Heuristic Alpha-Beta Search for Chess Game

Meng Zhou

December 16, 2014

1 Introduction

A chess game is a search problem that involves search strategy and evaluation function. In the development of computer chess games, the main purpose is developing algorithms that can defeat human players. Perhaps the most famous computer developed in the history to play chess game is Deep Blue by IBM. It won the game against world champion Garry Kasparov[1]. Since then, many researchers have improved computer chess algorithms and today even the best human player cannot win against artificial intelligence. Due to the complexity of the chess game it self, this project is mainly focus on the design and implementation of basic search algorithm and heuristic search strategies. AI to AI games will be set between our algorithm and other popular open source engines to test the performance of the algorithm.

2 Preliminary

A chess game is a two-player game. The players play alternatively and try their best to win the game. Besides developing search algorithms to make a move, some fundamental issues must be considered. Typically to play a chess game, one must understand the game, know the game rules and think moves to win the game. Thus a computer needs the following software components to play chess[5] :

- Data structure to represent chessboard in the memory so that the processor can access the status of the game.
- Algorithms to generate legal moves based on the status of the game.
- Search algorithms to choose from legal moves which one may lead to victory. This will include algorithms to evaluate chessboard status and search algorithms to quickly find the better or best solution.

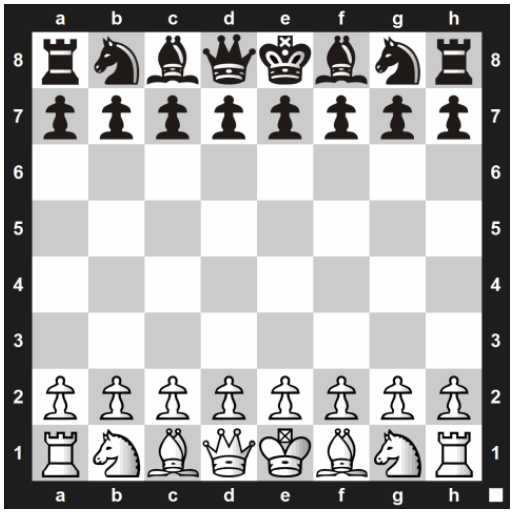
The focus of this project will be the search algorithm. However in order to implement a chess game, these components must be discussed first.

2.1 Chessboard Presentation

As it is presented in figure 1, the chessboard is a 8×8 square. The simplest representation of the board will be a 8×8 array. Each element of the array indicates which piece is present in the particular square(i.e. empty, wK, wN, wB, wR, wQ, wP, bK, bN, bB, bR, bQ). Algorithm 1 describes this representation.

```
1 for i:=0 to 7 do
2 |   for j:=0 to 7 do
3 |   |   value[square[i , j]] = chess piece on board[i, j];
4 |   end
5 end
```

Algorithm 1: 8×8 array representation



56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43	44	45	46	47
32	33	34	35	36	37	38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

Figure 1: ChessBoard

The basic 8×8 representation maybe easy to implement, but there are certain problems. When performing move generation, sometimes we need to test whether the piece is still on the board after a move. In order to do this we have to check the position with the boundary, which requires 4 comparison operations. We can extend the array to 10×10 and put special elements in the boundary squares.

A better solution is 0x88 representation illustrated in Figure 2. It uses an array of 128 to represent the board. Only half of the array corresponding to actual position on the board. The high 4 bits are the row and the low 4 bits are the column. As we can see in Figure 2, if we have a position i, then the square left of i this i-1, right is i+1, up is i+16, down is i-16. The advantages of this representation are

- Only one index is used.
- Testing whether a move stays on the board is simply doing $i\&0x88 == 0$

Another technique modern chess engines use is bitboards(see right of Figure 2)[3]. Instead of having an array of squares, each containing a piece types, have an array of piece types, each of which stores a packed array of bits listing the squares containing that piece. Since there are 64 possible squares, each of these packed arrays can be stored in a 64-bit number (two 32-bit words). The big advantage is that you can perform certain evaluation and move generation operations very quickly using bitwise Boolean operations.

