VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

PARALLELIZING CHESS MOVES

Parallel and Distributed Computing

PROJECT REPORT

Course Code: CSE4001

Slot: B2

Submitted to

Prof. Narayanan Prasanth

By

Group Members:

Rachit Gupta	17BCE0622
Tanishq Bafna	17BCE2001
Archit Agarwal	17BCE2089
Siddharth Singh Solanki	17BCI0017
Ishika Ahuja	18BCE0894

Base Paper: Guidry, Matthew, and Charles McClendon. "Techniques to Parallelize Chess."

ABSTRACT

The project aims at optimizing a simple chess program, starting by parallelizing its search algorithm, then applying some optimization techniques, and finally trying out the iterative deepening variation of the classical minimax algorithm. Tentative search algorithms would be Younger brothers wait concept, Lazy SMP, Root Splitting. After parallelizing the search algorithms, the game will be stimulated using Machine Learning techniques to compare efficiency and compatibility of each algorithm with the others. While implementing iterative deepening, execution time and number of nodes visited will be analysed against depth of the root tree.

INTRODUCTION

Chess programming is considered one of the classical challenges of artificial intelligence and computer science in general. It might be a single game, but it gets complicated due to 10 to the power of 120 possible moves. The most commonly used algorithm for most of the games is the Minimax algorithm. The minimax; from a given node in the search tree, computes all its children (that is the set of all the possible moves from the current configuration) and visits each child one by one, computing each time an evaluation function that gives away a kind of performance of that child. We repeat that recursively until we reach the leaves. At the end, the algorithm chooses the branch with the highest evaluation. However, the search tree being too large, we generally stop at a maximum depth, otherwise it would be computationally heavy. The main agenda of this project is to parallelize the Minimax algorithm, making it more robust, allowing s to compute independent branches simultaneously, gaining a considerable time that we could use to visit an extra layer and thus resulting in a more confident response from it. To compare the different parallelizing techniques used, we use python to compute the statistics and visualize them.

PROBLEM STATEMENT

Experimenting various parallelizing techniques on the Minimax algorithm through a chess engine to obtain the statistical data useful for comparing these techniques. Finding the optimal parallelizing technique that gives better performance statistics amongst the considered parallelizing techniques.

OBJECTIVES

- Understand the game theory behind chess.
- Implement minimax algorithm as searching algorithm to find all possible moves for all the pieces.
- Understand and implement root splitting as the base parallelization technique.
- Implement and compare various parallelization techniques such as move reordering, beam search and iterative deepening.

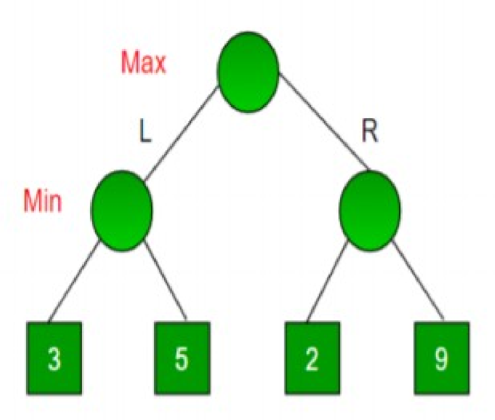
METHODS/MODELS

1. MINIMAX ALGORITHM

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.

In Minimax the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.

Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.



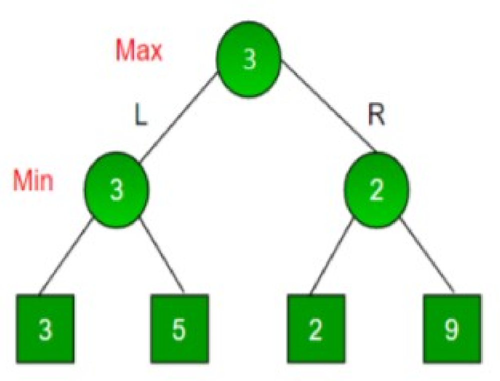


Fig 1(a). A game with 4 final states

Fig 1(b). Game tree after applying Minimax algorithm

The above figures show the backtracking technique that Minimax algorithm incorporates.

2. ALPHA BETA PRUNING

Alpha-Beta pruning is an optimization technique for minimax algorithm, which reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.
Alpha is the best value that the maximiser currently can guarantee at that level or above. Beta is the best value that the minimizer currently can guarantee at that level or above.

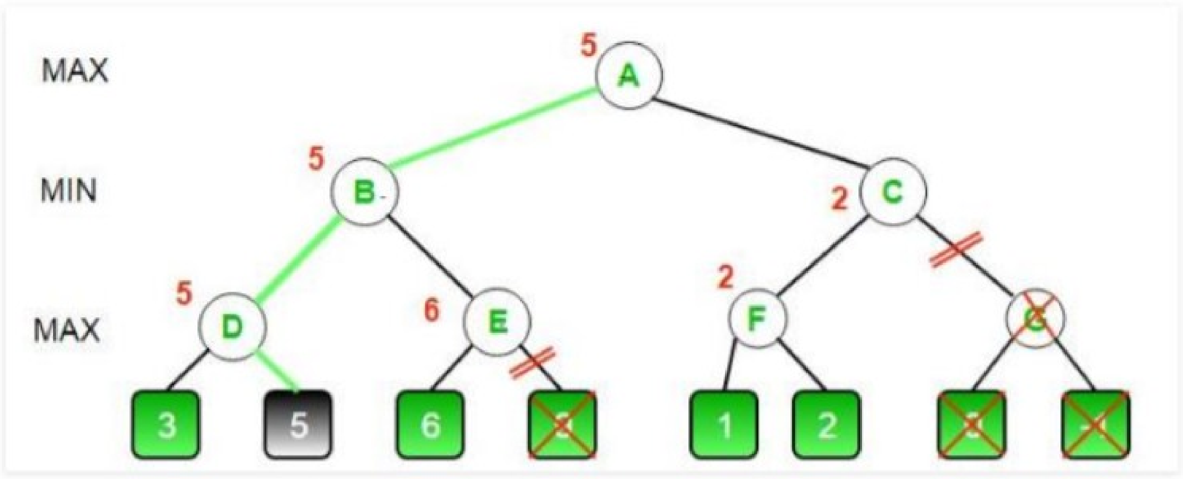


Fig 2. The game tree after the alpha beta pruning has been applied.

3. ROOT SPLITTING

The main idea of this technique is to ensure that each node except for the root, is visited by only one processor.
To keep the effect of the alpha beta pruning, we split the children nodes of the root into clusters, each cluster being served by only one thread. Each child is then processed as the root of a sub search tree that will be visited in a sequential fashion, thus respecting the alpha beta constraints. When a thread finishes computing its subtree(s) and returns its evaluation to the root node, that evaluation is

compared to the current best evaluation (that’s how minimax works), and that best evaluation may be updated.
So, to ensure coherent results, given that multiple threads may be returning to the root node at the same time, threads must do this part of the work in mutual exclusion: meaning that comparing to and updating the best evaluation of the root node must be inside a critical section so that it’s execution remains sequential.

The OpenMP directives used for parallelization are:

- #pragma omp parallel private (/*variables private to thread here */)
- #pragma omp for schedule (dynamic)

The two above directives need to be put right before the for loop to parallelize. In the first one we declare the variables that must be private to each thread. In the second directive we specify a dynamic scheduling between threads, meaning that if a thread finishes its assigned iterations before others do,
it'll get assigned some iterations from another thread, that way making better use of the available threads.

Then we must ensure inside the for loop that after each minimax call returns, the thread enters in a critical section in mutual exclusion to compare (and modify) the best evaluation of the root node. We do that using the following directive:

- #pragma omp critical { // Code here is executed in mutual exclusion }

All the code written inside this directive will be run by at most one thread at a time, ensuring thus the coherence of the value of our best evaluation variable.

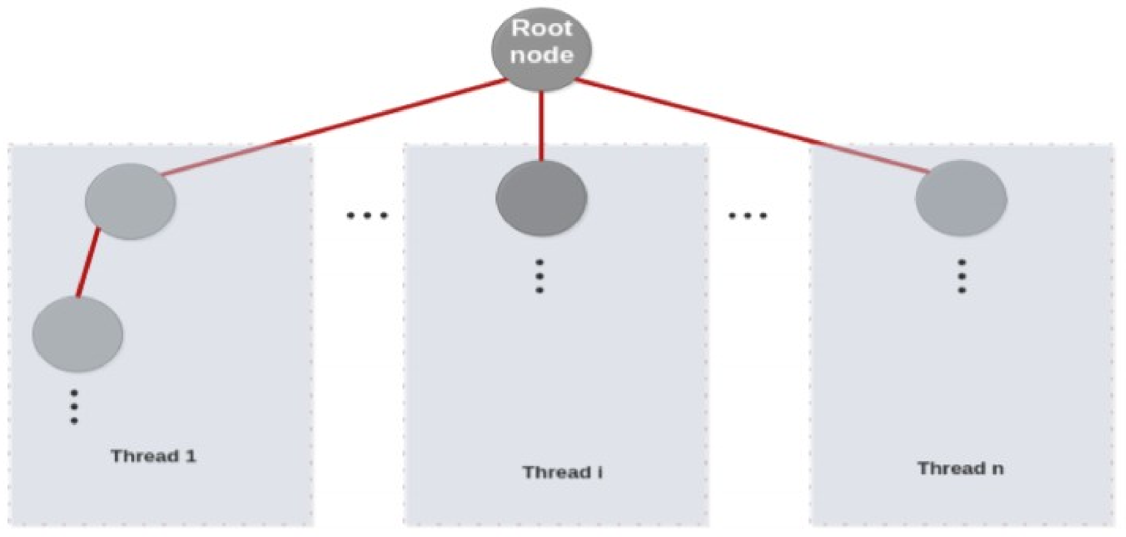


Fig 3. Root Splitting

4. MOVE REORDERING

The minimax algorithm enumerates all the possible moves from a given node. At each step, the minimax evaluates the current node and assesses it as being promising or average. It is the job of the alpha beta technique to cut off the worst nodes, preventing the minimax from visiting them, thus gaining significant amounts of time.

The sooner we prune the better, as cutting off a node eliminates its entire descendance from the search tree. Reordering the children of a given node in a way to start exploring the most promising branches (a priori) generates more cut-offs. To reorder child nodes before exploring them, we must rely on the estimation function to evaluate those nodes. The estimation we are working with is good enough, and is unchanged for all versions of the engine in this simulation.

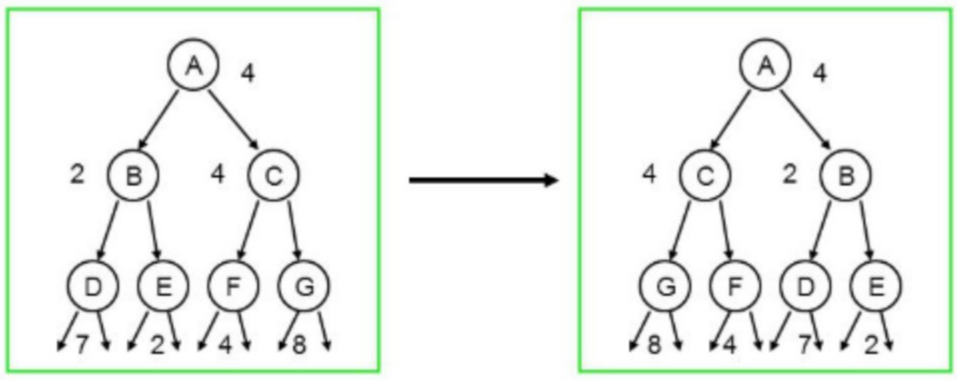


Fig 4. Mover reordering

5. BEAM SEARCH

The beam search technique uses the move reordering to only explore the most promising nodes of the search tree; that is, after we reorder the children of a given node, we only explore a subset of those child nodes, by ignoring the least promising ones (according to the estimation function!), and thus making the search faster. Of course, the trade-off of doing so is to risk not exploring parts of the search tree that might lead to better moves.

