Discovering Search Algorithms with Program Transformation

Tristan Cazenave[1]

Abstract. Minimax is the basic algorithm used in two-persons player complete information computer games. It is also used in other games. Early research on the MiniMax algorithm have improved it giving the Alpha-Beta algorithm. We propose a list of program transformations that enable to rediscover the Alpha-Beta algorithm given the original MiniMax algorithm. This example is intended to serve as a basis for a more general program transformation system aimed at improving algorithms in games and problem solving. We also consider program transformation as a search algorithm, and outline the architecture of a system that can automatically perform the transformations we have described and its self-application.

Key words: Program Synthesis, Meta-programming, Minimax, Alpha-Beta, Search, Reflection.

[1] Laboratoire d'Intelligence Artificielle, Département Informatique, Université Paris 8,
2 rue de la Liberté, 93526 Saint Denis, France.
e-mail: cazenave@ai.univ-paris8.fr
tel: 33 1 49 40 64 04 fax: 33 1 49 40 64 10

Discovering Search Algorithms with Program Transformation

Abstract. Minimax is the basic algorithm used in two-persons player complete information computer games. It is also used in other games. Early research on the MiniMax algorithm have improved it giving the Alpha-Beta algorithm. We propose a list of program transformations that enable to rediscover the Alpha-Beta algorithm given the original MiniMax algorithm. This example is intended to serve as a basis for a more general program transformation system aimed at improving algorithms in games and problem solving. We also consider program transformation as a search algorithm, and outline the architecture of a system that can automatically perform the transformations we have described and its self-application.

Key words: Program Synthesis, Meta-programming, Minimax, Alpha-Beta, Search, Reflection.

1 Introduction

The use of meta-programming tools in games has enabled to automatically generate efficient specific search program for a given game by specializing the definition of the goal of the game with the rules of the game [Cazenave 1998a,b]. A follow-up of this research would be to discover general search algorithm that can be applied to many games, using program transformation techniques such as the rules+strategies approach [Pettorossi 2000].

We show that it is possible to improve the efficiency of standard search algorithms with program transformation. The most well-known of the AI search algorithm may be the Minimax and its enhanced counterpart: Alpha-Beta. We will describe some of the transformations that enable to discover Alpha-Beta with program transformation given the Minimax algorithm. We also consider program transformation as a search algorithm, and outline the architecture of a system that can automatically perform the transformations we have described.

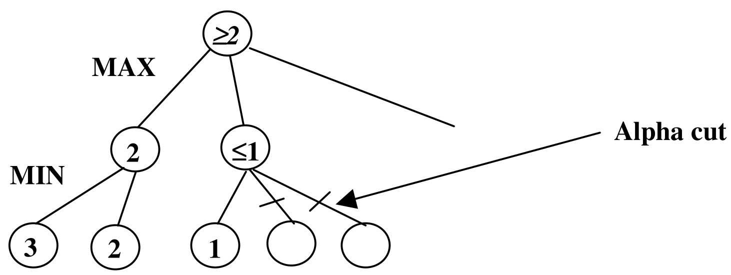
In section 2, we briefly describe the Minimax search algorithm, and its improvements such as Negamax and Alpha-Beta. In section 3, we give the transformations that enable to discover Beta cuts, given a simplified Minimax algorithm. Section 4 shortly gives hints on the program transformations needed to discover Alpha-Beta cuts given a realistic Minimax algorithm. Section 5 analyzes the architecture of a general search program that can perform the discovery of Alpha-Beta cuts. Section 6 concludes and outlines future work.

2 Minimax, Alpha-Beta, Negamax and Nega-Alpha-Beta

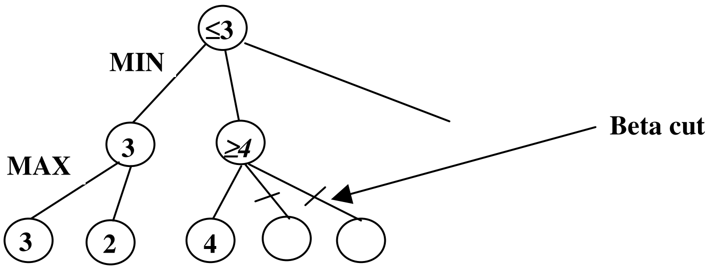
The Minimax algorithm and its related improved search algorithms are used in most two players, zero-sum complete information games. The algorithm was proposed half a century ago by von Neumann and Morgenstern [Neumann 1944] to decide which move to make in Chess. Alan Turing [Turing 1953] proposed some search strategies based on the Minimax principle, and an important improvement on the algorithm was Alpha-Beta. A weak form of Alpha-Beta first appeared in early Chess programs such as NSS, by Newell, Shaw and Simon [Newell 1958].