112	113	114	115	116	117	118	119
96	97	98	99	100	101	102	103
80	81	82	83	84	85	86	87
64	65	66	67	68	69	70	71
48	49	50	51	52	53	54	55
32	33	34	35	36	37	38	39
16	17	18	19	20	21	22	23
0	1	2	3	4	5	6	7

1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

Figure 2: 0x88 and bit board representation

For example if we want to know squares occupied by black we can compute using

$$bOcc \;=\; bP \mid bN \mid bB \mid bR \mid bQ \mid bK$$

And the possible white pawn one-square move destinations can be computed by

$$pawn\_moves \;=\; (wP \;<<\; 8) \;\&\; \sim occupied$$

Bitboards can also be used to quickly exam whether a piece can be attacked by another. For example in figure 3 the left matrix is the squares rock can attack on d5, the middle matrix is the position of opponent's knight. By doing an "and" operation we know that the knight will be attacked.

2.2 Move Generator

The chess game is a complicated game, it got so many rules to deal with. So the move generator in a chess engine might be the most complicated part. Typically a status of the

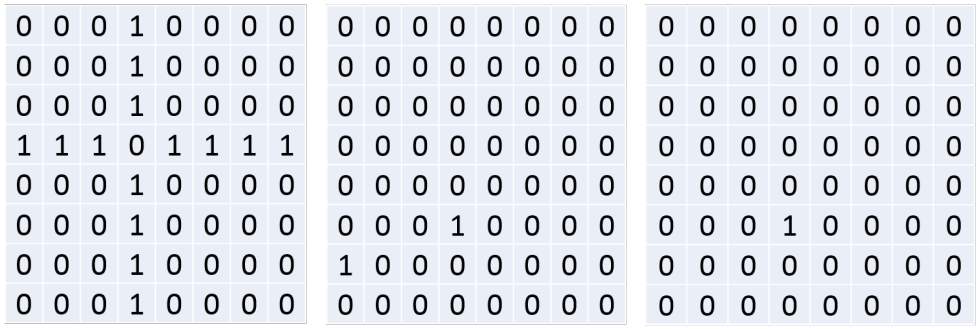


Figure 3: Bitboards representation

board may generate 30 legal moves. Historically there are three major strategies: selective generation, incremental generation and complete generation.

Selective generation can be dated back to the 1940s. Claude Shannon described two ways to build a chess-playing algorithm: Look at all possible moves, and all the possible moves resulting from each, recursively; Only examine the "best" moves and then only the "best" replies to each, recursively. The second one may seems to be better because humans think in this way and they do well in chess games. But engines using this strategy do not play very well[3]. The problem is that, for a best move generator to be any good, it has to be almost perfect. If an algorithm looks for 5 best moves given a status, and we assume that the best move is among those 5 at least 95% of the time. During a 40 move game, the probability that the generator's list will always contain the best choice at all time is $95\%^{40} = 13\%$. This means even if the generator select the best move at a very high possibility at each step, the overall outcome may not be good.

A full-width searching consists of[5]:

- Finding all of the legal moves available in a position.
- Ordering them in some way, hopefully speeding up search by picking an advantageous order
- Searching them all one at a time, until all moves have been examined or a cutoff occurs.

A simple implementation will be scanning the board one square at a time and looking for pieces of the moving side, and computing possible move destinations on the fly. A technique called transposition table can be used to accelerate the search: If one move has already been searched before, when encounter similar situations after that, the algorithm do not need to search it again. Such a table is a hash table of each of the positions analyzed so far up to a certain depth. The number of positions searched by a computer often greatly

exceeds the memory constraints of the system it runs on, thus not all positions can be stored. When the table fills up, less-used positions are removed to make room for new ones; this makes the transposition table a kind of cache.

Search efficiency is also depend on the order in which moves are searched. As we know it's not possible to search all possible solutions at a given time. Therefore try the "best" move at first may lead to better solution. This is a tricky question because if we know the best solution, why we have to search for it? In practice, algorithm can guess a better solution by sort by certain conditions. For example start with captures and pawn promotions may be a good idea.