6. ITERATIVE DEEPENING

The iterative deepening is a variation of the minimax fixed-depth "d" search algorithm. It does the following: Explore at all depths from 1 to "d", and after each exploration, reorder the child nodes according to the value returned by that exploration. Of course, this means exploring all the nodes between depth 1 and d-1 many times. This may seem redundant and time wasting, but the overhead of expanding the same nodes many times is small in comparison to growth of the exponential search tree.

At each depth explored, we get a better estimation of the child nodes, and thus a better reordering that leads to a better exploring of the tree, which generates more pruning and significantly decreases the search tree size.

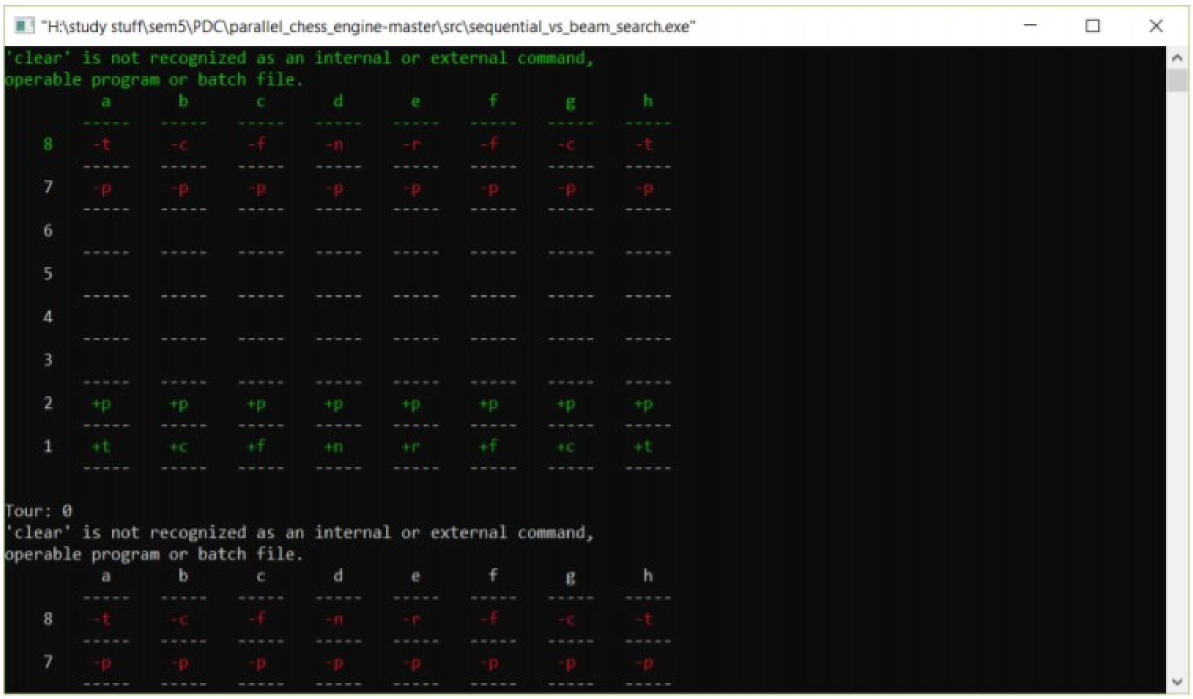
This technique is often preferred over other search methods for large search trees where the depth is not known, as it is for the chess search tree.

7. SIMULATION

To compare the performance of the parallelized version of the chess engine against the sequential one, we wrote a python script that computes some statistics. This simple program makes the old engine play against the parallel one at different search depths. For each depth, we make them play 10 games, we record the time taken for each move (in chess terms: each ply) of the game for both players (separately, of course!) and then we compute the mean of all the moves for all the games. In addition to execution times, we store the number of nodes visited to compare the search spaces explored by both players. The machine used has Intel I5 4th generation processor and 8 GB of RAM.

RESULT ANALYSIS

The chess programs when run, sequentially visualise the moves made by both the algorithms.



```
"H:\study stuff\sem5\PDC\parallel_chess_engine-master\src\sequential_vs_beam_search.exe"
'clear' is not recognized as an internal or external command,
operable program or batch file.
           a       b       c       d       e       f       g       h
         -----   -----   -----   -----   -----   -----   -----   -----
    8     -t      -c      -f      -n      -r      -f      -c      -t
         -----   -----   -----   -----   -----   -----   -----   -----
    7     -p      -p      -p      -p      -p      -p      -p      -p
         -----   -----   -----   -----   -----   -----   -----   -----
    6
         -----   -----   -----   -----   -----   -----   -----   -----
    5
         -----   -----   -----   -----   -----   -----   -----   -----
    4
         -----   -----   -----   -----   -----   -----   -----   -----
    3
         -----   -----   -----   -----   -----   -----   -----   -----
    2     +p      +p      +p      +p      +p      +p      +p      +p
         -----   -----   -----   -----   -----   -----   -----   -----
    1     +t      +c      +f      +n      +r      +f      +c      +t
         -----   -----   -----   -----   -----   -----   -----   -----

Tour: 0
'clear' is not recognized as an internal or external command,
operable program or batch file.
           a       b       c       d       e       f       g       h
         -----   -----   -----   -----   -----   -----   -----   -----
    8     -t      -c      -f      -n      -r      -f      -c      -t
         -----   -----   -----   -----   -----   -----   -----   -----
    7     -p      -p      -p      -p      -p      -p      -p      -p
         -----   -----   -----   -----   -----   -----   -----   -----
```

Fig. Chess game execution

For each comparative analysis, two graphs have been plotted, each comparing execution time(s) and nodes visited with respect to depth.

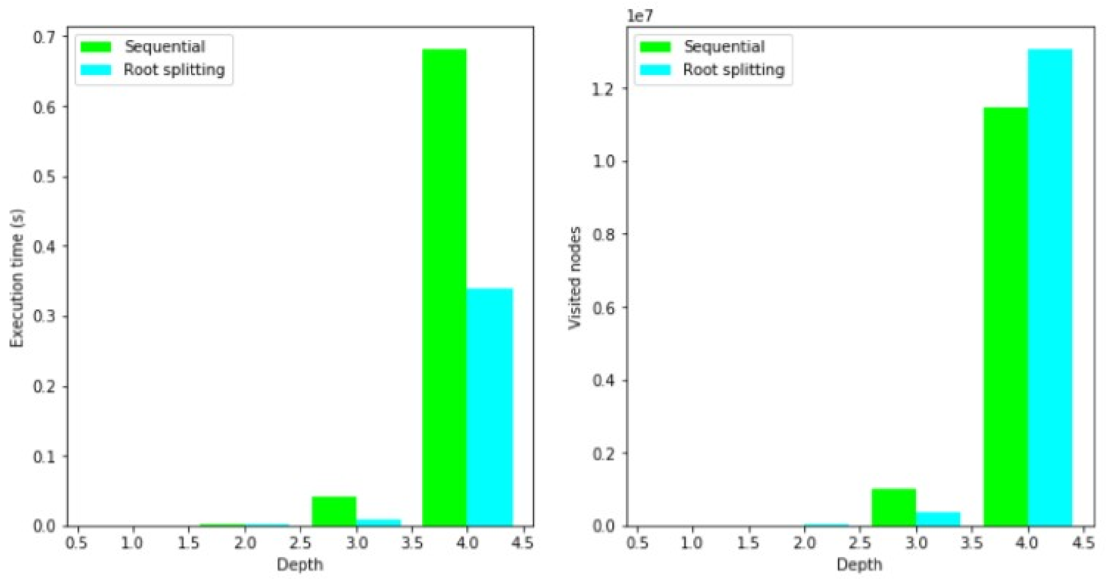


Fig.

Sequential vs Root splitting

As seen in the above graph, the parallel seems to be clearly faster at depths 3 and 4, even if the number of visited nodes is slightly higher for the parallel engine. The speedup at depth 4 is computed to be 2.000578. Even though 4 threads are used, going two times faster is a good result.

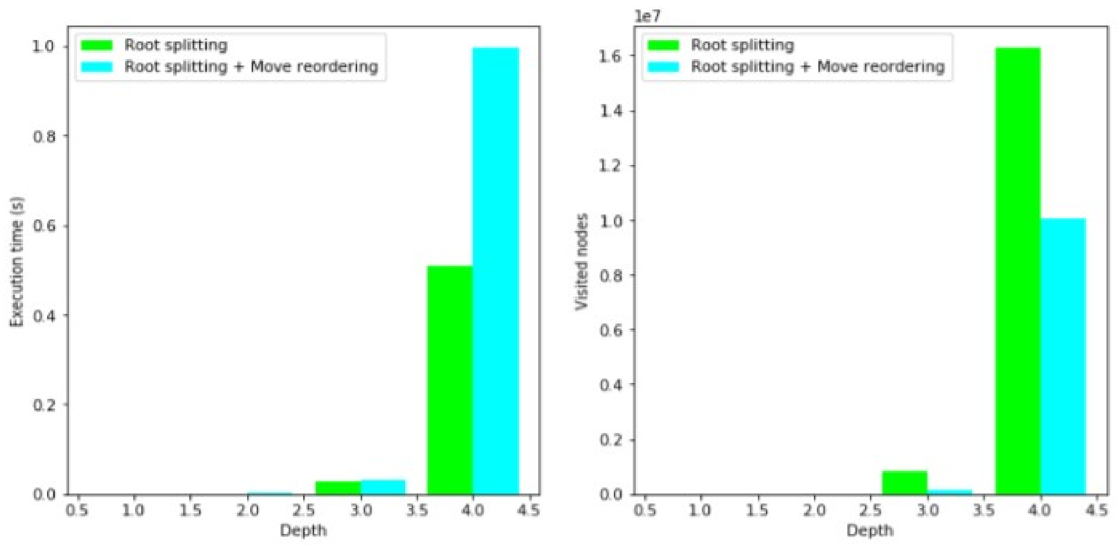


Fig. Root Splitting vs Root Splitting + Move reordering

The results obtained for move reordering is poor as reordering the moves means computing the estimation function for every child of the visited node, and then sorting them. While we do get more cut-offs as expected (a lot less nodes visited than before), we lose time in the overhead of sorting and computing the estimation at every single node visited.