The Minimax search algorithm is associated to an evaluation function, that takes a position of the game as input and computes a numerical evaluation for this position. The higher the evaluation, the better the position is. Given a perfect evaluation function, it is useless to search many moves ahead. However, in complex games, such as Go or Chess, there is no perfect evaluation function. A program is much better if it can search many moves ahead of the current position. The fundamental hypothesis of the Minimax algorithm is that the opponent uses the same evaluation function as the program. Therefore, the goal of the program is to play the moves that maximize the evaluation function, whereas the opponent plays the moves that minimize this same evaluation function. The two players play alternatively, so the recursive Minimax algorithm successively calls maximizing and minimizing functions.

An improvement on the Minimax algorithm is the Alpha-Beta algorithm that always look at less nodes than Minimax while always giving the same answer. The Alpha cut is made at the Min levels of the search tree.It is based on the reasoning that if the current value for a Min level is lower than the current value of the upper Max level, whatever are the remaining values of the Min node, they won't change the value of the upper Max level. Example:



The Beta cut is the symmetric of the Alpha cut for Max nodes. Example:



When the nodes are well ordered, the Alpha-Beta finds an answer in $1+2\sqrt{n}$ nodes, whereas Minimax finds the same answer in n nodes [Knuth 1975].

A more concise formulation of the Minimax algorithm is the Negamax algorithm. The sign of the evaluation is changed at each level, so that each level is a maximizing level. In the Negamax framework, the algorithm is described recursively as:

$$\text{Negamax}(n) = \max_i(-\text{Negamax}(n_i))$$

where the n_i are the successors of node n. In the Negamax framework, the same kind of Alpha-Beta cut can be performed at each node of the search tree, leading to the Nega-Alpha-Beta algorithm.

Many other enhancements, such as iterative deepening, transposition tables, the killer heuristic, the history heuristic or null move forward pruning, have been found that improve the Nega-Alpha-Beta algorithm [Marsland 2000].

3 Transforming a simplified version of Minimax

In order to identify the kind of program transformation knowledge needed to rediscover Alpha-Beta, we first transform a simplified MiniMax. In the simplified version, only two moves are possible in each position and we do not care about the termination of the search, so the program is:

```
1. maxeval(A):- mineval(B),mineval(C),maxi(B,C,A).
2. mineval(A):- maxeval(B),maxeval(C),mini(B,C,A).
3. maxi(B,C,B):- B>=C.
4. maxi(B,C,C):- C>=B.
5. mini(B,C,B):- B<=C.
6. mini(B,C,C):- C<=B.
```

Unfolding the second mineval predicate in 1 gives:

```
7. maxeval(A):-    mineval(B),maxeval(B1),maxeval(C1),
                   mini(B1,C1,C),maxi(B,C,A).
```

The goal is to prove that when B >= B1, there is no need to compute maxeval (C1). We unfold mini and maxi in clause 7. First we unfold mini:

```
8. maxeval(A):-   mineval(B),maxeval(B1),maxeval(C1),
                  B1<=C1,maxi(B,B1,A).
9. maxeval(A):-   mineval(B),maxeval(B1),maxeval(C1),
                  C1<=B1,maxi(B,C1,A).
```

Then we unfold maxi:

```
10. maxeval(B):-  mineval(B),maxeval(B1),maxeval(C1),
                  B1<=C1,B>=B1.
11. maxeval(B1):- mineval(B),maxeval(B1),maxeval(C1),
                  B1<=C1,B1>=B.
12. maxeval(B):-  mineval(B),maxeval(B1),maxeval(C1),
                  C1<=B1,B>=C1.
13. maxeval(C1):- mineval(B),maxeval(B1),maxeval(C1),
                  C1<=B1,C1>=B.
```