2.3 Evaluation Function

The evaluation function is to let the algorithm know how good or bad is the current status. We can represent the quality of a status as a number, and the evaluation should be the same as the quality of the opponent measures. There are two major types of evaluation function method. End-point evaluation is to evaluate each status independently. Pre-computation stores a map of hash value of a status and the evaluation quality. When the current status is close enough, the algorithm simply use the stored value instead of calculate again.

There are multiple metrics that can be used to give a status a quality(a number).

- Material balance is an account of which pieces are on the board for each side. The weight of each piece is given in advance. For example a queen may be worth 100 and a pawn 10. Therefore material balance can be simply computed by $MB = Sum(Np \times Vp)$, where Np is the number of pieces of a certain type on the board and Vp is that piece's value. Most engines add more features to material balance. For example once you are ahead on material, exchanging pieces of equal value is advantageous. But using material balance as primitive is not common.

- Mobility is the characteristic that if a player has more pieces and more options to choose from, then the player can evaluate then and make better decision rather than limited to few choices. Count the possible moves to calculate mobility may be the simplest choice, but it's not good because many chess moves are pointless. Therefore develop mobility for certain piece may be a good choice. For example rock can move around the board is better than limited at bottom line.

- Development is to say that bishops and knights should be brought into the battle as quickly as possible and rock should stay quiet until it is time for a decisive attack. If the conditions are satisfied, the status may get hight points on this.

- Pawn formations: some people say that pawns are the soul of the game. If the pawns are moving tactical, for example doubled or tripled pawns movement and they take care of each other it may be an advantage.

3 Search Algorithm

The chess game is basically an algorithm searching for the move to win the game. Therefore it's a search problem, and the problem space is a search tree that covers all the possible moves. As discussed above, due to the limitation of time, the algorithm cannot search all the leaves. So it is a depth limited DFS algorithm. The search algorithm is the key to a chess engine and we will discuss the frame of this part.

3.1 Min-Max Search

In the general case, there are two players Min and Max playing the game.

- Max's job is to make moves that will increase the board's evaluation.
- Min's job is to make moves that decrease the board's evaluation.

Assume that both players play flawlessly.

We define a rooted game tree in which the nodes correspond to game status, and children of a node are status that can be reached from it in one move. Figure 4 illustrate the game tree with two players Min and Max we discussed above. In the game tree there are three kinds of nodes: internal nodes correspond to Min's move; internal nodes correspond to Max's move; Leaves correspond to status that game ends with one player. A max level node will choose the maximal value of its children as its value, and a min level node will choose the minimal value of its children as its value.

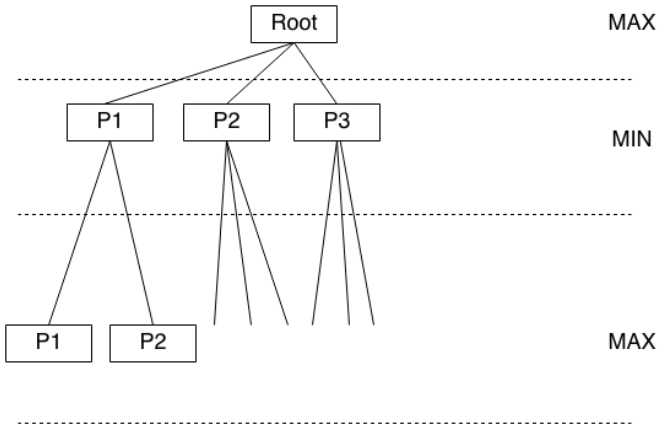


Figure 4: Game Tree

In order to make a move, the algorithm will select one of the children from the root node as the next move. This can be done by search the game tree. However, in a chess game, there might be 30 children for a node and 40 levels for the game tree, which makes