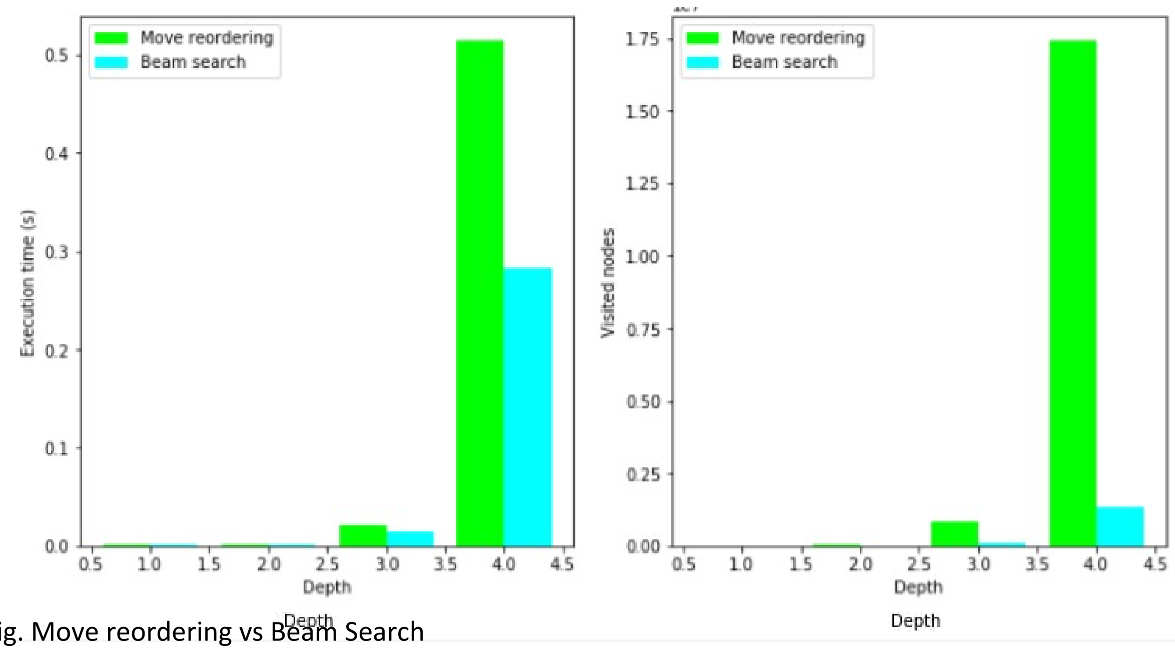
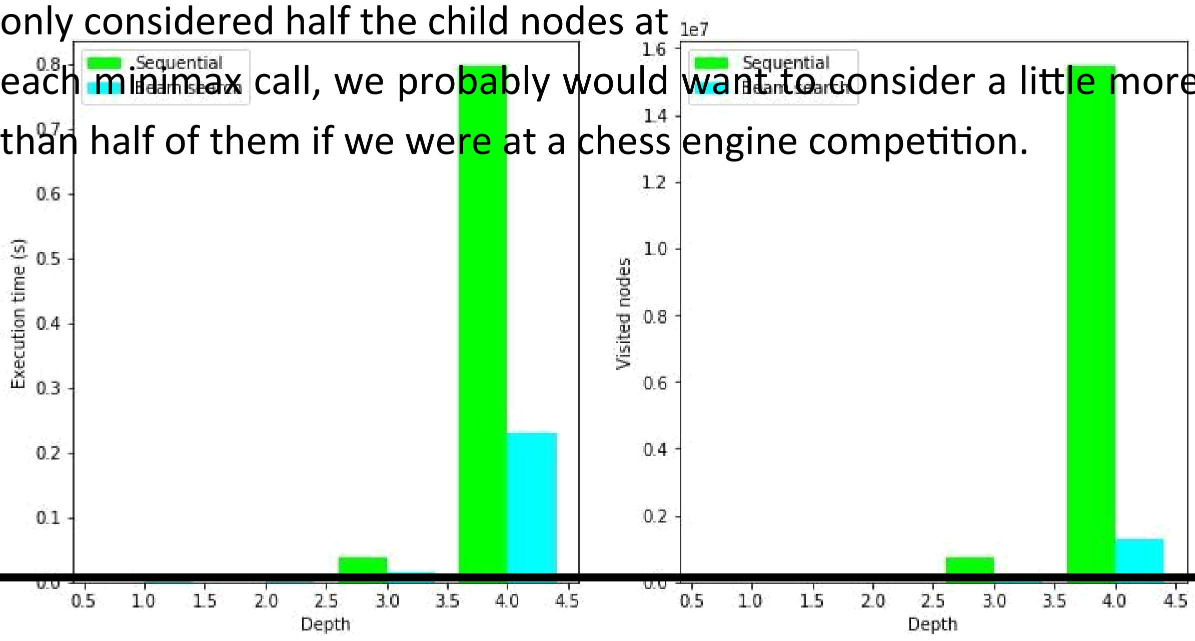


Fig. Move reordering vs Beam Search

The execution time is reduced due to exploring less nodes, at the risk of losing some good moves due to a bad estimation. So, for this technique to give good results, the estimation function must be good. There is a huge difference in the number of visited nodes. We only considered half the child nodes at each minimax call, we probably would want to consider a little more than half of them if we were at a chess engine competition.



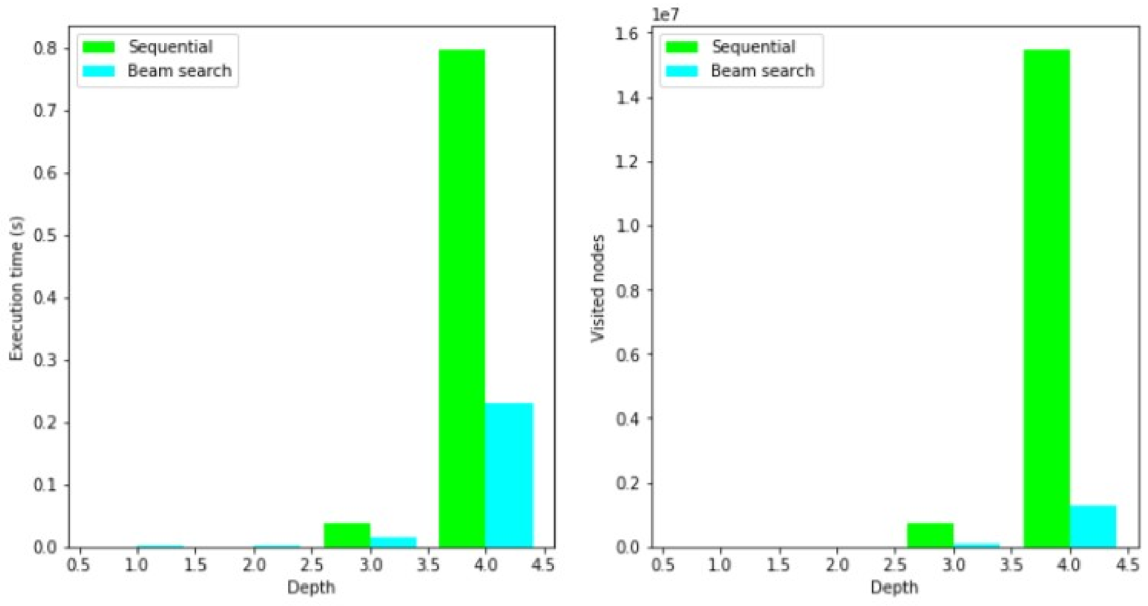


Fig. Sequential vs Beam Search

From the above graphs we can see the principle behind beam search, that is exploring lot less nodes. This is done firstly, because we generate more cut-offs thanks to the reordering, and second because we consider less possibilities using beam search. The latter is risky, but gives good results when the estimation function has a low probability of giving inaccurate evaluations. The speed up obtained is 3.467 at a depth of 3 and 4.

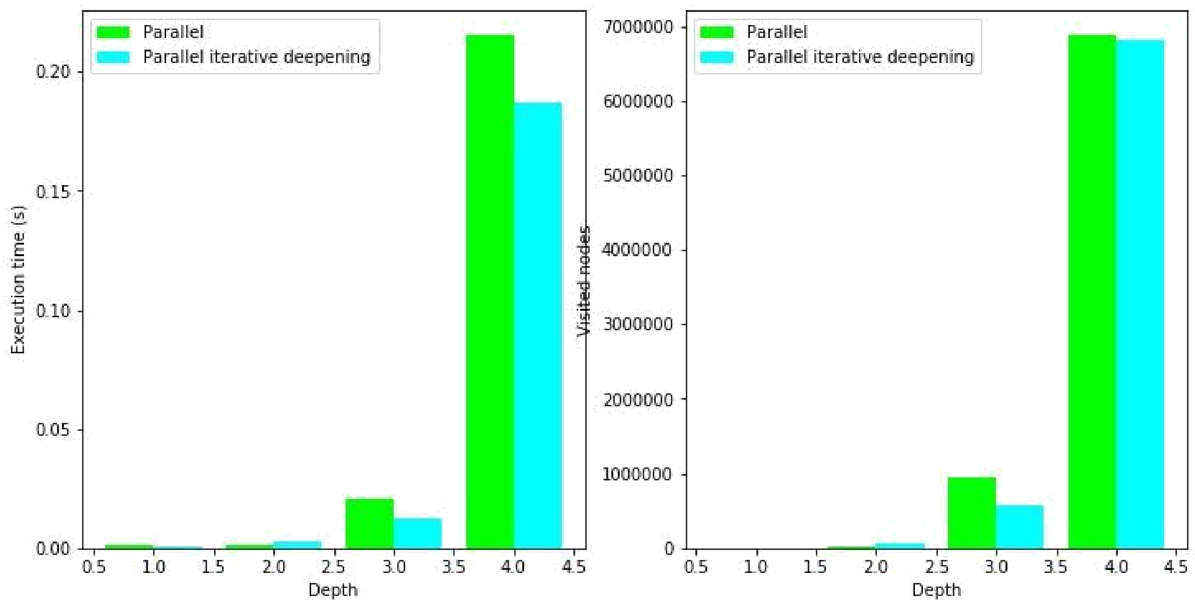


Fig. Parallel vs Parallel iterative deepening

From the above graphs it can be seen that the iterative deepening method explored slightly less nodes than the regular minimax search. However, the iterative deepening method explores the same nodes at many depths, so there is a lot of redundancy, and our implementation increments the number of explored nodes even if the node being explored has been seen already at a different depth.

So even if many nodes are explored many times, the total number of explored nodes (with repetitions) is less than the previous version.

This gives an idea about the huge reduction of the search tree size by using iterative deepening. This is what made it run faster, because otherwise it would be a lot slower that the original one, since we explore the nodes many times, and each time, re-do many of the computation that has been done already.

CONCLUSION

The chess engine is a decent built game with minimal chess-specific techniques and majorly focused on parallelizing the whole searching with backtracking algorithms like the minimax algorithm. The parallelizing algorithms used gave mixed outcomes when compared pair wise. One expected result was the better performance of root splitting parallelizing being better than the sequential process. This was obtained through the first simulation. However, move reordering did not give a promising outcome due to the additional execution time required for reordering. Although beam search gave a good performance with respect to execution time, the possibility of losing good moves, renders it non optimal. The iterative deepening technique had a very less run time, due to reduced computations and effective removal of redundancy. These mixed outcomes signify that parallelizing technique depends on the purpose and main focus of the game. for fast execution times we can use iterative deepening, for not optimal players we can use beam search and for optimal we can use beam search algorithm. Any parallelizing algorithm can be used and they will give a better performance than the sequential approach. The future scope of this project is to implement other chess-specific techniques to make the game more dynamic.

