary to complete the training. Humans have their own set of biases, predilections, and notion of "good" and "bad". The additional training will be most pronounced in the weights of the features that infrequently occur in machine-versus-machine play.

Despite the radical differences between the TD-learned and the human-tuned set of weights, one cannot dispute the success of each version. On the one hand, it is remarkable that TD learning is as successful as it is given that the learning is based solely on game-play feedback with no human intervention. On the other hand, it is a triumph of human cognitive abilities that the human solution to a complicated optimization problem can indeed be competitive with a computer solution. The final result, that the human-tuned solution and the TD-tuned solution are roughly equivalent in performance, reflects well on both man and machine.

6 Conclusions

There are two parts to an evaluation function: the function terms and the weighting of these terms. This paper strengthens the case that TD learning provides an effective solution to the latter problem. Learned weights can compete with (and perhaps exceed) the performance of the best hand-tuned weights in a high-performance game-playing program.

TD learning opens up new opportunities for improving a program's abilities. For example, the program could have a different set of weights for each opening, or for different classes of positions. Different weights could be used based on the expected depth of search. In addition, the program developer can experiment with new features, and let the learning algorithm decide what is relevant. None of this would be practical if these weights had to be tuned manually.

[3] [Berliner *et al.*, 1990] mentions the pitfalls that can arise from basing conclusions solely on self-play games.

	Original Weights				Learned Weights			
Name	1	2	3	4	1	2	3	4
Value of move	4	3	3	2	-2.30	-6.94	-2.48	0.46
Free mobility	1	2	3	4	3.40	6.50	2.77	6.47
Some mobility	-4	-6	-8	-10	0.89	-4.62	-8.97	-6.08
Recapture mobility	3	3	3	3	-1.72	2.33	5.77	3.25
No-move mobility	-1	-1	-2	-4	-2.13	-4.45	-4.17	-1.30
Exception mobility	0	0	0	0	-0.47	0.89	5.56	2.47
Double-cap mobility	-6	-6	-6	-6	-0.15	-1.45	-2.34	-1.08
Balance	5	4	3	2	1.14	4.53	1.35	-0.86
Advancement	-1	0	0	0	3.59	-3.54	-0.39	-0.68
Centrality	2	2	1	0	-1.91	8.44	1.25	-1.86
Angle	1	1	0	0	0.79	3.26	3.24	3.23
Back row	4	3	3	2	1.93	8.75	11.77	6.28
Shadow	3	2	1	0	0.74	4.05	1.23	-0.19
Trapped king	32	32	32	32	-0.01	-0.02	0.90	0.73
Loose checker	5	5	5	5	0.03	1.38	4.76	3.72
King center	3	3	3	3	-0.01	0.89	5.65	5.94
D2E7	3	2	1	0	0.09	-0.06	0.22	0.57
Free king	20	20	20	20	0.38	1.66	5.37	3.69
Dog-hole	5	5	5	5	-0.08	0.16	1.66	0.89
Loose men	15	15	15	15	0.25	1.88	5.33	5.56
Frozen	10	10	10	10	0.00	0.05	-0.09	-0.12

Table 1: Comparing weights.

The dream of creating a games engine that can achieve high performance for any board game is one step closer to reality. High-performance black box search engines now exist (e.g. [Brungger *et al.*, 1999; Romein, 2000]), as well as generic (mediocre performance) games engines (see www.zillionsofgames.com). The last piece of the puzzle, automatically discovering the features needed for the evaluation function, remains elusive.

7 Acknowledgments

Financial support was provided by NSERC and iCORE.

References

[Anantharaman, 1991] T. Anantharaman. *A Statistical Study of Selective Min-Max Search*. PhD thesis, Carnegie Mellon University, 1991.

[Baxter *et al.*, 1998a] J. Baxter, A. Tridgell, and L. Weaver. Experiments in parameter learning using temporal differences. *ICCA Journal*, 21(2):84–99, 1998.

[Baxter *et al.*, 1998b] J. Baxter, A. Tridgell, and L. Weaver. KnightCap: A chess program that learns by combining TD(λ) with game-tree search. *ICML*, pages 28–36, 1998.

[Baxter *et al.*, 2000] J. Baxter, A. Tridgell, and L. Weaver. Learning to play chess using temporal differences. *Machine Learning*, 40(3):243–263, 2000.