it impossible to search the tree thoroughly in limited time. To make a better choice, the algorithm can do a BFS stopping at given level, or DFS with maximum level. DFS is more space efficient because it does not need to store the whole game tree. Besides, we will see in the next section that some of the branches do not need to search at all. Algorithm 2 gives the pseudo code of Min-Max search. Figure 5 and 6 give example of how the algorithm update node value.

```
1  function minmax(node, depth, maximizingPlayer)
2  if depth = 0 Or node is a terminal node then
3  |  Return the heuristic value of node
4  end
5  if maximizingPlayer then
6  |  bestValue := −inf for each child of node do
7  |  |  val := minmax(child, depth-1, false)
8  |  |  bestValue := max(bestValue, val)
9  |  end
10 |  Return bestValue
11 end
12 else
13 |  bestValue := −inf
14 |  for each child of node do
15 |  |  val := minmax(child, depth-1, true)
16 |  |  bestValue := min(bestValue, val)
17 |  end
18 |  Return bestValue
19 end
```

Algorithm 2: Min-Max Search

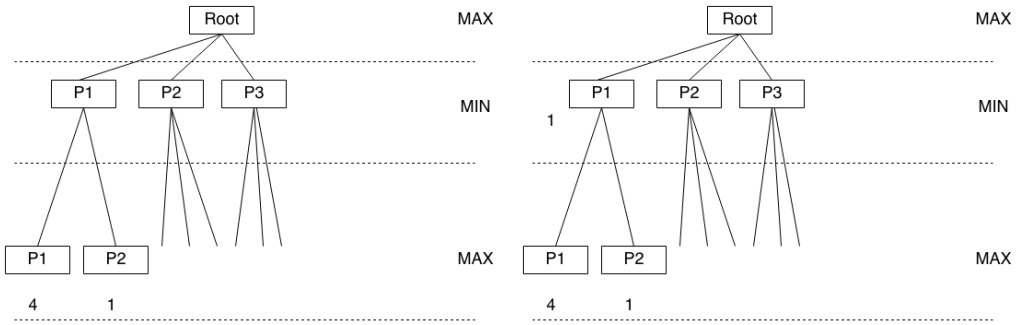


Figure 5: Min-Max algorithm example

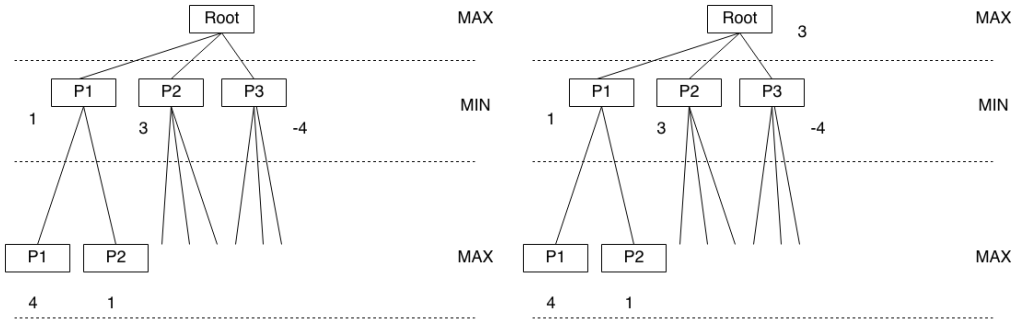


Figure 6: Min-Max algorithm example(cout.)

In order to analysis the size of the game tree, we make the assumption that each internal node has the same number of children b. The height of the game tree is limited to d. So the total number of nodes will be

$$1 + b + b^2 + b^3 + ... + b^{d-1} = O(b^d)$$

The complexity is exponential and we won't be able to search too many levels. Suppose $b = 35$, then search level 2 will visit 1225 nodes, search level 3 will visit 42875 nodes, search level 4 will visit 1500625 nodes. This greatly limits the depth of the game tree can reach, thus limits the sight of the algorithm.