REFERENCES

- https://fr.slideshare.net/JamesSwafford2/jfsmasters1
- https://www.duo.uio.no/bitstream/handle/10852/53769/master.pdf?sequence=1
- https://pdfs.semanticscholar.org/8e1c/9f70aa4849475199e26ccd3e21c662e2d801.pdf
- https://chessprogramming.wikispaces.com/Iterative+Deepening

MAIN CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>
#include<sys/time.h>


#define MAX +1
#define MIN -1
#define INFINI INT_MAX

#define KGRN "\x1B[32m"
#define KRED "\x1B[31m"
#define KWHT "\x1B[37m"


struct config {
        char mat[8][8];
        int val;
        char xrN, yrN, xrB, yrB;
        char roqueN, roqueB;

};


int PawnIsolated = -30;

int PawnDouble = -30;

int RookOpenFile = 70;

int RookSemiOpenFile = 50;


int QueenOpenFile = 50;

int QueenSemiOpenFile = 30;

int BishopPair = 50;

int PawnTable[64] = {
        0       ,    0     , 0   , 0   , 0   ,      0       ,     0       ,     0
        ,
        50      ,       50      , 0     , -10   , -10   ,       0       ,       50      ,       50
        ,
        25      ,       0       ,       0       ,       25      ,       25      ,       0       ,
        0       ,       25      ,
        0       ,       0       ,       50      ,       90      ,       90      ,       50      ,
        0       ,       0       ,
        25      ,       25      ,       25      ,       50      ,       50      ,       25      ,
        25      ,       25      ,
        50  ,   50      ,       50      ,       90      ,       90      ,       50      ,       50
        ,       50      ,
        90      ,   90  ,       90      ,       90      ,       90      ,       90      ,       90
        ,       90      ,
        0       ,       0       ,       0       ,       0       ,       0       ,       0       ,
        0       ,       0
};

int KnightTable[64] = {
        0     ,         -50     ,       0       ,       0       ,       0       ,       0       ,
        -50   ,         0       ,
        0     ,         0       ,       0       ,       25      ,       25      ,       0       ,
        0       ,       0       ,
        0       ,       0       ,       50      ,       50      ,       50      ,       50      ,
        0       ,       0       ,
        0       ,       25      ,       50      ,       90      ,       90      ,       50      ,
        25      ,       0       ,
        25      ,       50      ,       70      ,       90      ,       90      ,       70      ,
        50      ,       25      ,
        25      ,       50      ,       50      ,       90      ,       90      ,       50      ,
        50      ,       25      ,
        0       ,       0       ,       25      ,       50      ,       50      ,       25      ,
        0       ,       0       ,
        0       ,       0       ,       0       ,       0       ,       0       ,       0       ,
        0       ,       0
};

int BishopTable[64] = {
        0       ,       0       ,       -50     ,       0       ,       0       ,       -50     ,
        0       ,       0       ,
        0       ,       0       ,       0       ,       50      ,       50      ,       0       ,
        0       ,       0       ,
        0       ,       0       ,       50      ,       70      ,       70      ,       50      ,
        0       ,       0       ,
        0       ,       50      ,       70      ,       90      ,       90      ,       70      ,
        50      ,       0       ,
        0       ,       50      ,       70      ,       90      ,       90      ,       70      ,
        50      ,       0       ,
        0       ,       0       ,       50      ,       70      ,       70      ,       50      ,
        0       ,       0       ,
        0       ,       0       ,       0       ,       50      ,       50      ,       0       ,
        0       ,       0       ,
        0           ,       0         ,         0           ,       0           ,       0           ,       0         ,
        0           ,       0
};

int RookTable[64] = {
   0       ,          0            ,      25       ,          50          ,      50          ,      25           ,       0
        ,       0            ,
        0           ,       0           ,       25          ,       50          ,       50          ,       25          ,
        0           ,       0         ,
        0           ,       0           ,       25          ,       50          ,       50          ,       25          ,
        0           ,       0           ,
        0           ,       0           ,       25          ,       50          ,       50          ,       25        ,
        0           ,       0         ,
        0           ,       0           ,       25          ,       50          ,       50          ,       25          ,
        0           ,       0           ,
        0           ,       0           ,       25          ,       50          ,       50          ,       25          ,
        0           ,       0           ,
        90          ,       90          ,       90          ,       90          ,       90          ,       90          ,
        90          ,       90          ,
        0           ,       0           ,       25          ,       50          ,       50          ,       25          ,
        0           ,       0
};


int KingE[64] = {
        -500    , -50           ,       0         ,         0         ,         0         ,         0         ,         -50
        ,       -500      ,
        -50    ,  0            ,       50          ,       50          ,       50          ,       50          ,       0
        ,         -50     ,
        0           ,       50          ,       90          ,       90          ,       90          ,       90          ,
        50          ,       0           ,
        0           ,       50          ,       90          ,       99          ,       99          ,       90          ,
        50          ,       0           ,
        0           ,       50          ,       90          ,       99          ,       99          ,       90          ,
        50          ,       0           ,
        0           ,       50          ,       90          ,       90          ,       90          ,       90          ,
        50          ,       0           ,
        -50    ,  0            ,       50          ,       50          ,       50          ,       50          ,       0
        ,       -50           ,
        -500      ,         -50         ,       0           ,       0           ,       0           ,       0           ,
        -50         ,       -500
};


int KingO[64] = {
        0           ,       25          ,       25          ,       -50    ,  -50        ,         0           ,       50
        ,       25            ,
        -300      ,         -300      ,         -300      ,         -300      ,         -300      ,         -300      ,
        -300      ,         -300      ,
        -500      ,         -500      ,         -500      ,         -500      ,         -500      ,         -500      ,
        -500      ,         -500      ,
	-700	,	-700	,	-700	,	-700	,	-700	,	-700	,
	-700	,	-700	,
	-700	,	-700	,	-700	,	-700	,	-700	,	-700	,
	-700	,	-700	,
	-700	,	-700	,	-700	,	-700	,	-700	,	-700	,
	-700	,	-700	,
	-700	,	-700	,	-700	,	-700	,	-700	,	-700	,
	-700	,	-700	,
	-700	,	-700	,	-700	,	-700	,	-700	,	-700	,
	-700	,	-700
};


int dC[8][2] = { {-2,+1} , {-1,+2} , {+1,+2} , {+2,+1} , {+2,-1} , {+1,-2} , {-1,-2} , {-2,-1} };

int D[8][2] = { {+1,0} , {+1,+1} , {0,+1} , {-1,+1} , {-1,0} , {-1,-1} , {0,-1} , {+1,-1} };


int minmax_ab( struct config conf, int mode, int niv, int min, int max, long * nb_noeuds, long *
nb_coupes);



void copier( struct config *c1, struct config *c2 )
{
	int i, j;

	for (i=0; i<8; i++)
		for (j=0; j<8; j++)
			c2->mat[i][j] = c1->mat[i][j];

	c2->val = c1->val;
	c2->xrB = c1->xrB;
	c2->yrB = c1->yrB;
	c2->xrN = c1->xrN;
	c2->yrN = c1->yrN;

	c2->roqueB = c1->roqueB;
	c2->roqueN = c1->roqueN;
}



int egal(char c1[8][8], char c2[8][8] )
{
	int i, j;

	for (i=0; i<8; i++)
		for (j=0; j<8; j++)
			if (c1[i][j] != c2[i][j]) return 0;
	return 1;
```

```
} // egal

int nbrPieces(struct config board, bool type) /// Compter le nombre des piÃ©ces
{
   int i, j, nbr = 0;
   if(type)
   {
      for(i = 0 ; i <= 7 ; i++)
      {
         for(j = 0 ; j <= 7 ; j++)
         {
            if(board.mat[i][j] > 0)
            {
               nbr ++;
            }
         }
      }
   }
   else
   {
      for(i = 0 ; i <= 7 ; i++)
      {
         for(j = 0 ; j <= 7 ; j++)
         {
            if(board.mat[i][j] < 0)
            {
               nbr ++;
            }
         }
      }
   }
   return nbr;
}


int estim(struct config board) /// La fonction d'evaluation de la configuration
{
   int i, j;
   int matrice = 0;
   int isole = 0, rowB, rowN, nbrPionB = 0, nbrPionN = 0, doubl = 0;
   bool pionPosB_1 = false, pionPosB_2 = false, pionPosB = false;
   bool pionPosN_1 = false, pionPosN_2 = false, pionPosN = false;
   int k, rockB_nbrOpen = 0, rockN_nbrOpen = 0, rockValue = 0;
   int queenN_nbrOpen = 0, queenB_nbrOpen = 0, queenValue = 0;
   int nbrBishopB = 0, nbrBishopN = 0, bishopValue = 0;
   int resultat, materiel=0;
   for(j = 0 ; j <= 7 ; j++)
   {
      for(i = 0 ; i <= 7 ; i++)
      {
         switch(board.mat[i][j])
```

```
{
   case 'p':
      materiel += 100;

      matrice += PawnTable[j + i * 8];

      nbrPionB ++;
      pionPosB = true;
      if(!pionPosB_1)
      {
         if(!pionPosB_2)
         {
            pionPosB_2 = true;
            rowB = j;
         }
         else
         {
            if (rowB != j)
            {
               pionPosB_1 = true;
            }
         }
      }
   break;
   case -'p':
      materiel -= 100;

      matrice -= PawnTable[j + (7 - i) * 8];

      nbrPionN ++;
      pionPosN = true;
      if(!pionPosN_1)
      {
         if(!pionPosN_2)
         {
            pionPosN_2 = true;
            rowN = j;
         }
         else
         {
            if (rowN != j)
            {
               pionPosN_1 = true;
            }
         }
      }
   break;
   case 'C':
      materiel += 300;

      matrice += KnightTable[j + i * 8];
```

```
break;
case -'C':
   materiel -= 300;

   matrice -= KnightTable[j + (7 - i) * 8];
break;
case 'f':
   materiel += 325;

   matrice += BishopTable[j + i * 8];

   nbrBishopB ++;
break;
case -'f':
   materiel -= 325;

   matrice -= BishopTable[j + (7 - i) * 8];

   nbrBishopN ++;
break;
case 't':
   materiel += 500;

   matrice += RookTable[j + i * 8];

   k = 0;
   while((k <= 7) && (board.mat[k][j] != 'p'))
   {
      if(((board.mat[k][j] == 0) || (board.mat[k][j] == 't')) || (board.mat[k][j] < 0))
      {
         rockB_nbrOpen ++;
      }
      k++;
   }
break;
case -'t':
   materiel -= 500;

   matrice -= RookTable[j + (7 - i) * 8];

   k = 7;
   while((k >= 0) && (board.mat[k][j] != -'p'))
   {
      if(((board.mat[k][j] == 0) || (board.mat[k][j] == -'t')) || (board.mat[k][j] < 0))
      {
         rockN_nbrOpen ++;
      }
      k --;
   }
break;
case 'n':
```

```
            materiel += 1000;
            k = 0;
            while((k <= 7) && (board.mat[k][j] != 'p'))
            {
               if(((board.mat[k][j] == 0) || (board.mat[k][j] == 'n')) || (board.mat[k][j] < 0))
               {
                  queenB_nbrOpen ++;
               }
               k++;
            }
         break;
         case -'n':
            materiel -= 1000;
            k = 7;
            while((k >= 0) && (board.mat[k][j] != -'p'))
            {
               if(((board.mat[k][j] == 0) || (board.mat[k][j] == -'n')) || (board.mat[k][j] < 0))
               {
                  queenN_nbrOpen ++;
               }
               k --;
            }
         break;
         case 'r':
            if(nbrPieces(board, true) > 8)
            {
               matrice += KingO[j + i * 8];
            }
            if(nbrPieces(board, true) < 7)
            {
               matrice += KingE[j + i * 8];
            }
         break;
         case -'r':
            if(nbrPieces(board, true) > 8)
            {
               matrice -= KingO[j + (7 - i) * 8];
            }
            if(nbrPieces(board, true) < 7)
            {
               matrice -= KingE[j + (7 - i) * 8];
            }
         break;
      }
   }

   if(nbrPionB > 0)
   {
      doubl = doubl + nbrPionB - 1;
   }
   if(nbrPionN > 0)
```

```
{
   doubl = doubl + nbrPionN - 1;
}
nbrPionB = 0;
nbrPionN = 0;

if(!pionPosB && !pionPosB_1 && pionPosB_2)
{
   isole ++;
}
if(!pionPosB)
{
   pionPosB_1 = false;
   pionPosB_2 = false;
}
pionPosB = false;

nbrPionN = 0;
if(!pionPosN && !pionPosN_1 && pionPosN_2)
{
   isole --;
}
if(!pionPosN)
{
   pionPosN_1 = false;
   pionPosN_2 = false;
}
pionPosN = false;

if(rockB_nbrOpen == 8)
{
   rockValue += RookOpenFile;
}
else
{
   if(rockB_nbrOpen > 5)
   {
      rockValue += RookSemiOpenFile;
   }
}
if(rockN_nbrOpen == 8)
{
   rockValue -= RookOpenFile;
}
else
{
   if(rockN_nbrOpen > 5)
   {
      rockValue -= RookSemiOpenFile;
   }
}
```