[Beal, 1997] D. Beal. Learning piece values using temporal differences. *ICCA Journal*, 20(3):147–151, 1997.

[Berliner *et al.*, 1990] H. Berliner, G. Goetsch, M. Campbell, and C. Ebeling. Measuring the performance potential of chess programs. *Artificial Intelligence*, 43(1):7–21, 1990.

[Brungger *et al.*, 1999] A. Brungger, A. Marzetta, K. Fukuda, and J. Nievergelt. The parallel search bench ZRAM and its applications. *Annals of Operations Research*, 90:45–63, 1999.

[Buro, 1995] M. Buro. Statistical feature combination for the evaluation of game positions. *JAIR*, 3:373–382, 1995.

[Buro, 2001] M. Buro. Improving heuristic mini-max search by supervised learning. *Artificial Intelligence*, 2001. To appear.

[Fawcett and Utgoff, 1992] T. Fawcett and P. Utgoff. Automatic feature generation for problem solving systems. *ICML*, pages 144–153, 1992.

[Romein, 2000] J. Romein. *Multigame – An Environment for Distributed Game-Tree Search*. PhD thesis, Vrije Universiteit, 2000.

[Schaeffer, 1997] J. Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer Verlag, 1997.

[Sutton and Barto, 1998] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[Tesauro, 1995] G. Tesauro. Temporal difference learning and TD-Gammon. *CACM*, 38(3):58–68, 1995.

[van der Muelen, 1989] M. van der Muelen. Weight assessment in evaluation functions. *Advances in Computer Chess 5*, pages 81–89, 1989.

[van Rijswijck, 2001] J. van Rijswijck. Learning from perfection (a data mining approach to evaluation function learning in awari). *2nd International Conference on Computers and Games*, 2001. To appear.

Satisficing and Learning Cooperation in the Prisoner's Dilemma

Jeff L. Stimpson
Computer Science Department
Brigham Young University
Provo, UT 84602
jstim@cs.byu.edu

Michael A. Goodrich
Assistant Professor of Computer Science
Brigham Young University
Provo, UT 84602
mike@cs.byu.edu

Lawrence C. Walters
Associate Professor of Public Policy
Brigham Young University
Provo, UT 84602
larry_walters@byu.edu

Abstract

The prisoner's dilemma is a useful model for studying the balance between self-interest and group-interest in multi-agent systems. Although many strategies have been developed that perform well, most of these strategies make strong assumptions about the information available to the agent. It is in this context that we describe a satisficing learning strategy for the prisoner's dilemma and present evidence that stable outcomes other than the Nash equilibrium are possible. In addition, we offer empirical evidence that under typical circumstances, mutual cooperation is the most likely outcome and identify conditions under which two satisficing agents will learn to cooperate.

1 Introduction

In situations involving several interacting agents, each agent is often forced to choose between two types of behavior: those that benefit the group as a whole, and those that lead to rewards for the individual at the expense of the group. The situation becomes interesting when, in the long run, poor outcomes for the group lead to negative consequences for each individual.

The iterated prisoner's dilemma is an elegant and well-known example of such circumstances that has been studied in a wide variety of disciplines. A typical payoff matrix for the prisoner's dilemma is given in Figure 1. The dilemma is

(A's payoff, B's payoff)		Agent B's Choice	
		Cooperate	Defect
Agent A's Choice	Cooperate	(3, 3)	(1, 4)
	Defect	(4, 1)	(2, 2)

Figure 1: A typical payoff matrix for the prisoner's dilemma.

that every pair of actions is either unstable or sub-optimal. Formally stated, the unique Nash equilibrium is the only outcome that is not Pareto optimal. Mutual defection is the dominant strategy in the sense that a player will be better off by defecting regardless of what his or her opponent does. Yet if both players make this "rational" decision to defect, both receive less than if they had cooperated.