3.2 Alpha-Beta Search

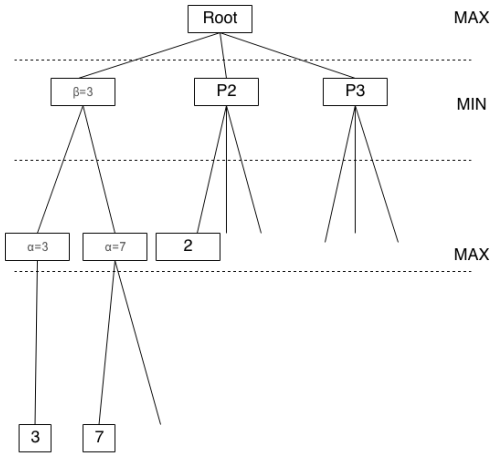


Figure 7: Beta pruning

In Alpha-Beta search, we keep two values for a node: alpha value is the infimum of the node can be considering its children, and beta value is the supremum of it. In figure 7, suppose we search the game tree from left to right. When we reach the MAX level node with $\alpha = 7$ and its parent's $\beta = 3$, we do not need to search any other of its children any more. Because no matter what value they have, this node will not report a value lower than 7. But the parent is a MIN node and already has another child reports 3, so the parent will never report a value greater than 3. This is called Beta Pruning.

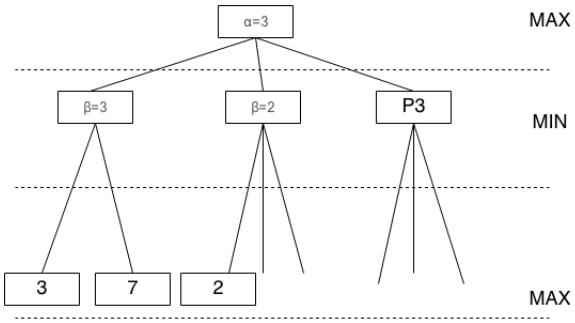


Figure 8: Alpha pruning

Correspondingly we have Alpha Pruning. In figure 8 when we reach the MIN level node with $\beta = 2$ and its parent's $\alpha = 3$, we do not need to search any other of its children any more. Because no matter what value they have, this node will not report a value greater than 2. But he parent is a MAX node and already has another child reports 3, so the parent will never repot a value smaller than 3.

The analysis of alpha-beta pruning is extracted from [3]. In the best case, each node at depth D-1 will only examine one child at depth D before pruning, except that one node on the principal variation will not prune. At depth D-2, however, nobody can prune, because all the children returned values greater than or equal to the values of beta they were passed, which at depth D-2 are negated and become less than or equal to alpha. Continuing up the tree, at depth D-3 everyone (except on the principal variation) prunes, and at depth D-4 nobody prunes, etc. So, if the branching factor of the tree is B, the number of nodes increases by a factor of B at half the levels of tree, and stays pretty much constant (ignoring the principal variation) at the other half of the levels. So the total size of the part of the tree that gets searched ends up being roughly $B^{D/2} = sqrt(B)^D$. Effectively, alpha-beta search ends up reducing the branching factor to the square root of its original value, and lets one search twice as deeply. For this reason it is an essential part of any minimax-based game-playing program.

```
function alphabeta(node, depth, α, β maximizingPlayer)
if depth = 0 Or node is a terminal node then
    Return the heuristic value of node
end
if maximizingPlayer then
    bestValue := − inf for each child of node do
        α := max(α, alphabeta(child, depth-1, α, β, false))
        if β ≤ α then
            Break
        end
    end
    Return α
end
else
    bestValue := − inf
    for each child of node do
        β := min(β, alphabeta(child, depth-1, α, β, true))
        if β ≤ α then
            Break
        end
    end
    Return β
end
```

Algorithm 3: Alpha-Beta Search

3.3 Accelerate Search Algorithms

There are several optimal methods that can improve performance of the Alpha Beta Search Algorithm. Permutation table saves branches that already searched so that when encounter a similar situation, it immediately report the evaluation value using hash table. But it might make the algorithm unstable due to the definition of similar situation.

Quiescent Search is also helpful for chess game. Due to the limitation of time, a program can search limited steps. However, some detrimental moves maybe happen after these steps and the computer cant see it. Usually experienced human players can see this and Quiescent Search is used to simulate this behavior.