```
        rockB_nbrOpen = 0;
        rockN_nbrOpen = 0;

        if(queenB_nbrOpen == 8)
        {
           queenValue += QueenOpenFile;
        }
        else
        {
           if(queenB_nbrOpen > 5)
           {
              queenValue += QueenSemiOpenFile;
           }
        }
        if(queenN_nbrOpen == 8)
        {
           queenValue -= QueenOpenFile;
        }
        else
        {
           if(queenN_nbrOpen > 5)
           {
              queenValue -= QueenSemiOpenFile;
           }
        }
        queenB_nbrOpen = 0;
        queenN_nbrOpen = 0;

        if(nbrBishopB == 2)
        {
           bishopValue += BishopPair;
           nbrBishopB = 0;
        }
        if(nbrBishopN == 2)
        {
           bishopValue -= BishopPair;
           nbrBishopN = 0;
        }
     }
     resultat = materiel + doubl * PawnDouble + isole * PawnIsolated + rockValue + queenValue +
  bishopValue + matrice;

     return resultat;
  }
   // estim

  /************************************************************/
  /*********** Partie:  Evaluations et Estimations ***********/
  /************************************************************/
```

```
/* Teste s'il n'y a aucun coup possible dans la configuration conf */
int AucunCoupPossible( struct config conf )
{
        // ... A completer pour les matchs nuls
        // ... vÃ©rifier que generer_succ retourne 0 configurations filles ...
        return 0;

} // AucunCoupPossible


/* Teste si conf reprÃ©sente une fin de partie et retourne dans 'cout' la valeur associÃ©e */
int feuille( struct config conf, int *cout )
{
        //int i, j, rbx, rnx, rby, rny;

        *cout = 0;

        // Si victoire pour les Noirs cout = -100
        if ( conf.xrB == -1 ) {
          *cout = -100;
          return 1;
        }

        // Si victoire pour les Blancs cout = +100
        if ( conf.xrN == -1 ) {
          *cout = +100;
          return 1;
        }

        // Si Match nul cout = 0
        if (  conf.xrB != -1 &&  conf.xrN != -1 && AucunCoupPossible( conf ) )
          return 1;

        // Sinon ce n'est pas une config feuille
        return 0;

}  // feuille



/***********************************************************/
/*********** Partie:  GÃ©nÃ©ration des Successeurs ***********/
/***********************************************************/


/* GÃ©nÃ¨re dans T les configurations obtenues Ã  partir de conf lorsqu'un pion atteint la limite de
l'Ã©chiq */
void transformPion( struct config conf, int a, int b, int x, int y, struct config T[], int *n )
{
        int signe = +1;
        if (conf.mat[a][b] < 0 ) signe = -1;
        copier(&conf, &T[*n]);
        T[*n].mat[a][b] = 0;
        T[*n].mat[x][y] = signe *'n';
        (*n)++;
        copier(&conf, &T[*n]);
        T[*n].mat[a][b] = 0;
        T[*n].mat[x][y] = signe *'c';
        (*n)++;
        copier(&conf, &T[*n]);
        T[*n].mat[a][b] = 0;
        T[*n].mat[x][y] = signe *'f';
        (*n)++;
        copier(&conf, &T[*n]);
        T[*n].mat[a][b] = 0;
        T[*n].mat[x][y] = signe *'t';
        (*n)++;

} // transformPion


// VÃ©rifie si la case (x,y) est menacÃ©e par une des piÃ¨ces du joueur 'mode'
int caseMenaceePar( int mode, int x, int y, struct config conf )
{
        int i, j, a, b, stop;

        // menace par le roi ...
        for (i=0; i<8; i += 1) {
          // traitement des 8 directions paires et impaires
          a = x + D[i][0];
          b = y + D[i][1];
          if ( a >= 0 && a <= 7 && b >= 0 && b <= 7 )
                if ( conf.mat[a][b]*mode == 'r' ) return 1;
        } // for

        // menace par cavalier ...
        for (i=0; i<8; i++)
          if ( x+dC[i][0] <= 7 && x+dC[i][0] >= 0 && y+dC[i][1] <= 7 && y+dC[i][1] >= 0 )
                if ( conf.mat[ x+dC[i][0] ] [ y+dC[i][1] ] * mode == 'c' )
                  return 1;

        // menace par pion ...
        if ( (x-mode) >= 0 && (x-mode) <= 7 && y > 0 && conf.mat[x-mode][y-1]*mode == 'p' )
          return 1;
        if ( (x-mode) >= 0 && (x-mode) <= 7 && y < 7 && conf.mat[x-mode][y+1]*mode == 'p' )
          return 1;

        // menace par fou, tour ou reine ...
        for (i=0; i<8; i += 1) {
          // traitement des 8 directions paires et impaires
          stop = 0;
          a = x + D[i][0];
```

```
            b = y + D[i][1];
            while ( !stop && a >= 0 && a <= 7 && b >= 0 && b <= 7 )
                    if ( conf.mat[a][b] != 0 )  stop = 1;
                    else {
                       a = a + D[i][0];
                       b = b + D[i][1];
                    }
            if ( stop )  {
                    if ( conf.mat[a][b]*mode == 'f' && i % 2 != 0 ) return 1;
                    if ( conf.mat[a][b]*mode == 't' && i % 2 == 0 ) return 1;
                    if ( conf.mat[a][b]*mode == 'n' ) return 1;
            }
          } // for

          // sinon, aucune menace ...
          return 0;

} // caseMenaceePar


/* GÃ©nere dans T tous les coups possibles de la piÃ¨ce (de couleur N) se trouvant Ã  la pos x,y */
void deplacementsN(struct config conf, int x, int y, struct config T[], int *n )
{
          int i, j, a, b, stop;

          switch(conf.mat[x][y]) {
          // mvmt PION ...
          case -'p' :
                    //***printf("PION N Ã  la pos (%d,%d) \n", x,y);
                    if ( x > 0 && conf.mat[x-1][y] == 0 ) {                                    // avance d'une
case
                              copier(&conf, &T[*n]);
                              T[*n].mat[x][y] = 0;
                              T[*n].mat[x-1][y] = -'p';
                              (*n)++;
                              if ( x == 1 ) transformPion( conf, x, y, x-1, y, T, n );
                    }
                    if ( x == 6 && conf.mat[5][y] == 0 && conf.mat[4][y] == 0) {          // avance de 2 cases
                              copier(&conf, &T[*n]);
                              T[*n].mat[6][y] = 0;
                              T[*n].mat[4][y] = -'p';
                              (*n)++;
                    }
                    if ( x > 0 && y >0 && conf.mat[x-1][y-1] > 0 ) {                  // mange Ã  droite (en
descendant)
                              copier(&conf, &T[*n]);
                              T[*n].mat[x][y] = 0;
                              T[*n].mat[x-1][y-1] = -'p';
                              // cas oÃ¹ le roi adverse est pris...
                              if (T[*n].xrB == x-1 && T[*n].yrB == y-1) {
                                        T[*n].xrB = -1; T[*n].yrB = -1;
                              }

                              (*n)++;
                              if ( x == 1 ) transformPion( conf, x, y, x-1, y-1, T, n );
                    }
                    if ( x > 0 && y < 7 && conf.mat[x-1][y+1] > 0 ) {              // mange Ã  gauche (en
descendant)
                              copier(&conf, &T[*n]);
                              T[*n].mat[x][y] = 0;
                              T[*n].mat[x-1][y+1] = -'p';
                              // cas oÃ¹ le roi adverse est pris...
                              if (T[*n].xrB == x-1 && T[*n].yrB == y+1) {
                                        T[*n].xrB = -1; T[*n].yrB = -1;
                              }

                              (*n)++;
                              if ( x == 1 ) transformPion( conf, x, y, x-1, y+1, T, n );
                    }
                    break;

                // mvmt CAVALIER ...
                case -'c' :
                    for (i=0; i<8; i++)
                      if ( x+dC[i][0] <= 7 && x+dC[i][0] >= 0 && y+dC[i][1] <= 7 && y+dC[i][1] >= 0 )
                              if ( conf.mat[ x+dC[i][0] ] [ y+dC[i][1] ] >= 0 )  {
                                copier(&conf, &T[*n]);
                                T[*n].mat[x][y] = 0;
                                T[*n].mat[ x+dC[i][0] ][ y+dC[i][1] ] = -'c';
                                // cas oÃ¹ le roi adverse est pris...
                                if (T[*n].xrB == x+dC[i][0] && T[*n].yrB == y+dC[i][1]) {
                                        T[*n].xrB = -1; T[*n].yrB = -1;
                                }

                                (*n)++;
                              }
                    break;

                // mvmt FOU ...
                case -'f' :
                    for (i=1; i<8; i += 2) {
                      // traitement des directions impaires (1, 3, 5 et 7)
                      stop = 0;
                      a = x + D[i][0];
                      b = y + D[i][1];
                      while ( !stop && a >= 0 && a <= 7 && b >= 0 && b <= 7 ) {
                              if ( conf.mat[ a ] [ b ] < 0 )  stop = 1;
                              else {
                                copier(&conf, &T[*n]);
                                T[*n].mat[x][y] = 0;
                                if ( T[*n].mat[a][b] > 0 ) stop = 1;
                                T[*n].mat[a][b] = -'f';
```

```
                        // cas oÃ¹ le roi adverse est pris...
                        if (T[*n].xrB == a && T[*n].yrB == b) { T[*n].xrB = -1; T[*n].yrB = -1; }

                        (*n)++;
                        a = a + D[i][0];
                        b = b + D[i][1];
                      }
                  } // while
                } // for
                break;

        // mvmt TOUR ...
        case -'t' :
                for (i=0; i<8; i += 2) {
                  // traitement des directions paires (0, 2, 4 et 6)
                  stop = 0;
                  a = x + D[i][0];
                  b = y + D[i][1];
                  while ( !stop && a >= 0 && a <= 7 && b >= 0 && b <= 7 ) {
                      if ( conf.mat[ a ] [ b ] < 0 )  stop = 1;
                      else {
                        copier(&conf, &T[*n]);
                        T[*n].mat[x][y] = 0;
                        if ( T[*n].mat[a][b] > 0 ) stop = 1;
                        T[*n].mat[a][b] = -'t';
                        // cas oÃ¹ le roi adverse est pris...
                        if (T[*n].xrB == a && T[*n].yrB == b) { T[*n].xrB = -1; T[*n].yrB = -1; }

                        if ( conf.roqueN != 'e' && conf.roqueN != 'n' ) {
                          if ( x == 7 && y == 0 && conf.roqueN != 'p')
                                T[*n].roqueN = 'g'; // le grand roque ne sera plus possible
                          else if ( x == 7 && y == 0 )
                                  T[*n].roqueN = 'n'; // ni le grand roque ni le petit roque ne seront
possibles
                          if ( x == 7 && y == 7 && conf.roqueN != 'g' )
                                T[*n].roqueN = 'p'; // le petit roque ne sera plus possible
                          else if ( x == 7 && y == 7 )
                                  T[*n].roqueN = 'n'; // ni le grand roque ni le petit roque ne seront
possibles
                        }

                        (*n)++;
                        a = a + D[i][0];
                        b = b + D[i][1];
                      }
                  } // while
                } // for
                break;

        // mvmt REINE ...
        case -'n' :
```

```
                    for (i=0; i<8; i += 1) {
                      // traitement des 8 directions paires et impaires
                      stop = 0;
                      a = x + D[i][0];
                      b = y + D[i][1];
                      while ( !stop && a >= 0 && a <= 7 && b >= 0 && b <= 7 ) {
                              if ( conf.mat[ a ] [ b ] < 0 )  stop = 1;
                              else {
                                copier(&conf, &T[*n]);
                                T[*n].mat[x][y] = 0;
                                if ( T[*n].mat[a][b] > 0 ) stop = 1;
                                T[*n].mat[a][b] = -'n';
                                // cas oÃ¹ le roi adverse est pris...
                                if (T[*n].xrB == a && T[*n].yrB == b) { T[*n].xrB = -1; T[*n].yrB = -1; }