In searching for an effective strategy in the prisoner's dilemma, we look for a strategy exhibiting flexible behavior. It should cooperate whenever mutual cooperation is possible, but it must be able to defect when it is apparent that its opponent is unwilling to cooperate. Many such strategies have been developed and studied, but often these strategies involve at least one of the following assumptions:

- players are aware of the structure of the game such as the other players, the other player's possible actions, and the relationship between the actions and the payoffs,
- players are immediately aware of other player's decisions,
- players are aware of the other player's payoffs,
- players are aware that they are in a game situation, meaning that they are aware that the actions of other agents are affecting their outcomes

In computer simulations these requirements are easily met, but in real-world situations they may be quite limiting. For example, the prisoner's dilemma can be extended to multiple players. If there are many players choosing from many actions, keeping track of the game structure may be unrealistic in terms of storage requirements and computational capacity. In other cases, information about the structure of the game may not even be available to the decision maker. Finally, although situations analogous to a prisoner's dilemma are common occurences, they are rarely thought of in terms of game theory. Instead, we are more interested in meeting specific goals.

Removing these assumptions from the prisoner's dilemma takes the problem out of game theory and into areas of machine learning. It is in context of these types of situations that we consider a satisficing strategy for the prisoner's dilemma. Specifically, the purpose of this paper is to present the strategy and then (1) show that stable outcomes other than the Nash equilibrium frequently occur and (2) describe the circumstances under which two agents employing a satisficing strategy will learn to cooperate.

2 Related Literature

The prisoner's dilemma was conceived in the 1950s to question some of the basic tenets of game theory. Standard rational decision mechanisms, such as minimax, lead to mutual defection and poor outcomes for both players. Since then there have been numerous attempts to "solve" the prisoner's dilemma by showing that mutual cooperation is rational after all. The most influential of these has been Axelrod's work in the repeated prisoner's dilemma [1984]. He shows that mutual cooperation is rational and stable when the following conditions hold: (1) the future is important, (2) there is sufficient difference between payoffs for mutual cooperation and mutual defection, and (3) one is facing an adaptive opponent. In summary, Axelrod shows that rationality in repeated-play games is not tantamount to Nash equilibrium.

The idea of applying game theory to learning in multi-agent systems is far from new. For example, Minimax-Q [Littman, 1994] is a reinforcement learning algorithm that learns the Nash equilibrium in zero-sum, or purely competetive, stochastic games. Further work such as [Hu and Wellman, 1998] has attempted to extend the same idea to general-sum stochastic games. Typically, the focus of this literature has been towards learning the Nash equilibrium. While this may be a desirable property in many circumstances, this approach has drawbacks. First, these algorithms usually require significant assumptions and knowledge about the game structure that can be quite limiting. Second, in light of Axelrod's work, in a repeated-play situation, the Nash equilibrium may not be the only stable solution with desirable properties.

Like much of the work done in the prisoner's dilemma, the concept of satisficing came about as a modification of rationality. Traditional rational choice theory holds that an agent faced with a decision will choose the alternative that maximizes a utility function. However, as noted in [Conlisk, 1996] and others, there is little empirical evidence that people make decisions in this manner; indeed evidence strongly suggests otherwise. As a replacement, Herbert Simon has proposed satisficing. He explains the difference between optimizing and satisficing: "A decision maker who chooses the best available alternative according to some criteria is said to optimize; one who chooses an alternative that meets or exceeds specified criteria, but that is not guaranteed to be either unique or in any sense the best, is said to satisfice" [Simon, 1997]. Rather than calculating optimal actions, a satisficing agent simply selects an alternative that meets a set of aspiration levels. As long as these aspiration levels are being met, the agent can continue to act without expending any search costs. When aspiration levels are not met, a search is executed until a satisfactory alternative is found.

In order to handle a variety of environments, aspiration levels can be adaptive. According to Simon, "if it turns out to be very easy to find alternatives that meet the criteria, the standards are gradually raised; if search continues for a long while without finding satisfactory alternatives, the standards are gradually lowered" [Simon, 1997].

We see several advantages in applying satisficing to multi-agent systems. First, because satisficing is simple and flexible, it can be applied when information, storage space, and execution time are limited. This means that agents do not need complex models of other agents. Satisficing is also robust—even if the environment changes (or initial information about the environment is wrong), a satisficing algorithm can typically adapt.