Another improvement is Principal Variation Search (PVS)[6], which is also known as Minimal Window Search. When doing Alpha Beta Search, nodes can be categorized as Alpha Node, where value of child node is no more than alpha value, Beta Node, where value of child node is no greater than beta value, and PV node. Alpha and Beta node can be pruned immediately, but PV node has to be searched thoroughly. PVS makes

the assumption first that result value is no greater than Alpha Value and uses a minimal window to search PV node. If its true, then it saves a lot of time, otherwise it has to search the node with normal window again. In chess engines, this gives a 10 percent performance increase[5].

Finally Null-Move Forward Pruning[2] can make search program faster by cutting more useless branches. It first let the opponent make a move. If the value is no smaller than Beta Value, simply return Beta Value. Null Forward does not bring any instability and always report the optimal value.

4 Implementation

The main purpose of this project is implement a playable chess game, and focus on the search algorithm. So this project uses Cassandra, an open source project to generate possible moves.

There are many popular protocols in the chess industry that allow separation of engine and GUI. This project uses Winboard/XBoard protocol and GNU Winboard as GUI/engine interface. An screen shoot can be found in figure 9.

Figure 9: GUI screen shot

5 Experiments

In order to test the implementation, this project makes several AI to AI games between our algorithm with other famous open source engines such as Fair-Max and Fruit. We make several tests and record their steps and results, which can be found in the following form. The details of steps can be found in the appendix.

Engine	Black/White	Result	#Steps
Fairy	W	lose	18
	W	lose	19
	B	lose	12
	B	lose	9
Fruit	W	lose	17
	W	lose	12
	B	lose	18
	B	lose	14

As we can see in the experiment our engine lose all the competitions. The results indicate that when playing black side, our algorithm lose more quickly with Fairy. Compared with the famous and much more complicated engine, our implementation can't win the game. But it's still worth trying to improve our algorithm in the future.

References

[1] Wikipedia, Computer Chess.[Online; accessed 15-Dec-2014]

[2] C. Donninger. Null move and deep search: Selective-search heuristics for obtuse chess programs. ICCA Journal, page 137143, 1993

[3] David Eppstein. Strategy and board game programming, 2014. [Online; accessed 27-Oct-2014]

[4] Debasis Ganguly, Johannes Leveling, and Gareth J.F. Jones. Retrieval of similar chess positions. In Proceedings of the 37th International ACM SIGIR Conference on Research, SIGIR 14, pages 687-696, New York, 2014

[5] Franois Dominic Larame. Chess programming, 2000. [Online; accessed 27-Oct-2014].

[6] T. A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. ACM Comput. Surv., 14(4):533?551, December 1982.

[7] Von Neumann, J: Zur Theorie der Gesellschaftsspiele Math. Annalen. 100 (1928) 295-320