                                (*n)++;
                                a = a + D[i][0];
                                b = b + D[i][1];
                              }
                      } // while
                    } // for
                    break;

          // mvmt ROI ...
          case -'r' :
                    // vÃ©rifier possibilitÃ© de faire un roque ...
                    if ( conf.roqueN != 'n' && conf.roqueN != 'e' ) {
                      if ( conf.roqueN != 'g' && conf.mat[7][1] == 0 && conf.mat[7][2] == 0 &&
conf.mat[7][3] == 0 )
                        if ( !caseMenaceePar( MAX, 7, 1, conf ) && !caseMenaceePar( MAX, 7, 2, conf )
&& \
                                !caseMenaceePar( MAX, 7, 3, conf ) && !caseMenaceePar( MAX, 7, 4,
conf ) )  {
                              // Faire un grand roque ...
                              copier(&conf, &T[*n]);
                              T[*n].mat[7][4] = 0;
                              T[*n].mat[7][0] = 0;
                              T[*n].mat[7][2] = -'r'; T[*n].xrN = 7; T[*n].yrN = 2;
                              T[*n].mat[7][3] = -'t';
                              T[*n].roqueN = 'e'; // aucun roque ne sera plus possible Ã  partir de cette
config
                              (*n)++;
                        }
                      if ( conf.roqueN != 'p' && conf.mat[7][5] == 0 && conf.mat[7][6] == 0 )
                        if ( !caseMenaceePar( MAX, 7, 4, conf ) && !caseMenaceePar( MAX, 7, 5, conf )
&& \
                                !caseMenaceePar( MAX, 7, 6, conf ) )  {
                              // Faire un petit roque ...
                              copier(&conf, &T[*n]);
                              T[*n].mat[7][4] = 0;
                              T[*n].mat[7][7] = 0;
                              T[*n].mat[7][6] = -'r'; T[*n].xrN = 7; T[*n].yrN = 6;
                              T[*n].mat[7][5] = -'t';
                              T[*n].roqueN = 'e'; // aucun roque ne sera plus possible Ã  partir de cette
config
                              (*n)++;

                         }
                    }

                    // vÃ©rifier les autres mouvements du roi ...
                    for (i=0; i<8; i += 1) {
                      // traitement des 8 directions paires et impaires
                      a = x + D[i][0];
                      b = y + D[i][1];
                      if ( a >= 0 && a <= 7 && b >= 0 && b <= 7 )
                              if ( conf.mat[a][b] >= 0 ) {
                                copier(&conf, &T[*n]);
                                T[*n].mat[x][y] = 0;
                                T[*n].mat[a][b] = -'r'; T[*n].xrN = a; T[*n].yrN = b;
                                // cas oÃ¹ le roi adverse est pris...
                                if (T[*n].xrB == a && T[*n].yrB == b) { T[*n].xrB = -1; T[*n].yrB = -1; }

                                T[*n].roqueN = 'n'; // aucun roque ne sera plus possible Ã  partir de cette
config
                                (*n)++;
                              }
                    } // for
                    break;

          }

} // deplacementsN


/* GÃ©nere dans T tous les coups possibles de la piÃ¨ce (de couleur B) se trouvant Ã  la pos x,y */
void deplacementsB(struct config conf, int x, int y, struct config T[], int *n )
{
          int i, j, a, b, stop;

          switch(conf.mat[x][y]) {
          // mvmt PION ...
          case 'p' :
                    if ( x <7 && conf.mat[x+1][y] == 0 ) {                              // avance d'une
case
                              copier(&conf, &T[*n]);
                              T[*n].mat[x][y] = 0;
                              T[*n].mat[x+1][y] = 'p';
                              (*n)++;
                              if ( x == 6 ) transformPion( conf, x, y, x+1, y, T, n );
                    }
                    if ( x == 1 && conf.mat[2][y] == 0 && conf.mat[3][y] == 0) {          // avance de 2 cases
```

```
                        copier(&conf, &T[*n]);
                        T[*n].mat[1][y] = 0;
                        T[*n].mat[3][y] = 'p';
                        (*n)++;
                }
                if ( x < 7 && y > 0 && conf.mat[x+1][y-1] < 0 ) {               // mange Ã  gauche (en
montant)
                        copier(&conf, &T[*n]);
                        T[*n].mat[x][y] = 0;
                        T[*n].mat[x+1][y-1] = 'p';
                        // cas oÃ¹ le roi adverse est pris...
                        if (T[*n].xrN == x+1 && T[*n].yrN == y-1) {
                                T[*n].xrN = -1; T[*n].yrN = -1;
                        }

                        (*n)++;
                        if ( x == 6 ) transformPion( conf, x, y, x+1, y-1, T, n );
                }
                if ( x < 7 && y < 7 && conf.mat[x+1][y+1] < 0 ) {               // mange Ã  droite (en
montant)
                        copier(&conf, &T[*n]);
                        T[*n].mat[x][y] = 0;
                        T[*n].mat[x+1][y+1] = 'p';
                        // cas oÃ¹ le roi adverse est pris...
                        if (T[*n].xrN == x+1 && T[*n].yrN == y+1) {
                                T[*n].xrN = -1; T[*n].yrN = -1;
                        }

                        (*n)++;
                        if ( x == 6 ) transformPion( conf, x, y, x+1, y+1, T, n );
                }
                break;

        // mvmt CAVALIER ...
        case 'c' :
                for (i=0; i<8; i++)
                  if ( x+dC[i][0] <= 7 && x+dC[i][0] >= 0 && y+dC[i][1] <= 7 && y+dC[i][1] >= 0 )
                        if ( conf.mat[ x+dC[i][0] ] [ y+dC[i][1] ] <= 0 )  {
                          copier(&conf, &T[*n]);
                          T[*n].mat[x][y] = 0;
                          T[*n].mat[ x+dC[i][0] ][ y+dC[i][1] ] = 'c';
                          // cas oÃ¹ le roi adverse est pris...
                          if (T[*n].xrN == x+dC[i][0] && T[*n].yrN == y+dC[i][1]) {
                                T[*n].xrN = -1; T[*n].yrN = -1;
                          }

                          (*n)++;
                        }
                break;

        // mvmt FOU ...
        case 'f' :
                for (i=1; i<8; i += 2) {
                  // traitement des directions impaires (1, 3, 5 et 7)
                  stop = 0;
                  a = x + D[i][0];
                  b = y + D[i][1];
                  while ( !stop && a >= 0 && a <= 7 && b >= 0 && b <= 7 ) {
                        if ( conf.mat[ a ] [ b ] > 0 )  stop = 1;
                        else {
                          copier(&conf, &T[*n]);
                          T[*n].mat[x][y] = 0;
                          if ( T[*n].mat[a][b] < 0 ) stop = 1;
                          T[*n].mat[a][b] = 'f';
                          // cas oÃ¹ le roi adverse est pris...
                          if (T[*n].xrN == a && T[*n].yrN == b) { T[*n].xrN = -1; T[*n].yrN = -1; }

                          (*n)++;
                          a = a + D[i][0];
                          b = b + D[i][1];
                        }
                  } // while
                } // for
                break;

        // mvmt TOUR ...
        case 't' :
                for (i=0; i<8; i += 2) {
                  // traitement des directions paires (0, 2, 4 et 6)
                  stop = 0;
                  a = x + D[i][0];
                  b = y + D[i][1];
                  while ( !stop && a >= 0 && a <= 7 && b >= 0 && b <= 7 ) {
                        if ( conf.mat[ a ] [ b ] > 0 )  stop = 1;
                        else {
                          copier(&conf, &T[*n]);
                          T[*n].mat[x][y] = 0;
                          if ( T[*n].mat[a][b] < 0 ) stop = 1;
                          T[*n].mat[a][b] = 't';
                          // cas oÃ¹ le roi adverse est pris...
                          if (T[*n].xrN == a && T[*n].yrN == b) { T[*n].xrN = -1; T[*n].yrN = -1; }

                          if ( conf.roqueB != 'e' && conf.roqueB != 'n' ) {
                            if ( x == 0 && y == 0 && conf.roqueB != 'p')
                                T[*n].roqueB = 'g'; // le grand roque ne sera plus possible
                            else if ( x == 0 && y == 0 )
                                  T[*n].roqueB = 'n'; // ni le grand roque ni le petit roque ne seront
possibles
                            if ( x == 0 && y == 7 && conf.roqueB != 'g' )
                                T[*n].roqueB = 'p'; // le petit roque ne sera plus possible
                            else if ( x == 0 && y == 7 )
```

```
                                        T[*n].roqueB = 'n'; // ni le grand roque ni le petit roque ne seront
possibles
                               }

                               (*n)++;
                               a = a + D[i][0];
                               b = b + D[i][1];
                             }
                     } // while
                   } // for
                   break;

         // mvmt REINE ...
         case 'n' :
                   for (i=0; i<8; i += 1) {
                     // traitement des 8 directions paires et impaires
                     stop = 0;
                     a = x + D[i][0];
                     b = y + D[i][1];
                     while ( !stop && a >= 0 && a <= 7 && b >= 0 && b <= 7 ) {
                             if ( conf.mat[ a ] [ b ] > 0 )  stop = 1;
                             else {
                               copier(&conf, &T[*n]);
                               T[*n].mat[x][y] = 0;
                               if ( T[*n].mat[a][b] < 0 ) stop = 1;
                               T[*n].mat[a][b] = 'n';
                               // cas oÃ¹ le roi adverse est pris...
                               if (T[*n].xrN == a && T[*n].yrN == b) { T[*n].xrN = -1; T[*n].yrN = -1; }

                               (*n)++;
                               a = a + D[i][0];
                               b = b + D[i][1];
                             }
                     } // while
                   } // for
                   break;

         // mvmt ROI ...
         case 'r' :
                   // vÃ©rifier possibilitÃ© de faire un roque ...
                   if ( conf.roqueB != 'n' && conf.roqueB != 'e' ) {
                     if ( conf.roqueB != 'g' && conf.mat[0][1] == 0 && conf.mat[0][2] == 0 &&
conf.mat[0][3] == 0 )
                         if ( !caseMenaceePar( MIN, 0, 1, conf ) && !caseMenaceePar( MIN, 0, 2, conf )
&& \
                               !caseMenaceePar( MIN, 0, 3, conf ) && !caseMenaceePar( MIN, 0, 4, conf )
)  {
                             // Faire un grand roque ...
                             copier(&conf, &T[*n]);
                             T[*n].mat[0][4] = 0;
                             T[*n].mat[0][0] = 0;
```

```
                        T[*n].mat[0][2] = 'r'; T[*n].xrB = 0; T[*n].yrB = 2;
                        T[*n].mat[0][3] = 't';
                        T[*n].roqueB = 'e'; // aucun roque ne sera plus possible Ã  partir de cette
config
                        (*n)++;
                   }
                  if ( conf.roqueB != 'p' && conf.mat[0][5] == 0 && conf.mat[0][6] == 0 )
                    if ( !caseMenaceePar( MIN, 0, 4, conf ) && !caseMenaceePar( MIN, 0, 5, conf )
&& \
                          !caseMenaceePar( MIN, 0, 6, conf ) )  {
                        // Faire un petit roque ...
                        copier(&conf, &T[*n]);
                        T[*n].mat[0][4] = 0;
                        T[*n].mat[0][7] = 0;
                        T[*n].mat[0][6] = 'r'; T[*n].xrB = 0; T[*n].yrB = 6;
                        T[*n].mat[0][5] = 't';
                        T[*n].roqueB = 'e'; // aucun roque ne sera plus possible Ã  partir de cette
config
                        (*n)++;