Here the clauses are written in a declarative way, so that the order of the litterals in the clause can be changed subject to usual constraints on the possibility of matching the litteral (for example, B>=B1 can only be matched when B and B1 are already known). Therefore 10 and 12 are the same clauses with a different ordering, and 11 and 13 also, they can be rewritten:

```
14. maxeval(B):-  mineval(B),maxeval(B1),
                  B>=B1,maxeval(C1),B1<=C1.
15. maxeval(B):-  mineval(B),maxeval(B1),
                  maxeval(C1),B>=C1,C1<=B1.
```

Either B<B1 or B>=B1, so we can split the clause 15 with these two cases:

```
16. maxeval(B):-  mineval(B),maxeval(B1),B<B1,
                  maxeval(C1),B>=C1,C1<=B1.
17. maxeval(B):-  mineval(B),maxeval(B1),B>=B1,
                  maxeval(C1),B>=C1,C1<=B1.
```

Then, the test B>=C1 in clause 17 can be removed because we already have the tests B>=B1 and B1>=C1 that ensure that B>=C1. Therefore 17 can be rewritten:

```
18. maxeval(B):- mineval(B),maxeval(B1),
                  B>=B1,maxeval(C1),C1<=B1.
```

By considering that clauses 14 and 18 are the same except for the complementary tests at the end, we can join them in a new clause:

```
19. maxeval(B):-  mineval(B),maxeval(B1),B>=B1,
                  maxeval(C1).
```

In clause 19, the variable C1 is not linked any more to the other variables, so we can remove the associated predicate (provided the program are declarative and that sub-goals do not have side effects), and we have:

```
20. maxeval(B):- mineval(B),maxeval(B1),B>=B1.
```

Which is a Beta cut! So the resulting program is now:

```
20. maxeval(B):-  mineval(B),maxeval(B1),B>=B1.
16. maxeval(B):-  mineval(B),maxeval(B1),B<B1,
                  maxeval(C1),B>=C1,C1<=B1.
11. maxeval(B1):- mineval(B),maxeval(B1),
                  maxeval(C1),B1<=C1,B1>=B.
13. maxeval(C1):- mineval(B),maxeval(B1),
                  maxeval(C1),C1<=B1,C1>=B.
```

Which can be transformed in:

```
21. maxeval(Res):-      mineval(B), maxeval(B1),
                        maxeval (B,B1,Res).
22. maxeval(B,B1,B):-   B>=B1.
23. maxeval(B,B1,Res):- maxeval(C1),maxeval(B,B1,C1,Res).
24. maxeval(B,B1,C1,B):- B<B1,B>=C1,C1<=B1.
25. maxeval(B,B1,C1,B1):- B1<=C1,B1>=B.
26. maxeval(B,B1,C1,C1):- C1<=B1,C1>=B.
```

This program is performing Beta cuts. So, on this simplified version of Minimax, we have been able to discover Beta cuts using simple program transformation tools.

4 Transforming Negamax into Nega-Alpha-Beta

We will now show how to transform a realistic Negamax algorithm into a Nega-Alpha-Beta algorithm. The definition of a Negamax algorithm is:

```
1. value(Pos,Depth,Move,Eval):-
         possiblemoves(Pos,List),
         maxmovelist(Pos,List,D,Move,Eval).
2. maxmovelist(Pos,[],D,M,-1000).
3. maxmovelist(Pos,[Move|Ls],D,M,Eval):-
         maxmove(Pos,Move,D,M1,Eval1),
         maxmovelist(Pos,Ls,D,M2,Eval2),
         takemax(M1,Eval1,M2,Eval2,M,Eval).
4. takemax(M1,Ev1,M2,Ev2,M1,Ev1):-Ev1>=Ev2.
5. takemax(M1,Ev1,M2,Ev2,M2,Ev2):-Ev1<=Ev2.
```

```
6. maxmove(Pos,M,D,M1,Ev1):-
          play(Pos,M,P1), D1 is D-1,
          possiblemoves(P1,LM),
          maxmovelist(P1,LM,D1,M1,Ev2),
          Ev1 is -Ev2.
```

In order to find cuts, we will follow the same pattern as in the previous simplified example. So we begin with unfolding maxmovelist in clause 3:

```
8. maxmovelist(Pos,[Move|[Move2|Ls]],D,M,Eval):-
          maxmove(Pos,Move,D,M1,Eval1),
          maxmove(Pos,Move2,D,M2,Eval2),
          maxmovelist(Pos,Ls,D,M3,Eval3),
          takemax(M2,Eval2,M3,Eval3,M4,Eval4).
          takemax(M1,Eval1,M4,Eval4,M,Eval).
```