Engine	Black/White	Steps
Fairy	W	1. Nc3 e5 2. d3 c5 +0.18/8 3. Nb1 Nc6 +0.23/7 4. a3 Nf6 +0.16/7 5. b3 Be7 +0.14/8 6. Bf4 exf4 +2.87/9 7. c4 O-O +3.05/8 8. Nc3 d5 +3.03/8 9. a4 d4 +3.02/7 10. Nd5 Nxd5 +3.69/10 11. Rc1 Nc3 +6.09/9 12. f3 Bh4+ +12.51/10 13. Kd2 Nxd1 +12.42/10 14. Ra1 Qa5+ +79.94/11 15. Kc1 Qc3+ +79.96/15 16. Kxd1 Qxa1+ +79.97/28 17. Kd2 Be1+ +79.98/28 18. Kc2 Nb4# +79.99/28
	W	1. Nh3 c5 +0.16/8 2. c3 f5 +0.22/7 3. b3 Nc6 +0.15/8 4. g3 Nf6 -0.08/8 5. Ba3 e5 -0.13/8 6. Bxc5 Bxc5 +2.22/9 7. Qc2 d5 +2.26/8 8. Qxf5 Bxf5 +10.18/8 9. Kd1 Bxh3 +11.94/9 10. d3 Ng4 +14.03/11 11. Kc2 Nxf2 +13.74/10 12. Bg2 Bxg2 +16.80/10 13. Kd2 Nxh1 +21.40/9 14. Kc2 Bg1 +21.74/10 15. a4 Bxh2 +22.61/10 16. a5 Nxg3 +23.12/9 17. Kd2 Qg5+ +23.92/9 18. Kd1 Qe3 +25.90/9 19. Ke1 Qxe2# +79.99/28
	B	1. e4 a5 2. Nc3 +0.20/8 Nc6 3. Bd3 +0.12/8 Nf6 4. Nf3 +0.03/8 Ne5 5. Nxe5 +2.59/9 a4 6. O-O +2.72/7 b6 7. f4 +2.65/7 Ra7 8. Re1 +2.59/7 c5 9. Bb5 +3.04/8 Ra5 10. Bxa4 +3.35/7 Nh5 11. Bb3 +5.98/8 Ra3 12. Bxf7# +79.99/28
	B	1. e4 c5 2. c3 d5 3. exd5 a6 4. Qa4+ +0.67/8 b5 5. Bxb5+ +1.43/10 Nc6 6. dxc6 +4.66/8 Qd4 7. cxd4 +14.42/9 g6 8. c7+ +20.87/12 Bd7 9. Bxd7# +79.99/28
Fruit	W	1. a4 Nc6 +0.13/12 2. Ra2 e5 +0.85/12 3. e3 Nf6 +0.88/12 4. b3 Be7 +1.14/11 5. Ne2 d5 +1.30/12 6. d4 O-O +1.19/10 7. Na3 Ne4 +2.45/10 8. Nf4 Bb4+ +9.00/11 9. Ke2 Nc3+ +11.29/7 10. Ke1 Nxd1+ +11.50/8 11. Kxd1 exf4 +11.74/9 12. exf4 Qe7 +12.53/10 13. Bd2 Bxa3 +13.98/10 14. Be3 Bg4+ +15.36/10 15. Kd2 Qb4+ +20.37/11 16. Kd3 Qe1 +99.95/45 17. Ra1 Nb4# +99.99/43
	W	1. d4 Nf6 -0.18/12 2. b3 d5 +0.24/12 3. g4 Bxg4 +1.40/11 4. a4 c5 +1.84/10 5. Bh6 gxh6 +4.81/11 6. e3 Bxd1 +12.14/10 7. b4 Bxc2 +15.41/11 8. e4 dxe4 +18.01/10 9. Nc3 cxd4 +19.46/12 10. Nd1 d3 +23.09/11 11. a5 d2+ +99.97/40 12. Ke2 Qd3# +99.99/40
	B	1. Nf3 +0.10/13 a5 2. e4 +0.72/12 Nc6 3. d4 +0.89/12 Nf6 4. d5 +0.95/12 Nh5 5. dxc6 +3.42/10 dxc6 6. Qxd8+ +3.52/11 Kxd8 7. Nc3 +3.46/12 Bf5 8. exf5 +6.98/10 e6 9. Ng5 +6.96/11 b5 10. Nxf7+ +10.52/11 Kc8 11. Nxh8 +11.02/12 Kb8 12. fxe6 +13.09/12 b4 13. Ne4 +13.92/13 Bc5 14. Nxc5 +17.93/11 Ra6 15. e7 +28.67/11 Nf4 16. e8=Q+ +99.93/45 Ka7 17. Nxa6 +99.95/45 g5 18. Qb8# +99.99/42
	B	1. Nf3 +0.10/13 Nh6 2. e4 +0.74/12 a6 3. Nc3 +1.19/11 Nc6 4. d4 +1.35/11 Nxd4 5. Qxd4 +3.82/10 Ng4 6. h3 +3.83/11 e5 7. Nxe5 +4.29/11 Bc5 8. Qxc5 +5.78/11 f6 9. Nxg4 +10.66/9 Qe7 10. Qxc7 +11.69/11 Qe5 11. Nxe5 +25.24/9 h6 12. Bc4 +99.89/9 b5 13. Bf7+ +99.95/43 Kf8 14. Qd6# +99.99/42