                   }
                }

                // vÃ©rifier les autres mouvements du roi ...
                for (i=0; i<8; i += 1) {
                  // traitement des 8 directions paires et impaires
                  a = x + D[i][0];
                  b = y + D[i][1];
                  if ( a >= 0 && a <= 7 && b >= 0 && b <= 7 )
                        if ( conf.mat[a][b] <= 0 ) {
                          copier(&conf, &T[*n]);
                          T[*n].mat[x][y] = 0;
                          T[*n].mat[a][b] = 'r'; T[*n].xrB = a; T[*n].yrB = b;
                          // cas oÃ¹ le roi adverse est pris...
                          if (T[*n].xrN == a && T[*n].yrN == b) { T[*n].xrN = -1; T[*n].yrN = -1; }

                          T[*n].roqueB = 'n'; // aucun roque ne sera plus possible Ã  partir de cette
config
                          (*n)++;
                        }
                } // for
                break;

        }

} // deplacementsB


/* GÃ©nÃ¨re les successeurs de la configuration conf dans le tableau T,
   retourne aussi dans n le nb de configurations filles gÃ©nÃ©rÃ©es */
void generer_succ( struct config conf, int mode, struct config T[], int *n )
```

```
{
	int i, j, k, stop;

	*n = 0;

	if ( mode == MAX )
	{				// mode == MAX
	  for (i=0; i<8; i++)
	    for (j=0; j<8; j++)
		   if ( conf.mat[i][j] > 0 )
		     deplacementsB(conf, i, j, T, n );

	  // vÃ©rifier si le roi est en echec, auquel cas on ne garde que les succ Ã©vitants
l'Ã©chec...
	  for (k=0; k < *n; k++) {
		  i = T[k].xrB; j = T[k].yrB;  // pos du roi B dans T[k]
		  // vÃ©rifier s'il est menacÃ© dans la config T[k] ...
		  if ( caseMenaceePar( MIN, i, j, T[k] ) ) {
		     T[k] = T[(*n)-1];		// alors supprimer T[k] de la liste des succ...
		     (*n)--;
		     k--;
		  }
	   } // for k
	}

	else
	{						// mode == MIN
	  for (i=0; i<8; i++)
	    for (j=0; j<8; j++)
		   if ( conf.mat[i][j] < 0 )
		     deplacementsN(conf, i, j, T, n );

	  // vÃ©rifier si le roi est en echec, auquel cas on ne garde que les succ Ã©vitants
l'Ã©chec...
	  for (k=0; k < *n; k++)
	  {
		  i = T[k].xrN; j = T[k].yrN;
		  // vÃ©rifier s'il est menacÃ© dans la config T[k] ...
		  if ( caseMenaceePar( MAX, i, j, T[k] ) ) {
		     T[k] = T[(*n)-1];		// alors supprimer T[k] de la liste des succ...
		     (*n)--;
		     k--;
		  }
	  } // for k
	} // if (mode == MAX) ... else ...

} // generer_succ



/***********************************************************************/
```

```
/*********** Partie:  AlphaBeta, Initialisation et affichahe ***********/
/**********************************************************************/
int comp (const void * elem1, const void * elem2)
{
   struct config f = *((struct config *)elem1);
   struct config s = *((struct config *)elem2);
   if (f.val < s.val) return  1;
   if (f.val > s.val) return -1;
   return 0;
}

void trier_config_table(struct config T [], int n)
{
          int i;
          for (i = 0; i < n; i++)
                    T[i].val = estim(T[i]);

          qsort(T, n, sizeof(T[0]), comp);
          return;
}


/* MinMax avec elagage alpha-beta + Hill Climbing*/
int minmax_ab2( struct config conf, int mode, int niv, int alpha, int beta, long * nb_noeuds, long *
nb_coupes)
{

          int n, i, score, score2;
          struct config T[100];
          *nb_noeuds += 1;

          if ( feuille(conf, &score) )
                    return score;

          if ( niv == 0 )
                    return estim(conf);

          if ( mode == MAX )
          {

            generer_succ( conf, MAX, T, &n );
            trier_config_table(T, n);

            score = alpha;
            for ( i=0; i<n/2; i++ )
            {
                    score2 = minmax_ab2( T[i], MIN, niv-1, score, beta, nb_noeuds, nb_coupes);
                              if (score2 > score) score = score2;
                              if (score > beta)
                              {
                                        // Coupe Beta
                                        *nb_coupes += 1;
                                        //printf("Beta %d\n", beta);
                                        return beta;
                              }
          }
        }
        else
        { // mode == MIN

          generer_succ( conf, MIN, T, &n );
          trier_config_table(T, n);

          score = beta;
          for ( i=n-1; i >=n/2; i--)
          {
                score2 = minmax_ab2( T[i], MAX, niv-1, alpha, score, nb_noeuds, nb_coupes);
                              if (score2 < score) score = score2;
                              if (score < alpha)
                              {
                                   // Coupe Alpha
                                        *nb_coupes += 1;
                                        //printf("Alpha %d\n", alpha);
                                        return alpha;
                }
          }
        }

        return score;

} // minmax_ab

int iterative_deepening(struct config conf, int mode, int niv, int alpha, int beta, long * nb_noeuds,
long * nb_coupes)
{
        int n,j, i, score, score2;
        struct config T[100];

        if ( feuille(conf, &score) )
                return score;

        if ( niv == 0 )
                return estim(conf);

        if (mode == MAX)
        {

                generer_succ(conf, MAX, T, &n);

                for (i = 0; i < niv; i++)
                {
                        for (j = 0; j < n; j++)
```

```
                                {
                                        T[j].val =  minmax_ab(T[j], MIN, i, alpha, beta, nb_noeuds,
nb_coupes);
                                }

                                trier_config_table(T, n);

                        }

                        score = T[0].val;
                }
                else //MIN
                {
                        generer_succ(conf, MIN, T, &n);

                        for (i = 0; i < niv; i++)
                        {
                                for (j = n-1; j >= 0; j--)
                                {
                                        T[j].val =  minmax_ab(T[j], MAX, i, alpha, beta, nb_noeuds,
nb_coupes);
                                }

                                trier_config_table(T, n);
                        }

                        score = T[n-1].val;
                }
                return score;
}

/* MinMax avec elagage alpha-beta */
int minmax_ab( struct config conf, int mode, int niv, int alpha, int beta, long * nb_noeuds, long *
nb_coupes)
{

        int n, i, score, score2;
        struct config T[100];
        *nb_noeuds += 1;

        if ( feuille(conf, &score) )
                return score;

        if ( niv == 0 )
                return estim(conf);

        if ( mode == MAX )
        {

          generer_succ( conf, MAX, T, &n );
```

```
              score = alpha;
              for ( i=0; i<n; i++ )
              {
                     score2 = minmax_ab( T[i], MIN, niv-1, score, beta, nb_noeuds, nb_coupes);
                              if (score2 > score) score = score2;
                              if (score > beta)
                              {
                                       // Coupe Beta
                                                *nb_coupes += 1;
                                                //printf("Beta %d\n", beta);
                                                return beta;
                              }
              }
            }
            else
            { // mode == MIN

              generer_succ( conf, MIN, T, &n );

              score = beta;
              for ( i=0; i<n; i++ )
              {
                     score2 = minmax_ab( T[i], MAX, niv-1, alpha, score, nb_noeuds, nb_coupes);
                              if (score2 < score) score = score2;
                              if (score < alpha)
                              {
                                       // Coupe Alpha
                                                *nb_coupes += 1;
                                                //printf("Alpha %d\n", alpha);
                                                return alpha;
                     }
              }
            }

            if ( score == +INFINI ) score = +100;
     if ( score == -INFINI ) score = -100;
            return score;

} // minmax_ab

int minmax( struct config conf, int mode, int niv)
{
            int n, i, score, score2;
            struct config T[100];

            if ( feuille(conf, &score) )
                     return score;

            if ( niv == 0 )
                     return estim( conf );
```

```
        if ( mode == MAX ) {

          generer_succ( conf, MAX, T, &n );

          score = -INFINI;
          for ( i=0; i<n; i++ )
          {
                score2 = minmax( T[i], MIN, niv-1);
                        if (score2 > score) score = score2;

          }
        }
        else  { // mode == MIN

          generer_succ( conf, MIN, T, &n );

          score = +INFINI;
          for ( i=0; i<n; i++ ) {
                score2 = minmax( T[i], MAX, niv-1);
                if (score2 < score) score = score2;

          }
        }

    if ( score == +INFINI ) score = +100;
    if ( score == -INFINI ) score = -100;

        return score;

}
/* Intialise la disposition des pieces dans la configuration initiale conf */
void init( struct config *conf )
{
        int i, j;

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        conf->mat[i][j] = 0;        // Les cases vides sont initialisÃ©es avec 0

        conf->mat[0][0] =  't'; conf->mat[0][1] =  'c'; conf->mat[0][2] = 'f'; conf->mat[0][3] =  'n';
        conf->mat[0][4] =  'r'; conf->mat[0][5] =  'f'; conf->mat[0][6] = 'c'; conf->mat[0][7] =  't';

        for (j=0; j<8; j++) {
                conf->mat[1][j] = 'p';
                conf->mat[6][j] = -'p';
                conf->mat[7][j] = -conf->mat[0][j];
        }

        conf->xrB = 0; conf->yrB = 4;
        conf->xrN = 7; conf->yrN = 4;
```

```
		conf->roqueB = 'r';
		conf->roqueN = 'r';

		conf->val = 0;

	} // init

	/* Affiche la configuration conf */
	void affich( struct config conf )
	{
		system("clear");
		int i, j, k;
		for (i=0;  i<8; i++)
			printf("\t  %c", i+'a');
		printf("\n");

		for (i=0;  i<8; i++)
			printf("\t----- ");
		printf("\n");

		for(i=8; i>0; i--)  {
			printf("    %d", i);
			for (j=0; j<8; j++)
				if ( conf.mat[i-1][j] < 0 ) {
					printf("\t %s-%c",KRED, -conf.mat[i-1][j]);
					printf("%s", KWHT);
				}
				else if ( conf.mat[i-1][j] > 0 ) {
					printf("\t%s +%c",KGRN,  conf.mat[i-1][j]);
					printf("%s", KWHT);
				}
				    else printf("\t  ");
			printf("\n");

			for (k=0;  k<8; k++)
				printf("\t----- ");
			printf("\n");

		}
		printf("\n");

	} // affich




	/*******************************************/
	/*********** Programme princiapl ***********/
	/*******************************************/
```

```
int main( int argc, char *argv[] )
{
	char sy, dy, ch[10];
	int sx, dx, n, i, j, score, stop, cout, cout2, legal, hauteur, sauter;
	int cmin, cmax, mode, cpt;
	double stats1[100];
	double stats2[100];
	long nb_noeuds1=0;
	long nb_noeuds2=0;
	long nb_coupes1 = 0;
	long nb_coupes2 = 0;


  struct config T[100], conf, conf1;

  if ( argc == 1 )
	hauteur = 5;  // par dÃ©faut on fixe la profondeur d'Ã©valuation Ã  4
  else
	hauteur = atoi( argv[1] ); // sinon elle est rÃ©cupÃ©rÃ©e depuis la ligne de commande

  printf("\n\nProfondeur d'exploration = %d\n\n", hauteur);

  // Initialise la configuration de dÃ©part
  init( &conf );



  // Boucle principale du dÃ©rouleÃ¹ment d'une partie ...
  stop = 0;
  mode = MAX;

  struct timeval begin, end;
  int alpha, beta;
  double result;
  cpt = 0;
  nb_noeuds1=0;
  nb_noeuds2=0;
  long local_nb_noeuds1 = 0, local_nb_coupes1= 0, local_nb_coupes2 = 0, local_nb_noeuds2 = 0;

  while (!stop && (cpt < 50))
  {
	  alpha= -INFINI;
		beta= +INFINI;

		affich( conf );
		printf("Tour: %d\n", cpt);

		generer_succ(conf, mode, T, &n);

		score = -INFINI*mode;
		j = -1;
		//Timing
		gettimeofday(&begin, NULL);
		#pragma omp parallel private (local_nb_noeuds2, local_nb_coupes2,
local_nb_noeuds1, local_nb_coupes1) if (mode == MIN)
		{