Then we can unfold the second maxmove litteral in clause 8:

```
9. maxmovelist(Pos,[Move|[Move2|Ls]],D,M,Eval):-
          maxmove(Pos,Move,D,M1,Eval1),
          play(Pos,Move2,P1), D1 is D-1,
          possiblemoves(P1,LM),
          maxmovelist(P1,LM,D1,M2,Ev2),
          Eval2 is -Ev2,
          maxmovelist(Pos,Ls,D,M3,Eval3),
          takemax(M2,Eval2,M3,Eval3,M4,Eval4).
          takemax(M1,Eval1,M4,Eval4,M,Eval).
```

And then unfold the first litteral maxmovelist in clause 9 to have:

```
10. maxmovelist(Pos,[Move|[Move2|Ls]],D,M,Eval):-
          maxmove(Pos,Move,D,M1,Eval1),
          play(Pos,Move2,P1), D1 is D-1,
          possiblemoves(P1,[M5|LMs]),
          maxmove(P1,M5,D1,M6,Eval6),
          maxmovelist(P1,LMs,D1,M7,Eval7),
          takemax(M6,Eval6,M7,Eval7,M2,Ev2).
          Eval2 is -Ev2,
          maxmovelist(Pos,Ls,D,M3,Eval3),
          takemax(M2,Eval2,M3,Eval3,M4,Eval4).
          takemax(M1,Eval1,M4,Eval4,M,Eval).
```

Now, we can reason on the subset of litterals composed of:

```
11.       takemax(M6,Eval6,M7,Eval7,M2,Ev2).
          Eval2 is -Ev2,
          takemax(M2,Eval2,M3,Eval3,M4,Eval4).
          takemax(M1,Eval1,M4,Eval4,M,Eval).
```

If we unfold the takemax litterals, one of the unfolded set of litteral we get is:

```
12.        Eval6>=Eval7,
           -Eval6>=Eval3,
           Eval1>=-Eval6.
```

and another one is:

```
13.        Eval7>=Eval6,
           -Eval7>=Eval3,
           Eval1>=-Eval7.
```

and either Eval1>=-Eval6 or Eval1<-Eval6. So by splitting 13 with these two cases, we get:

```
14.        Eval7>=Eval6,
           -Eval7>=Eval3,
           Eval1>=-Eval6
           Eval1>=-Eval7.

15.        Eval7>=Eval6,
           -Eval7>=Eval3,
           Eval1<-Eval6
           Eval1>=-Eval7.
```

In 14, we have Eval1>=-Eval6 and Eval7>=Eval6 therefore we can remove Eval1>=-Eval7 which can be deduced of the two former tests. Moreover, -Eval7>=Eval3 and Eval7>=Eval6 enables to deduce that –Eval6 >= Eval3 in 14. And Eval6>=Eval7 and –Eval6>=Eval3 in 12 enable to deduce –Eval7>=Eval3 in 12. So now, we can join 12 and 14 by removing the complementary conditions Eval7>=Eval6 and Eval6>=Eval7 so as to get:

```
16.        -Eval6>=Eval3,
           -Eval7>=Eval3,
           Eval1>=-Eval6.
```

Another clause can obtained in a similar way, and contains:

```
17.        -Eval6>=Eval3,
           -Eval7<=Eval3,
           Eval1>=-Eval6.
```

So by joining them, we get:

```
18.        -Eval6>=Eval3,
           Eval1>=-Eval6.
```

In this clause, there are no more links between the Eval7 and M7 variables and the other variables (the predicate containing the M7 and Eval7 variables have all their other variables already closed). Moreover, the predicate closing the M7 and Eval7 variables always succeed, but its results are not taken into account. So we can remove it. Then we get the following transformed clause:

```
19. maxmovelist(Pos,[Move|[Move2|Ls]],D,M,Eval1):-
          maxmove(Pos,Move,D,M1,Eval1),
          play(Pos,Move2,P1), D1 is D-1,
          possiblemoves(P1,[M5|LMs]),
          maxmove(P1,M5,D1,M6,Eval6),
          Eval1>=-Eval6,
          maxmovelist(Pos,Ls,D,M3,Eval3),
          -Eval6>=Eval3.
```

This transformed clause performs Alpha-Beta cuts. Associated to other transformed clauses, this clause can be further refined to give a more concise definition of a kind of Nega-Alpha-Beta.