3 A Satisficing Strategy For the Prisoner's Dilemma

Applying Simon's satisficing algorithm to the prisoner's dilemma is straightforward. In this paper, we adapt the algorithm and notation presented in [Karandikar, *et al.* 1998]. The state at time t for an agent using this strategy is given by the pair (A_t, α_t) where A_t is an action in {C, D} and α_t is the current aspiration level. The players' actions determine the payoffs, π_t^A and π_t^B. After receiving a payoff π_t an agent employing a satisficing strategy updates its state in two steps. First, if $\pi_t \geq \alpha_t$ then $A_{t+1} = A_t$, otherwise $A_{t+1} \neq A_t$. Then, aspirations are updated as a weighted average between the current aspiration level and the received payoff. This update rule is given by equation (1) where $0 \leq \lambda \leq 1$.

$$\alpha_{t+1} = \lambda\alpha_t + (1-\lambda)\pi_t \quad (1)$$

It is worth pointing out that the decision algorithm makes no use of the payoff matrix or the actions of the other players. Thus it can be applied to situations where this information is either complex or unknown. All that is needed is the ability to associate a payoff with an action. In addition, it is important to note that this algorithm requires three parameters for each agent: the update rate λ, an initial action A_0 and an initial aspiration α_0.

Before moving into an analysis of the algorithm, a simple illustration is worthwhile. Given that $\lambda = 0.5$, $A_0 = \text{C}$, and $\alpha_0 = 4.0$, consider the example in Figure 2.

t	Tit-for-Tat	A_t	π_t	α_t
0	C	C	3	4
1	C	D	4	3.5
2	D	D	2	3.75
3	D	C	1	2.87

Figure 2: A brief example of a satisficing strategy against a tit-for-tat strategy

In this example, a satisficing agent is playing against a tit-for-tat strategy that simply cooperates on the first move and then repeats its opponent's last move on subsequent iterations. Initially, both players cooperate, receiving a payoff of 3. However, because this payoff is less than the satisficing agent's aspiration of 4, $A_1 = \text{D}$ and the aspirations are updated as an average of the old aspiration and the new payoff.

4 Cooperation Among Satisficing Agents

Before describing our results in detail, we make two observations. First, reinforcement learning has been applied to the prisoner's dilemma with mixed results. In [Sandholm and Crites, 1996], several types of Q-learners were shown to play optimally against a fixed tit-for-tat strategy. However, due to the interaction of their learning, these Q-learners had difficulty playing optimally against each other. Second, although the satisficing algorithm described in the last section is simple, the dynamic interaction between two agents is difficult to theoretically characterize. Thus, in this paper we restrict our analysis to two satisficing agents playing against each other. In addition, we focus on presenting empirical evidence of circumstances under which these two agents will learn to cooperate.

In order to extend the notation to a two-player game, we introduce B_t and β_t as the second player's action and aspiration level respectively. For simplicity, λ is set to the same value for both players. We also generalize the payoff matrix by setting the off-diagonal payoffs to (0,1) and (1,0) and then use σ as the reward for mutual cooperation and δ as the reward for mutual defection with the constraints that $0 < \delta < \sigma < 1$ and $\sigma > 0.5$. This modified payoff matrix is shown in Figure 3.

(A's payoff, B's payoff)		Agent B's Choice	
		Cooperate	Defect
Agent A's Choice	Cooperate	(σ, σ)	(0, 1)
	Defect	(1, 0)	(δ, δ)

Figure 3: Generalized payoff matrix for the prisoner's dilemma

4.1 Convergence and Stability

Before presenting our results, we discuss the possible outcomes of a repeated prisoner's dilemma played by satisficing agents. The simplest outcome is convergence to a pair of actions (A, B). This occurs when $\alpha_t \le \pi_t^A$ and $\beta_t \le \pi_t^B$ meaning that both players are satisfied with their current payoffs and thus both players will repeat their actions indefinitely. At subsequent iterations, α will asymptotically approach π^A and β will asymptotically approach π^B. This can be seen as an equilibrium in the sense that neither player has an incentive to change, given their goals and what they have learned about their environment.

A second possible outcome is convergence to some action cycle, meaning that both players repeat a sequence of action pairs indefinitely. As a formal definition we say that the players have converged to a cycle of duration N at time τ, if for all $t > \tau$, and all k such that $0 \le k \le N-1$, $A_{t+k} = A_{t+k+N}$ and $B_{t+k} = B_{t+k+N}$.