			#pragma omp for  schedule (dynamic)
			for (i=0; i<n; i++)
			{

				if (mode == MAX)
				{

					local_nb_coupes1 = 0;
					local_nb_noeuds1 = 0;
					//printf("thread %d iteration %d\nnb_noeuds: %ld \
nlocal_nb_noeuds: %ld\n\n",omp_get_thread_num(),i, nb_noeuds1, local_nb_noeuds1);
					cout = minmax_ab(T[i], MIN, hauteur-1, alpha, beta,
&local_nb_noeuds1, &local_nb_coupes1);
					#pragma omp critical
					{
						if (cout > score)
						{
							alpha =  cout;
							score = cout;
							j = i;
						}
						nb_noeuds1 += local_nb_noeuds1;
						nb_coupes1 += local_nb_coupes1;

					}
				//printf("thread %d iteration %d\nnb_noeuds: %ld \
nlocal_nb_noeuds: %ld\n\n",omp_get_thread_num(),i, nb_noeuds1, local_nb_noeuds1);

				}
				else
				{
					local_nb_coupes2 = 0;
					local_nb_noeuds2 = 0;
					cout = minmax_ab2(T[i], MAX, hauteur-1, alpha, beta,
&local_nb_noeuds2, &local_nb_coupes2);

					#pragma omp critical
					{
						if ( cout < score )
						{
							beta = cout;
							score = cout;
							j = i;
						}
```

```
                                                        nb_coupes2 += local_nb_coupes2;
                                                        nb_noeuds2 += local_nb_noeuds2;
                                                }

                                        }

                                }
                        }
                        gettimeofday(&end, NULL);
                        result = (double)(end.tv_usec - begin.tv_usec)/1000000 + end.tv_sec - begin.tv_sec;
                        if (mode == MAX)
                                stats1[cpt] = result;
                        else
                        {
                                stats2[cpt] = result;
                                cpt++;
                        }
                        //Fin timing 2


                        if ( j != -1 )
                        { // jouer le coup et aller Ã  la prochaine itÃ©ration ...
                                copier( &T[j], &conf );
                                conf.val = score;
                                //printf("score: %d\n", conf.val);
                        }
                        else
                        { // S'il n'y a pas de successeur possible, l'ordinateur Ã  perdu
                                printf(" *** J'ai perdu ***\n");
                                stop = 1;
                        }
                  mode *= -1;

        } // while

                int iiii;
                char res[30];
                FILE * f = fopen("results.txt", "w");

                snprintf(res, 30, "%ld", nb_noeuds1);
                fputs(res, f);
                fputs("\n", f);

                snprintf(res, 30, "%ld", nb_noeuds2);
                fputs(res, f);
                fputs("\n", f);



                for (iiii = 0;iiii < cpt; iiii++)
                {
```

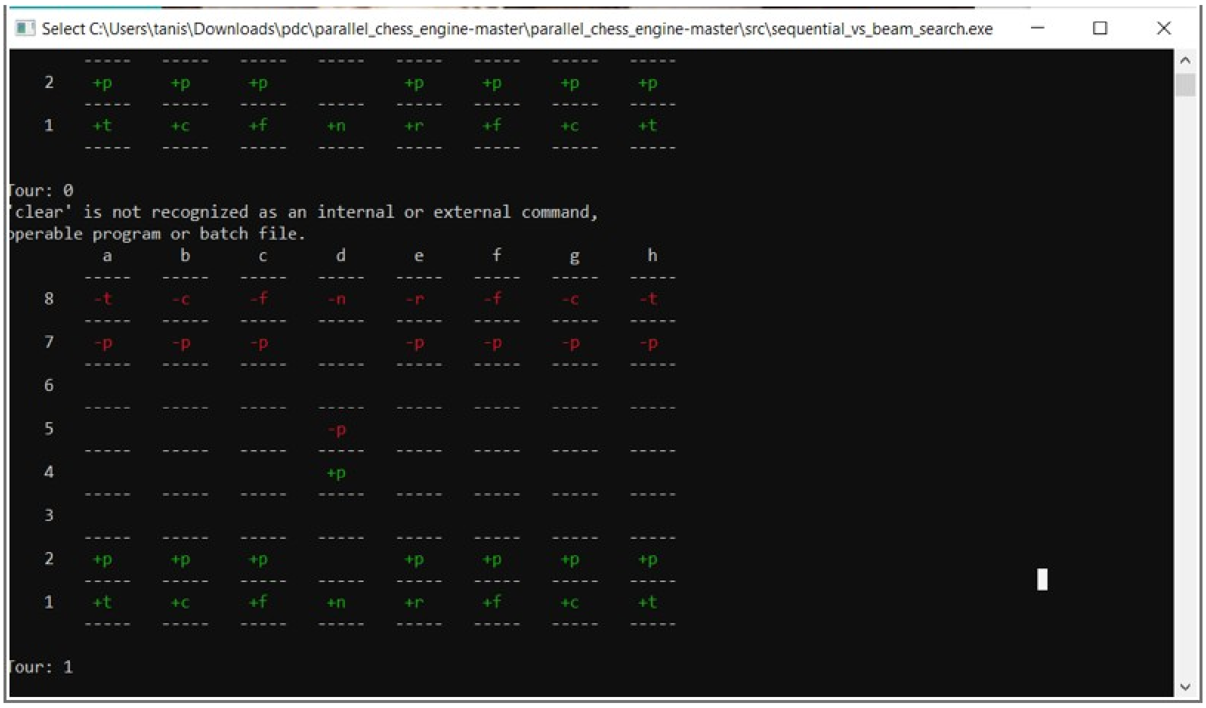
```
                snprintf(res, 30, "%f", stats1[iiii]);
                fputs(res, f);
                fputs(" ", f);
        }

        fputs("\n", f);
        for (iiii = 0;iiii < cpt; iiii++)
        {
                snprintf(res, 30, "%f", stats2[iiii]);
                fputs(res, f);
                fputs(" ", f);
        }

        fclose(f);

        return 0;
}
```

SAMPLE SCREENSHOT

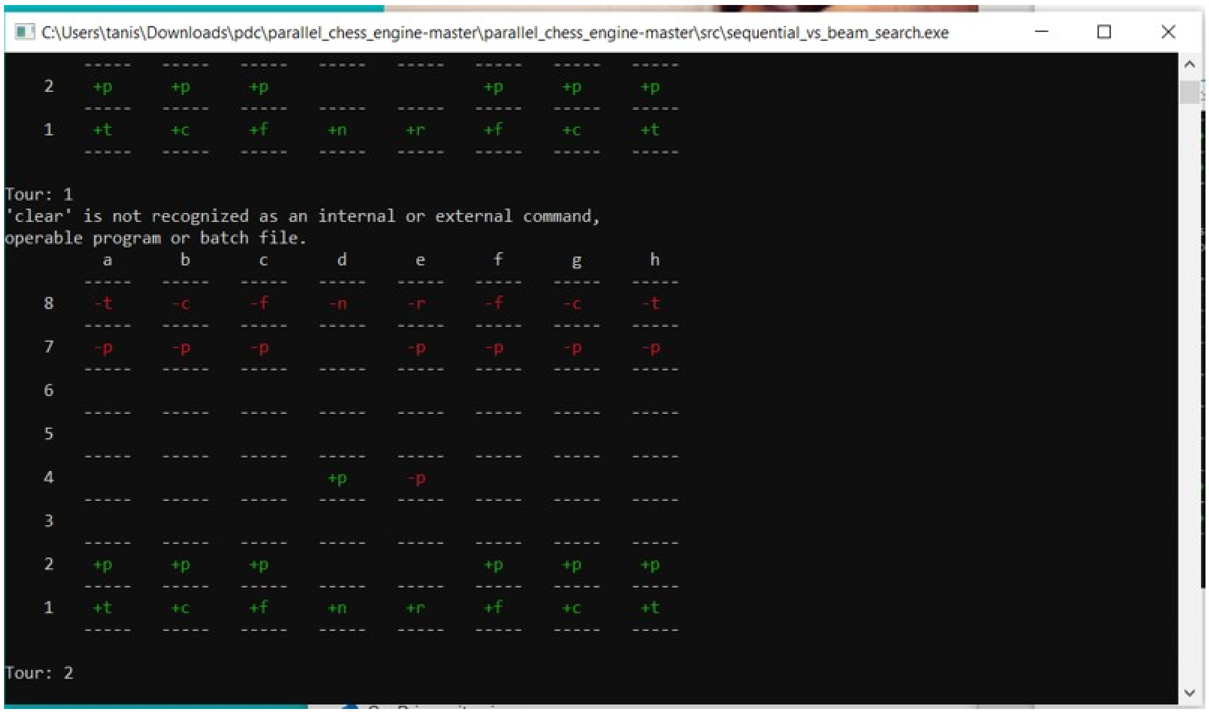


Select C:\Users\tanis\Downloads\pdc\parallel_chess_engine-master\parallel_chess_engine-master\src\sequential_vs_beam_search.exe

```
        -----   -----   -----   -----   -----   -----   -----   -----
    2    +p      +p      +p              +p      +p      +p      +p
        -----   -----   -----   -----   -----   -----   -----   -----
    1    +t      +c      +f      +n      +r      +f      +c      +t
        -----   -----   -----   -----   -----   -----   -----   -----

Tour: 0
'clear' is not recognized as an internal or external command,
operable program or batch file.
          a       b       c       d       e       f       g       h
        -----   -----   -----   -----   -----   -----   -----   -----
    8    -t      -c      -f      -n      -r      -f      -c      -t
        -----   -----   -----   -----   -----   -----   -----   -----
    7    -p      -p      -p              -p      -p      -p      -p
        -----   -----   -----   -----   -----   -----   -----   -----
    6
        -----   -----   -----   -----   -----   -----   -----   -----
    5                            -p
        -----   -----   -----   -----   -----   -----   -----   -----
    4                            +p
        -----   -----   -----   -----   -----   -----   -----   -----
    3
        -----   -----   -----   -----   -----   -----   -----   -----
    2    +p      +p      +p              +p      +p      +p      +p
        -----   -----   -----   -----   -----   -----   -----   -----
    1    +t      +c      +f      +n      +r      +f      +c      +t
        -----   -----   -----   -----   -----   -----   -----   -----

Tour: 1
```



C:\Users\tanis\Downloads\pdc\parallel_chess_engine-master\parallel_chess_engine-master\src\sequential_vs_beam_search.exe

```
        -----   -----   -----   -----   -----   -----   -----   -----
    2    +p      +p      +p                      +p      +p      +p
        -----   -----   -----   -----   -----   -----   -----   -----
    1    +t      +c      +f      +n      +r      +f      +c      +t
        -----   -----   -----   -----   -----   -----   -----   -----

Tour: 1
'clear' is not recognized as an internal or external command,
operable program or batch file.
          a       b       c       d       e       f       g       h
        -----   -----   -----   -----   -----   -----   -----   -----
    8    -t      -c      -f      -n      -r      -f      -c      -t
        -----   -----   -----   -----   -----   -----   -----   -----
    7    -p      -p      -p              -p      -p      -p      -p
        -----   -----   -----   -----   -----   -----   -----   -----
    6
        -----   -----   -----   -----   -----   -----   -----   -----
    5
        -----   -----   -----   -----   -----   -----   -----   -----
    4                            +p      -p
        -----   -----   -----   -----   -----   -----   -----   -----
    3
        -----   -----   -----   -----   -----   -----   -----   -----
    2    +p      +p      +p                      +p      +p      +p
        -----   -----   -----   -----   -----   -----   -----   -----
    1    +t      +c      +f      +n      +r      +f      +c      +t
        -----   -----   -----   -----   -----   -----   -----   -----

Tour: 2
```