5 The Search Space of Program Transformations

In order to estimate the complexity of a problem or a game, a useful heuristic is to determine the search space of the problem. This means estimating the average number of possible moves and the depth of the search. If we consider program transformation as a search in the space of possible transformed programs, at each node (i. e. transformed program), we can apply a given number of transformations, say N. In order to find an interesting improved program, a minimal number of such transformation is necessary, say D. The size of the search space is N^D. A way to account for the efficacy of the transformed programs is to test them on some standard test sets. At each node of the search graph, the program is tested and in the end the transformed programs are sorted according to the time they took to solve the standard problems. The transformed programs are stopped as soon as they use more time than the original program to give their answer using the same data.

In this architecture, the time used for transformation is small compared to the time needed to test the transformed programs. Some usual algorithms used for computer games can be used to speed-up the search for a good program. For example iterative deepening can be used: the search is first performed at depth one, then at depth two and so on until no more time is available for search. Another useful optimization is the detection of identical nodes in the search space, some method such as transposition tables can de adapted to efficiently detect equivalent programs, and save a lot of time by not re-searching the same sub-trees many times.

In our example, the number of transformations required to discover the Alpha-Beta algorithm is on the order of 30. At each node of the transformation search space, between 4 and 20 transformations are possible. The size of the search space is then of the order of 10^{30}, which is too large to be completely searched. The usual way to reduce the size of the search space is to define macro-moves in the search space that apply a list of transformations instead of applying atomic transformations.

This reduction is similar to the rule+strategies approach [Pettorossi 2000]. We believe that working on the transformation of search program can uncover new interesting program transformation strategies. Some interesting strategies can already be devised looking at the transformations we have used to discover Alpha-Beta cuts.

All these ideas need further tests, but the architecture described gives a new perspective on program transformation systems. It is concerned with the automatic discovery of efficient search algorithm and with practical aspects of the efficient implementations of program transformation systems as search programs. It is a bridge between program transformation and Artificial Intelligence search algorithms. There may even be a kind of reflection: program transformation can be used to improve search algorithms, and the improvements in search algorithms can be used to improve program transformation. The ultimate goal of this work being to discover by mean of program transformation some new search algorithms better than the state of the art.

6 Conclusion and Future Work

Further testing is needed to evaluate empirically the efficiency of our approach to search algorithm discovery. We have shown in this paper, as a preliminary result, that it is possible to discover the Alpha-Beta algorithm, given the Minimax algorithm and some simple program transformation tools. We have proposed an architecture based on a search algorithm to completely automate the discovery of search algorithms. A necessary transformation to make the program transformation tractable is the definition of macro moves in this search space: defining some macro-transformation as a combination of basic transformations. A promising area of research is the self application of this algorithm: using the architecture to discover a search algorithm than can improve the search that discovers new search algorithms.

7 References

1. Cazenave T.: *Metaprogramming Forced Moves*. Proceedings ECAI98 (ed. H. Prade), pp. 645-649. John Wiley & Sons Ltd., Chichester, England. ISBN 0-471-98431-0. 1998.
2. Cazenave T.: *Controlled Partial Deduction of Declarative Logic Programs*. ACM Computing Surveys, Special issue on Partial Evaluation, vol. 30 n°3es, 1998.
3. Knuth D. E., Moore R. W.: *An Analysis of Alpha-Beta Pruning*. Artificial Intelligence vol. 6, n°4, pp 293-326, North-Holland 1975.
4. 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.
5. Neumann J., Morgenstern O.: *Theory of Games and Economic Behaviour.* Second Edition, 1947. Princeton University Press, Princeton, N. Y. 1944.

6. Newell A., Shaw J. C., Simon H. A.: *Chess Playing Programs and the Problem of Complexity.* IBM Journal of Research and Development, vol. 4 n° 2, pp. 320-335, 1958. Reprinted (1963) in *Computers and Thought*, pp. 39-70, McGraw-Hill, N. Y.
7. Pettorossi A., Proietti M.: *Automatic Derivation of Logic Programs by Transformation*. Course notes for ESSLLI 2000.
8. Turing A. M.: *Digital Computers Applied to Games*. Faster than Thought, ed. B. V. Bowden, pp. 286-297, 1953. Pitman, London.