A third and final possibility to consider is that the interaction between two agents is entirely chaotic. This is at least very unlikely, as throughout our research the process has always converged to some stable outcome regardless of the payoff matrix or initial conditions. However, this remains to be shown theoretically.

Figure 4 is a brief illustration of the complexity of the process. It depicts the outcome as a function of the initial aspirations for three possible game structures and initial actions. Clearly there is no simple mathematical characterization of the relationship between game structure and initial parameters and convergence to cooperation. However, empirical results presented in the next section do allow us to identify conditions under which these agents will learn to cooperate.

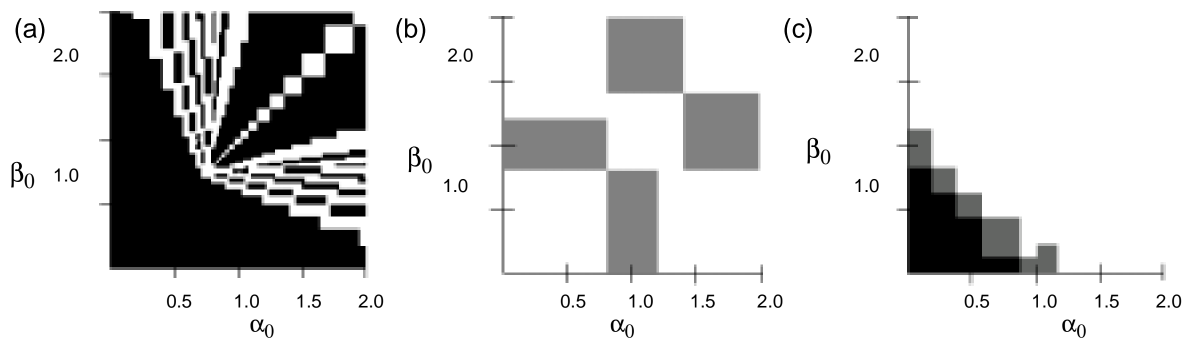

Figure 4: These three graphs show the relationship between initial aspirations and the final outcome for three different game structures. For each pair of initial aspirations in the graph, the outcome of the game was recorded. White indicates convergence to mutual cooperation, black indicates convergence to mutual defection, and gray indicates convergence to some cycle. In Figure 4a, A_0 = D, B_0 = D, σ = 0.8, δ = 0.7, and λ = 0.9. In Figure 4b, A_0 = C, B_0 = C, σ = 0.8, δ = 0.5, and λ = 0.5. In Figure 4c, A_0 = D, B_0 = C, σ = 0.6, δ = 0.5, and λ = 0.8.

4.2 General Results

We set up a simulation that randomly selects the parameters for a game from uniform distributions as described in Table 1.

Parameter	Min. Value	Max. Value
α_0, β_0	0.5	2.0
λ	0.1	0.9
σ	0.51	1.0
δ	0.1	σ
A_0, B_0	50% = C, 50% = D	

Table 1: Distribution of parameters for simulations

The simulation then runs a repeated prisoner's dilemma until the process converges to some action pair or some action cycle. The final outcomes of 5,000 of these simulations are displayed in Figure 5.

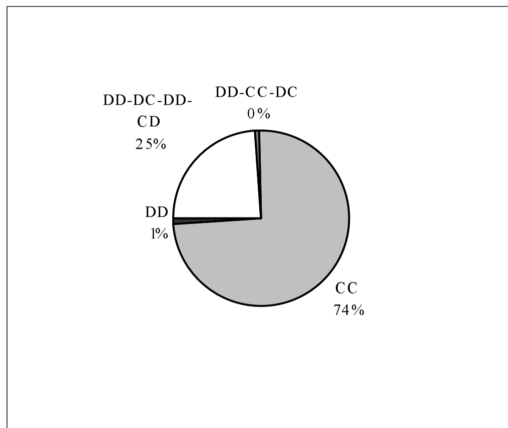

Figure 5: Frequencies of each of the possible outcomes from 5,000 trials. Parameters were randomly selected as described in Table 1.

It is interesting to note that every game converged to one of four possibilities: mutual cooperation, mutual defection, some variation on DD-DC-DD-CD, or some variation on DD-CC-DC.

4.3 Factors Leading to Cooperation

As shown previously, convergence to mutual cooperation is the most frequent outcome in a prisoner's dilemma played by two satisficing agents. Several factors influence this learning process between interacting agents. These are:

- initial aspirations,
- structure of the payoff matrix,
- learning rate, and
- initial actions

The remainder of this section focuses on analyzing how each of these parameters affect convergence to mutual cooperation.

Initial Aspirations

Figure 6 shows a contour plot of the frequency of mutual cooperation as a function of initial aspirations. It is clear that high aspirations are more likely to lead to cooperation. At first this may appear counter-intuitive—players with high aspirations might be unwilling to settle for cooperation. However, in most circumstances, both players are able to learn that they cannot expect more than mutual cooperation in the long run. On the other hand, players with low aspirations tend to remain satisfied with mutual defection or settle into cycles.

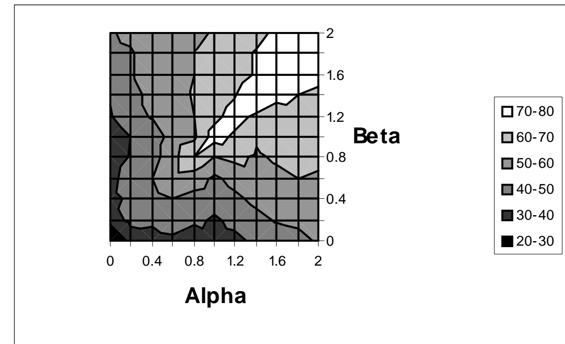

Figure 6: A contour plot of the percentage of trials out of 1,000 that converged to mutual cooperation as a function of initial aspirations. Light colors indicate that in most of the trials with the given initial aspirations, the agents learned to cooperate. Dark colors indicate that few of the trials led to mutual cooperation. Parameters other than α_0 and β_0 were selected randomly as described in Table 1.

Structure of the Payoff Matrix

The structure of the payoff matrix can also have considerable influence over the ability of the agents to converge to learn to cooperate. Figure 7 shows the frequency of mutual cooperation as a function of σ and δ.

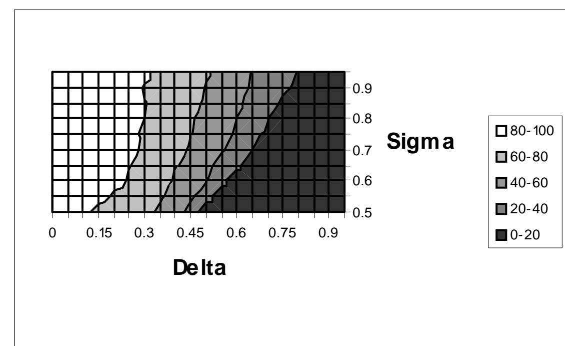

Figure 7: A contour plot of the percentage of trials out of 1,000 that converged to mutual cooperation as a function of each (δ, σ) pair. Light colors indicate that most of the trials converged to mutual cooperation, while dark colors indicate that few of the trials converged to cooperation. Parameters other than δ and σ were chosen randomly according to Table 1.

Note that cooperation is most likely when δ is small and σ is large. This is expected because the distinction between cooperation and defection blurs when σ and δ are close together. This type of behavior seems typical of non-optimizing algorithms. In describing his work in modeling human behavior, Arthur writes that human behavior (and his algorithm), "appear to 'discover' and exploit the optimal action with high probability, *as long as it is not difficult to discriminate*. But beyond a perceptual threshold, where differences in alternatives become less pronounced, non-optimal outcomes become more likely" [Arthur, 1991].

Initial Actions

To study the effects of initial actions on cooperation, we ran four sets of simulations, holding different initial actions constant each time. The percentages of samples that converge to cooperation for each group are shown in Table 2.

Initial Actions	% of Cooperation
Random	73.7 %
CC	81.6 %
DD	81.6 %
DC or CD	66.7 %

Table 2: Percentage of cooperation out of 1,000 trials as a function of initial actions. Parameters other than A_0 and B_0 where chosen according to Table 1.

While initial actions do not appear to be as significant as other factors, note that cooperation occurs with the same percentage regardless of whether the initial actions are cooperation or defection as long as both players choose the same action.

Learning Rate

The rate at which the aspirations are updated also has a considerable effect on whether mutual cooperation is learned. Figure 8 shows the relationship between λ and the percentage of trials that converged on mutual cooperation. As λ increases, the frequency of cooperation increases as well. The only exception is when λ = 1 (and thus aspirations are not updated at all), leading to virtually no cooperation.

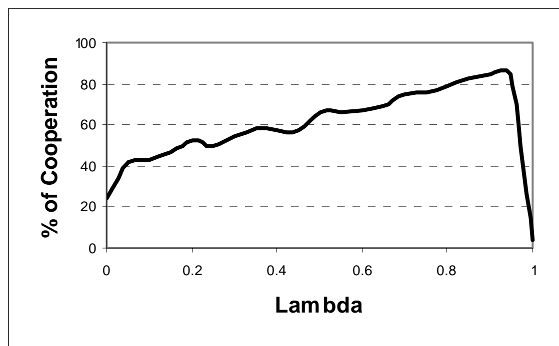

Figure 8: Percentage of trials out of 1,000 that converged to mutual cooperation as a function of the update rate, λ. Parameters other than λ were selected randomly as described in Table 1.

5 Conclusions and Further Work

To summarize the results of the previous section, we restate five important factors that increase the likelihood that two satisficing agents will learn to cooperate:

- Agents should learn, but slowly.
- The difference between payoffs for mutual defection and mutual cooperation should be maximized.
- Agents should have high initial aspirations.
- Agents should start out with similar behavior.

As a test of these principles, we ran a final set of simulations enforcing the following conditions: $A_0 = B_0$, $\sigma - \delta > 0.4, 1 > \lambda > 0.8$, $\alpha_0 > \sigma$, and $\beta_0 > \sigma$. Under these conditions, the agents learn to cooperate in 100% of 5,000 trials.

These results make a promising case for the use of satisficing in multi-agent systems as a way of balancing self-interest and common good when little information about the environment is available. Because agents do not directly model each other, the approach is fast, simple, and scalable to many players.

As a final note, we recognize that there are several directions for further work that should prove useful and interesting to researchers in multi-agent systems. We have limited our discussion of this satisficing algorithm to the prisoner's dilemma. However, because no assumptions about the relationships between the payoffs have been built into the algorithm, it should extend easily to other domains. In addition, the algorithm we have presented is limited to two-action decision problems with immediate feedback. Thus, the addition of a satisficing search algorithm for multiple actions is necessary and an extension to sequential decision problems would prove useful for many applications.

Acknowledgements

The authors gratefully acknowledge the support of the National Science Foundation under grant #CMS-9526018.

References

[Arthur 1991] W. Brian Arthur. Designing economic agents to act like human agents: A behavioral approach to bounded rationality. *The American Economic Review* 81 (May): 353-359, 1991.

[Axelrod 1984] R.M. Axelrod. *The Evolution of Cooperation*. Basic Books, 1984.

[Conlisk 1996] John Conlisk. Why bounded rationality? *Journal of Economic Literature* 34(2): 669-694, 1996.

[Hu and Wellman, 1998] J. Hu and M. P. Wellman. Multiagent reinforcement learning: Theoretical framework and an algorithm. *Proceedings of the Fifteenth International Conference on Machine Learning*, 242-250. San Francisco: Morgan Kaufman.

[Karandikar, *et al.* 1998] R. Karandikar, D. Mookherjee, D. Ray, and F. Vega-Redondo. Evolving aspirations and cooperation. *Journal of Economic Theory*, 80:292-331, 1998.

[Littman 1994] Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. *Proceedings of the Eleventh International Conference on Machine Learning*, 157-163. San Francisco: Morgan Kaufman.

[Sandholm and Crites, 1996] Tuomas W. Sandholm and Robert H. Crites. Multiagent reinforcement learning in the Iterated Prisoner's Dilemma. *BioSystems*, 37: 147-166, 1996.

[Sen, *et al.* 1994] S. Sen, M. Sekaran, and J. Hale. Learning to coordinate without sharing information. *Proceedings of the Twelfth National Conference on Artificial Intelligence* (AAAI-94). Seattle, WA. 426-431.

[Simon 1997] Herbert A. Simon. *Models of bounded rationality.* Vol. 3, *Empirically grounded economic reason*. Cambridge, Mass. MIT Press, 1997.